Section Notes 3

Computer Science 50 — Fall 2008

Updated by: David Malan, Doug Lloyd, and Glenn Holloway

Week of October 5, 2008

Contents

1 Functions

1.1 The Purpose of Functions

Functions in the C programming language help you to partition your code into manageable pieces. A function is like a black box with a set of inputs and a single output. The rest of the program does not need to know what goes on inside the function, but the program can call the function, provide it with the inputs, and then use the output. In this sense, a C function performs much like a mathematical function on your calculator. If you wanted to find the sine of `x` you could use the `sin` function (found in `math.h`) without worrying about how it actually finds the sine.

```
double
sin_times_cos(double x, double y)
{
    return sin(x) * cos(y);
}
```

Such a function is used for the *value* it computes and returns.

But functions can also be useful for the *side effects* they cause. For example, you might write a function called `print_instructions()`, which has no return value, but executes a series of `printf()` statements. `printf()` itself is a function that is executed for its side effect.

The main reasons for using functions in programming are:

- Organization. Having good functional decomposition in a piece of code is like having a good outline for a paper. Functions help to break up a complicated problem into more manageable subparts, and they also help to make sure that concepts flow logically into one another.

- Simplification. Smaller components are easier to design, easier to implement, and far easier to debug. Good use of functions makes code easier to read and problems easier to isolate.

- Reusability. Functions only need to be written once, and then can be used as many times as necessary, so you can avoid duplication of code. For instance, to ask the user a yes/no question several times throughout a program, you might write a helper function called `ask_yes_no()` to handle the dialog.

1.2 The Anatomy of Functions

The shape of a C function should look familiar to you by now, because every complete C program has at least one function, `main()`.[1] Here's a little function named `accumulate_interest` taking two numbers, which might be a bank balance and an interest rate, and producing another number, an updated balance after interest has accrued.

```
double
accumulate_interest(double balance, double rate)
{
    double accrued;
    double updated;

    accrued = balance * rate;
    updated = balance + accrued;

    return updated;
}
```

This function *definition* has the following structure:

⟨*return type*⟩ ⟨*function name*⟩`(`⟨*arg1 type*⟩ ⟨*arg1 name*⟩`,` ⟨*arg2 type*⟩ ⟨*arg2 name*⟩`,` ... `)`   `{...}`

[1] The difference between `main()` and other functions is that `main()` isn't normally called within the program itself. Its arguments are provided by the operating system and its result goes back to the system as an indication of whether the program finished successfully.

A function definition has a *header* and a *body*. The header always contains these parts:

1. Return type: The type of value that a call of the function will produce. In our example, it is `double`. Knowing the return type of `accumulate_interest` allows the compiler to know for example, that the variable declaration

   ```
   double val = accumulate_interest(bal, rate);
   ```

   has a valid initializer.

2. Function name: The name that is used in the program to call this function. In this example, the name is `accumulate_interest`.

3. Parameter list: The parameters, also called the function's *arguments*, tell the compiler what types of values to expect when the function is called. In our example, the list `(double balance, double rate)` tells the compiler to expect two arguments, both of type `double`. The parameter list can be empty, which means that the function doesn't expect any arguments.[2]

The *body* of a function definition is a sequence of declarations and statements wrapped in braces ({`...`}). When the function is called, the body is executed to produce the function's side effects and ultimately to return its value (if it has one).

The variables declared in a function's body (such as `accrued` and `updated` in `accumulate_interest` above) are *local* to the function. Assignments to those variables have no effect on variables declared outside of the function, even if they happen to have the same names. The variables we have been declaring in `main()` are actually local variables

One or more of the statements in a function body can be `return` statements. A `return` statement ends the execution of the function and passes the return value back to the function that called it (the "caller"). In our example function, the `return` statement returns the value `updated` to the caller of `accumulate_interest`. The fact that a `return` statement terminates the function immediate can be a useful way to escape from nested loops and conditional statements without having to exit each of them individually.

If a function returns a value, then it must have at least one `return` statement. (It could have more than one, returning different values under different conditions.) If the function doesn't return a value, then the `return` statement is optional. The function will return after it executes the last statement in its body. Or it might contain a `return` that has no value expression, which returns control to the calling function without passing a value. A function that doesn't return a value must have the special return type `void`.

A function body doesn't have to be as verbose as the one for `accumulate_interest` above. An equivalent definition of can be written with just one statement:

```
double
accumulate_interest(double balance, double rate)
{
    return balance + (balance * rate);
}
```

A function definition gives all the information that the compiler needs about a function. It describes both how the function should be used and how its body should be translated into machine language that the computer can execute. But just as we sometimes need to separate the declaration of a variable from its initialization and its uses, we sometimes want to be able to declare a function without giving its full definition. A function *declaration* is the header part of the corresponding function definition, followed by a semicolon. For example, here is the declaration of our example function `accumulate_interest`:

```
double accumulate_interest(double balance, double rate);
```

[2] You may also see functions that have the special parameter list `(void)`. That's an explicit way of saying that the function accepts no arguments.

It matches the definition, except that the body has been replaced with a semicolon. Once the compiler has seen the declaration of a function, it can compile calls on the function without having to have seen the full definition. To be manageable, a large C program will always be broken into separately compilable pieces. It is important that the compiler be able to see the declarations for functions used by the piece that it happens to be compiling without having to see their full definitions. For example, when it compiles your calls to `printf()`, it only sees a declaration of `printf()`, not its body.[3]

It is also considered good style to declare all the functions to be defined within a `.c` file early on in that file, before any of the actual definitions. That way both the compiler knows in advance how each function call should be treated, and the reader of the code has a single place to look for a description of a function's interface with the functions that call it. But make sure that a function's declaration agrees exactly with the header of its definition!

1.3 The Function Call

Call a function using its name and the arguments you are passing to it.

```
int
main()
{
    double start_balance, interest_rate, final_balance;

    start_balance = GetDouble();
    interest_rate = GetDouble();

    final_balance = accumulate_interest(start_balance, interest_rate);

    printf("The final balance is %.2f.\n", final_balance);
}
```

Here the arguments to `accumulate_interest` (named `start_balance` and `interest_rate`) happen to be variables, but they can be constants or other expressions. Even if the arguments are variables, the function will make its own copies of their values, so it can't modify any variables in the function that called it by assigning new values to its parameters.

Note also that the privilege of calling functions is not exclusive to `main()`. Any function can call another function.

1.4 Variable Scope

Variables that are declared inside of a function exist only inside of that function. They are called *local variables*, in contrast to variables declared outside of all functions, which are called *global variables*. Global variables can be accessed and changed in any function. In the following example, `greater_num` cannot be accessed in `main()`, and `num1` cannot be accessed in `max()`. `i_am_global` can be changed from anywhere in the program.

```
/* Always comment your global variables!! */
int i_am_global;

int
main()
{
    int num1, num2, larger;

    num1 = 3;
    num2 = 4;
    larger = max(num1, num2);
    printf("%d is the larger number computed by max().\n", larger);
```

[3] `printf()` is declared in the file `stdio.h`, which the compiler sees because of a `#include` directive in your program.

```
    max2(num1, num2);
    printf("%d is the larger number from the global variable via max2().\n", i_am_global);
}

int
max(int x, int y)
{
    int greater_num;

    greater_num = (x > y ? x : y); // more on this construct later
    return greater_num;
}

void
max2(int x, int y)
{
    i_am_global = (x > y ? x : y);
}
```

Since `i_am_global` is declared outside of all functions, including `main()`, it is a global variable. It can be referenced and changed by any function. *Do not use global variables unless they are necessary or make your code significantly simpler.* When you use global variables, be certain to document them explicitly. It is usually better to pass variables between functions than to have a lot of global variables that are difficult to trace through the program. For instance, in the examples above, use of the function `max()` is better than use of `max2()`.

In C, function arguments are *passed by value*. That means that the function being called will make its own copy of the arguments, and will not be able to change the value of the arguments in the calling function. The function's parameters are therefore local variables of the function. They're different from other local variables only because they're initialized directly from the argument values supplied by the caller.

In the example below, when `increment()` begins execution, it also has a clean slate, and it can name its variables whatever it wants. While in `increment`, we can name variables without regard to the names being used in other functions. Consider this silly program:

```
int increment(int x); // function declaration

int
main()
{
    int x = 1;
    int y;

    y = increment(x);
    printf("%d  %d \n", x, y);
}

int
increment(int x)
{
    x++;
    return x;
}
```

In this example, both `main()` and `increment()` have variables named `x`, but the two are distinct. When `increment()` is called, it grabs new space in memory and puts a copy of `x` from `main()` into it. This means that `increment()` knows only about the value passed to it, but nothing about what it was named in the calling function, and nothing about where the calling function has it stored in memory. Therefore, when `increment()` executes the statement `x++;`, it is changing only the `x` that is local to itself. When `main()` receives the return value from `increment()`, it assigns the value to `y`. The printed result is:

```
 1 2
```

You should not use global variables where they can reasonably be avoided. Because they can be altered by every function that uses them, it can become difficult to keep track of their current value. In many cases, the use of a global variable can be avoided by passing the needed value to certain functions. However, if there is a value that legitimately needs to be seen by most of the functions in your program, it may be more efficient to use a global variable.

2 Don't Rely on Magic

Consider the following incomplete snippet of code:

```
int
main(int argc, char *argv[])
{
    string tf_array[30];

       . . .         // populate the array

    // print out the contents of the array
    for (i = 0; i < 30; i++)
    {
        printf("%s\n", tf_array[i]);
    }
}
```

You can probably deduce that the for loop above will, beginning at `tf_array[0]`, print out the contents of each element of the array `tf_array`. But why do we pick the number 30 and not a different limit? What makes 30 so special?

In the Mario part of Problem Set 1, you probably wrote a loop that relied on the number 23 in some way, to account for the maximum height of the pyramid based on the expected window size of the user who runs your program. But how is the reader of your code expected to know that? If it isn't clear from the outset why a certain number makes an appearance in a line of code, then it's probably a "magic" number.

Fortunately, C provides us with a built-in construct to eliminate the use of magic numbers within our code. It's the `#define` directive, and here's how the previous snippet could be rewritten to use it:

```
#define NUM_OF_TFS 30

int
main(int argc, char *argv[])
{
    string tf_array[NUM_OF_TFS];

       . . .         // populate the array

    // print out the contents of the array
    for (i = 0; i < NUM_OF_TFS; i++)
    {
        printf("%s\n", tf_array[i]);
    }
}
```

The identifier `NUM_OF_TFS` is *not* a variable. The compiler begins its work by replacing every instance of this identifier by the constant 30. So the effect of the change is not on the compiled program, it's on the reader, who can better understand the role of the number 30 in this program. The magic number isn't so magic anymore.

But why use a `#define` directive when we could instead have declared a variable:

```
    int num_of_tfs = 30;
```

The reason is that we want to make it clear both to the compiler and to readers of the program that this quantity can't be changed. The compiler can often produce better machine code with that knowledge. And the reader gets a more precise understanding of the program's behavior.

Other data types besides `int` can exploit the `#define` directive. In Section 3, we'll define a character-valued identifier called `SENTINEL`:

```
#define SENTINEL '0'
```

Not every literal constant in your programs needs to be given a `#define` directive. But if the purpose of the constant is worth documenting, using `#define` is a good way to do it.

3 Arrays

Arrays are data structures that can be be used to hold multiple values of the same type in contiguous memory locations. A handy analogy for understanding arrays is to think about your local Harvard Mail Center (whether it's in an upperclass house or the Science Center basement). An array is a block of contiguous space in memory (the bank of mailboxes) which has been partitioned into smaller identically-sized blocks of space (individual mailboxes). These smaller spaces can each store a certain amount of data (mail), which can be accessed directly by index number (mailbox number).

Here is an array and some statements accessing it:

```
    int votes[3];

    votes[0] = 10;  // stuffing the ballot box
    votes[1] = 0;
    votes[2] = 0;
```

Some key properties of C arrays:

- C arrays always start at index 0. If an array has `n` elements, its last element is at index `n-1`.
- Unlike other languages, C *will not* prevent programs from accessing elements past the end (or before the beginning) of arrays, even though this can cause catastrophic errors (or worse, small baffling errors). It is up to the programmer to make sure that programs never access elements outside the bounds of an array.
- Array names are not variables. Whole arrays cannot be assigned the same way that regular values are; the contents of array `foo` cannot be assigned to another array `bar` just by saying `bar = foo`. Instead, you must copy over each element, one at a time (usually with a loop).

3.1 Multi-Dimensional Arrays

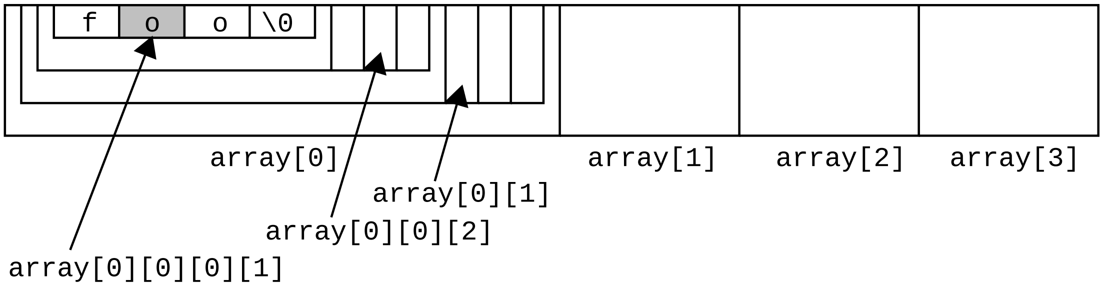
A multi-dimensional array is declared using more than on size specifiers. For example, to create a 3-by-3 array of integers to represent a tic-tac-toe board:

```
int board[3][3];
```

Access to multi-dimensional arrays is analogous to one-dimensional array access:

```
board[1][2] = 4;
```

The word "dimension" can be confusing in this context. C arrays can have any number of dimensions, unlike physical entities that are usually constrained to three dimensions. For two-dimensional arrays, it's best to consider the array to be a grid. Another analogy that may help you, when considering arrays of higher dimension (which aren't used too frequently) is to think about it sort of like Russian dolls. If you have a four-dimensional array in C, think of each element of the "outermost" array as containing another array, each element of one of those contains another array, et cetera, as in the following example:



This is only a partial cross-section of what an array declared as `char array[4][4][4][4];` would look like, of course. If we wanted to print the letter `o`, as highlighted in the picture, we could access it with the following statement:

```
printf("%c\n", array[0][0][0][1]);
```

This diagram may also be useful to you when considering how arrays are stored; remember that arrays are nothing more than consecutive, equal-sized "chunks" of data in memory. All that this boils down to is 256 ($4 \times 4 \times 4 \times 4$) side-by-side blocks of memory, each eight bits (one byte) wide, the size of a `char` in C. The way each block is accessible is merely an issue of syntax.

3.2 Arrays as Function Arguments

When a function is called with an array as an argument, a reference to the array is passed, not the entire array. This means that if the function being called changes the contents of the array, then the contents of the array seen by the calling function are changed as well. Note that this is quite different from the way that variables are handled. It appears to contradict the rule that a variable defined in a function can't be modified just by passing it as an argument to another function. But arrays are not variables.

The basis for the behavior of arrays in C will become much clearer when we discuss *pointers* later in the semester.

Consider the following code:

```
void
set_array(int array[])
{
    array[0] = 22;
}

void
set_int(int x)
{
    x = 22;
}

int
main()
{
   int a = 10;
   int b[4] = { 0, 1, 2, 3 };

   set_int(a);
   set_array(b);
   printf("%d %d\n", a, b[0]);
}
```

This program prints `10 22`. The call to `set_int` doesn't change `a` at all, but the call to `set_array` does change the contents of `b`.

3.3 Array Initialization

You can statically initialize arrays, that is, initialize each array element to its respective value in a list. One-dimensional statically-initialized arrays do not require you to give an explicit size (as in `four_elements[]` below) but for clarity, you may do so if you wish (as in `num_days_in_month[12]`). For multi-dimensional arrays, however, all dimensions except the first (the outermost array) must be declared explicitly.

```
int num_days_in_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
string four_elements[] = { "fire", "water", "wind", "earth" };
bool truth_table[2][3] = { { true, true, true }, { true, true, true } };
```

Below is an example of how a one-dimensional array might be used. This might be part of a program to keep track of a section's grades on a particular assignment:

```
#define CLASS_SIZE      30
#define EMPTY           0

void init_array(int array[], int size);

int
main()
{
    int i;
    int scores_array[CLASS_SIZE];

    init_array(scores_array, CLASS_SIZE);

    // populate the array
    for (i = 0; i < CLASS_SIZE; i++)
    {
        printf("Enter score for student %d: ", i);
        scores_array[i] = GetInt();
    }
}

void
init_array(int array[], int size)
{
    int i;

    // initialize all cells to 0
    for (i = 0; i < size; i++)
    {
        array[i] = EMPTY;
    }
}
```

This example illustrates several points about the use of arrays.

1. The `scores_array` is declared to be of size `30`. The indices run from `0` to `29`. Never "step" off the end of an array. The `for` loop stops before hitting the number `30`. If you were to use `scores_array[30]` you would probably get a segmentation fault.
2. Because arrays are passed by reference, the function `init_array()` does not receive a "local" copy of the array. Rather, it is told where in memory the array is located! So when `main()` passes the array, it is allowing `init_array()` to make any changes it wants.

Note that this is not the case if you pass an element of an array. If you were to call `foo(scores_array[i]);` you would not be sending the entire array, only one element. Since each element is simply an integer (in this case), this is just passing an integer to `foo()` and thus `foo()` gets a local copy.

3. Array names must have `[]` after them in the parameter list of a function declaration or definition.

   ```
   void init_array(int array[], int size);
   ```

   It is fine to put the size of the array in the declaration/definition. So this is also acceptable:

   ```
   void init_array(int array[CLASS_SIZE], int size);
   ```

   But specifying the size of a one-dimensional array parameter is optional, as is the size of the first dimension of a multi-dimensional array.

If we wanted to revise the program presented above to record scores for multiple assignments, we might use a two-dimensional array.

```
#define CLASS_SIZE      30
#define NUM_ASSIGNS     10
#define EMPTY           0

void init_2d_array(int array[][NUM_ASSIGNS], int num_rows, int num_cols);

int
main()
{
    int i,j;
    int scores_array[CLASS_SIZE][NUM_ASSIGNS];

    init_2d_array(scores_array, CLASS_SIZE, NUM_ASSIGNS);

    // loop through each student in the class
    for (i = 0; i < CLASS_SIZE; i++)
    {
        printf("Entering scores for student %d:\n", i);

        // loop through each assignment
        for (j = 0; j < NUM_ASSIGNS; j++)
        {
             printf("Enter score for assignment %d", j);
             scores_array[i][j] = GetInt();
        }
    }
}
```

```
void
init_2d_array(int array[CLASS_SIZE][NUM_ASSIGNS], int num_rows, int num_cols)
{
    int i,j;

    // set each assignment for each student to EMPTY
    for (i = 0; i < num_rows; i++)
    {
        for (j = 0; j < num_cols; j++)
        {
            array[i][j] = EMPTY;
        }
    }
}
```

You can declare arrays up to any reasonable number of dimensions. As shown in the example, you can declare an array of two dimensions with the declaration:

```
int scores_array[CLASS_SIZE][NUM_ASSIGNS];
```

The only other difference between one-dimensional and multi-dimensional arrays is in how you declare them as parameters to functions. In the function declaration and definition, the compiler requires an explicit size for each dimension except the first, which is optional.

3.4 An Example Program

Let’s write a program to count the number of occurrences for each alphabetic character in a given text input. Let’s also ignore cases, so that both ‘A’ and ‘a’ will count as ‘A’. Hints are below, but see if you can solve this on your own before reading them. It may also be helpful to look at an ASCII chart for ideas.

Hints: The simple step is realizing that we will need some sort of loop to read characters from input iteratively, and to know that we will need a sentinel to indicate when to stop reading. But how should we keep track of the counts? 26 different variables and a gigantic `if-else` statement sounds okay, but is there some convenient data structure to store these in? And, recalling that characters are really just integers, is there something clever we can do with the indices of our data structure?

Our implementation follows, but it has bugs! What’s wrong with it, and how could you fix it?

```
#include <stdio.h>
#include <ctype.h>

#define SENTINEL ’0’
#define ALPHABET_SIZE 26

int
main()
{
    int i;
    char ch;
    int counts[ALPHABET_SIZE];

    // initialize array
    for (i = 0; i < ALPHABET_SIZE; i++)
    {
        counts[i] = 0;
    }
    // count characters
    while ((ch = getc(stdin)) != SENTINEL)
    {
        ch = toupper(ch);     // convert characters to uppercase
        counts[ch - ’A’]++;   // increment count for current character
    }

    // print results
    for (i = 0; i < ALPHABET_SIZE; i++)
    {
        printf("%c: %d\n", i + ’A’, counts[i]);
    }
}
```

There are a couple of functions here that probably don’t look familiar. `getc(stdin)` is simply the C library function for reading a character at a time from input, and `toupper()` is a convenient function to transform characters to uppercase. `ctype.h` has many such functions that you may find useful at some point.

Could you do a (horizontal) bar chart when printing out the results? Could you write `toupper()` yourself? How would you extend this to handle lowercase letters?

Finally, think about functional decomposition, since we wrote all our code in `main()`. Which parts of this code could be put into a separate function?

4 Reading from the Command Line

For the last two weeks, we’ve had you write the following to begin each of your programs:

```
int
main(int argc, char *argv[])
```

Now that you’ve learned about arrays, you might have some idea what the second parameter, `argv[]`, is all about. It’s an array. But an array of what? In CS50, we’ve tried to make your lives a little easier by creating our own primitive type, `string`. In C, however, there is no such thing. Instead, if one wishes to deal with multiple characters in a row, one works with the type `char*`. For now, we’ll wave our hands and tell you that they’re the same thing, but in a few weeks when we learn about pointers, we’ll explain how C represents string data.

The parameters `argc` and `argv` provide a representation of the programs command line. The first, `argc` is the number of strings that make up the command line (including the command itself), and `argv` is an array that contains those strings. In Problem Set 1, you wrote the program `mario.c` and when it came time to run it, you ran the command following:

```
myname@nice (˜/cs50/pset1): mario
```

In that case, only one string appeared on the command line. If you had written your program to print the values of `argc` and `argv`, you would have seen only one string, `mario`. But if we had modified the problem specification so that the program reads the pyramid height from the command line, here’s how it might have looked:

```
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf("Sorry...not enough arguments!");
        return 1;              // what does this do?
    }
    int height = atoi(argv[1]); // see below

    // et cetera
       . . .
}
```

The function `atoi()` converts ASCII strings to integers. That is, the string `"329"` becomes the integer value 329. So instead of relying on `GetInt()` to provide user input, we take the value supplied by the user on the command line. For example, if the user wanted a pyramid with a height of 10:

```
myname@nice (˜/cs50/pset1): mario 10
```

What are the values of `argc` and `argv` passed into `main()` if we make this call? Consider using command-line arguments to accept input from the user. (And definitely use them where instructed in Problem Set 2!) However, always keep style in mind. Don't make it the case that you use the command line for so many things that the input line gets messy. Having clean code counts, and of course, you always want to make operation of your program as simple as possible for the user.

5 Advanced Constructs

5.1 The Conditional Operator

The ternary (three-argument) conditional operator `?:` (pronounced "question-mark-colon") is another kind of conditional. Whereas, the if-else statement determines which of two groups of *statements* should be executed, which presumably have different *effects*, the conditional operator determines which of two *expressions* is evaluated. The result of that expression becomes the value of the whole conditional expression. The syntax is as follows:

⟨*condition*⟩ `?` ⟨*true_expr*⟩ `:` ⟨*false_expr*⟩

If the condition evaluates to true, the value of the whole expression is the result of ⟨*true_expr*⟩, and in that case, ⟨*false_expr*⟩ is not evaluated at all. If the condition evaluates to false, it's the other way around. Expressions ⟨*true_expr*⟩ and ⟨*false_expr*⟩ must produce values of the same data type.

Use of `?:` is typically restricted to cases in which the ⟨*true_expr*⟩ and ⟨*false_expr*⟩ are short expressions. Long expressions tend to be cumbersome when used inside conditional expressions. A typical use of the conditional operator is in computing the maximum of two numbers. (See function `max` in Section 1.4.)

5.2 Type Casting

C provides an operator to force a conversion from one type to another, called a *type cast* (or simply a *cast*). To perform a cast, you place the name of the desired type, surrounded by parentheses, before the expression to be converted. For example:

```
    int i, j;
    double a, e, f;
    ...

    a = 100 / (double) i;
    j = (int) (e + f);
```

In the first expression, the integer `i` is cast to type `double` before it is used in the division. Note that this forces C to do this division using floating-point arithmetic, rather than integer arithmetic. In the second statement, the sum `(e + f)` is converted to type `int` before it is assigned to the integer `j`.

Casting cannot be used to convert strings to integers or doubles in the way you might expect. The following will compile, but will not do anything useful:

```
int
main()
{
    int foo;
    string bar;

    foo = (int) "123";
    bar = (string) 456;
}
```

In particular, the integer `foo` will not have the value 123, and the string `bar` will not have the value `"456"` after these assignments are executed.

To get conversions of the intended sort, other methods are necessary, You are encouraged and invited to check out the “Resources” page on the CS50 web site to discover ways this can be done, should you be interested.

6 More on Style

Here are a few more things to remember about style conventions.

- Use of white space. White space (blank lines, spaces, tabs, etc) can be used to visually group related parts of a program. For example, all the statements in a block are usually indented to the same level.
- Consistency. Two pieces of code that have similar structure should look similar. For example, you should format all your `if-else` statements one way (unless there is a good reason not to, which you might consider commenting).
- Commenting functions. The use of functions brings along new commenting requirements. When you write a function, you should write a “header comment” for it explaining what arguments it needs, what it does, and what it returns. Having this documentation present in your code will help you remember how to use your own functions, and will greatly help others (such as your TF) read your programs.
- Comments on comments. Comments should be visually distinct from the code, so the two do not get tangled up. Since the compiler does not care where you put extra blank lines, spaces, or comments, you have a considerable amount of freedom about how your program looks. If you use this freedom wisely, you will have programs that are much more readable, understandable, maintainable, and gradeable than if you do not.

7 Keeping Secrets

7.1 Cryptography and You

Encryption is pervasive in our world. Each time you use an ATM, or buy something on the internet, or log into `nice.fas.harvard.edu`, some sort of encryption is used to keep information about you secure.

The act and study of encrypting or enciphering information (putting it into code) is known as *cryptography*, and the people who attempt to “undo” and study cryptography are known as *cryptanalysts*. In Problem Set 2, if you choose to do the standard edition, you are primarily charged with the former task, to write some ciphers for encrypting information. The hacker edition also puts you in the position of being a cryptanalyst, as you are asked to decipher some sensitive information (passwords) as well.

7.2 Caesar Ciphers

The simplest type of encryption method we will be discussing in this course is the Caesar Cipher. To implement a Caesar Cipher, one simply chooses a “key”, some integer value from 0 to $k - 1$, where k is the length of the alphabet under consideration. For our purposes, we will be using the English alphabet, so $k = 26$. Then, one takes each letter they wish to encode in the message and applies the following formula to it:

$$c_i = (p_i + k)\ mod\ 26$$

Here p_i is the ith character of the original (unenciphered) message, k is the “shift” the Caesar Cipher employs, and c_i is the encoded ith character of the enciphered message. The “mod 26” is included so that the alphabet can wrap around. Applying a shift of $k = 4$ to the letter Y for example would be impossible without the wrap-around. With it, it easily enciphers to a C.

7.3 The Vigenère Cipher

The Vigenère Cipher, at first glance, looks significantly more complex than the Caesar Cipher. In reality, it is simply a glorified version of the same. The formula to encipher, in fact, is nearly identical, save for one small (though important!) change:

$$c_i = (p_i + k_i)\ mod\ 26$$

Here, c_i and p_i have the same meaning as they did before, in the Caesar Cipher. What’s different this time is the value k_i. Whereas in the Caesar Cipher, one chooses a single key value k, in the Vigenère Cipher, one instead chooses a “keyword”, and each letter of the keyword represents a different shift: `A` shifts by 0, `1` shifts by 1, and so on. If, for instance, we choose the keyword `COMPUTER`, and our message to encipher is `THIS CLASS ROCKS!`, we would encipher the letter T by shifting it using the key C (a shift of 2) to get the letter V. Then, when we wanted to encode the second letter of the plaintext, we would use the shift corresponding to the second letter of the keyword, and so on. The following demonstrates how the whole text shifts:

T	H	I	S		C	L	A	S	S		R	O	C	K	S	!
C	*O*	*M*	*P*		*U*	*T*	*E*	*R*	*C*		*O*	*M*	*P*	*U*	*T*	
V	**V**	**U**	**H**		**W**	**E**	**E**	**J**	**U**		**F**	**A**	**R**	**E**	**L**	**!**

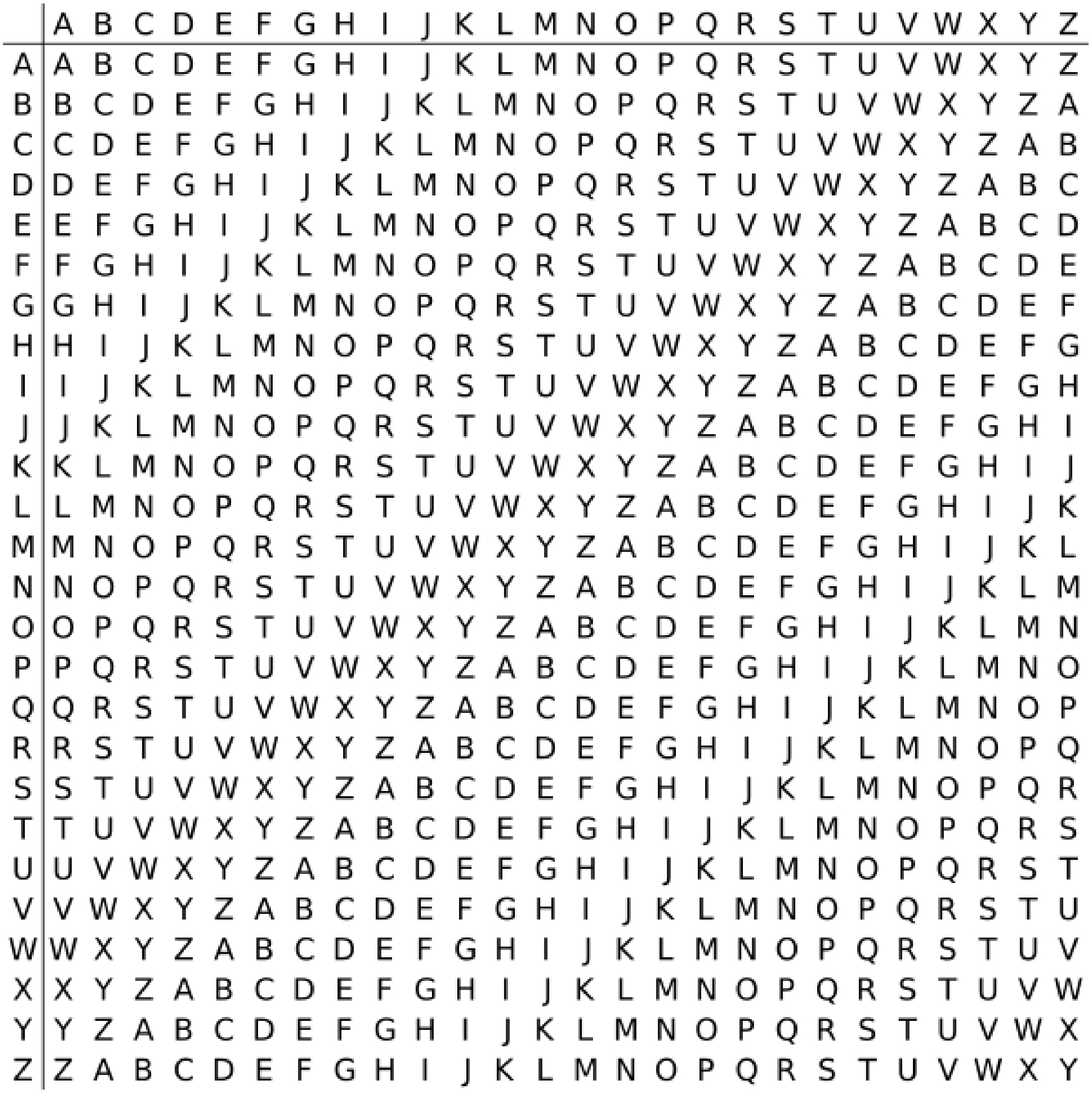
Notice how we do not encipher white space or punctuation marks. You can check the message using the *Vigenère Tableau*, also known as the *tabula recta*, a tool used by cryptanalysts, which we have attached as the final page of this week’s section notes. It is up to you, however, to figure out how you should use this table (if at all) and turn this long-used cryptography tool into something you can “encode” in your C program.

7.4 More Modern Methods

For some things, however, simple substitution ciphers don’t provide the right kind of protection. Imagine having your credit card information enciphered using the Caesar Cipher, or even the Vigenère Cipher. With enough brute force, a hacker will easily be able to obtain your information and steal your identity. Fortunately, cryptography as a science has advanced quite a bit since the time of Caesar and Vigenère, and many more means of encryption are now available to us.

David talked a bit in lecture about a few of these new methods: DES (data encryption standard), AES (advanced encryption standard), PGP (pretty good privacy), RSA (an initialism of its creators: Rivest, Shamir, and Adelman). These four, by no means, cover all of the possibilities for encryption available today. These topics are fairly advanced though. Delving into the algorithms would be far beyond the scope of this course. If you are interested, however, we recommend that you look up more information about these and other modern encryption methods. Additionally, for those who are working on the hacker edition of Problem Set 2, you’ll want to read all of the files and `man` pages suggested by the specifications quite carefully, so that you understand the algorithms in place and are more prepared to “hack” your way in.

8 The Vigenère Tableau



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

The left column specifies which “alphabet” is being used, and along the top row are the characters in plaintext. By matching enciphering alphabet to plaintext column, one can obtain the correctly ciphered character to place into the ciphertext.