C Programming Notes for Data File Lab

Richard Han
richard.han@mq.edu.au

Revised 26 July 2021

Introduction

The data file lab engages you with C programming including defining and using C struct, formatted printing, the command line, text and binary data files, memory allocation, sorting, pointers, and understanding binary data. These lab notes offer specific support for some relevant topics. Lectures will also cover relevant topics for the lab, and some relevant information can be found on the Internet and in the *C Programming Language* reference book.

Overview of C struct

A C struct is a data structure that contains items of different data types. For Java or C++ programmers, a C struct is like a class, except that it can only have data members (no methods) and all members are public.

In order to work with C structs, you need to *define* the contents of the structure, *declare* one of more struct variables (which are similar to instances of a class), and *manipulate* the members (or fields) of the struct variables. You can also *initialise* a struct variable, *copy* struct variables to others of the same type, and *pass* struct variables as parameters to functions. Finally, you can construct more complex data structures that include your defined struct: You can create an array of structs or you can include a struct as a member of another struct.

Although the C struct is much more limited than a class in Java or C++, the C struct actually closely reflects the way these other data structures are implemented by the compiler. The C language is designed to be a high level language that is also close to the machine, which is why it is used for implementing operating systems and one of the reasons for studying it in COMP2100.

Defining a struct

Here is a simple struct definition:

```
struct person {
    short int age;
    float weight;
    char gender;
};
```

The above piece of C code defines a structure of type `struct person` with three data members. The first member, `age`, is a short (16-bit signed) integer; the second, `weight`, is a single precision floating point number and the third, `gender`, is a character. This definition does not create any variables – all it does is to specify the fields of the structure `struct person`. Note that, in C, the data type is `struct person` not simply `person`.

The fields of a structure may be of any type known to C. This includes the simple types – integers of various sizes, floating point, characters and pointers. It also includes other structures and arrays – you can have a member of a structure that is an array.

The structure definition implicitly defines the size in bytes of each structure instance variable because it defines the information that needs to be stored in memory. The size of a structure may often be larger than the sum of the sizes of the individual components. We will discuss how structures are stored in memory in COMP2100.

Declaring a struct variable

The C type name of a structure is `struct` *sname*. You declare one or more variables of a structure type by naming the type and then naming the variable(s). For example:

```
struct person john, jill;
```

This C statement declares two variables – one call `john` and the other called `jill`. Both of these variables are of type `struct person`. Each of them has its own fields for `age`, `weight` and `gender`.

You can declare a struct variable in the same scopes as any other variable. Struct variables declared outside of any function will have file scope if they are qualified as `static`, or global scope otherwise. Struct variables declared inside a function will have scope restricted to the function. For example:

```
struct person yan;
static struct person donna;

void proc() {
    struct person petra;
    static struct person ali;
    …
    return;
}
```

In this example, `yan` has global scope and `donna` has file scope. `petra` is local to `proc` and there is a separate instance of `petra` each time that `proc` is called, while `ali` is local to `proc` but shared between all calls of `proc`[1] – because `ali` is static, it remembers its value from one call of `proc` to another.

Manipulating struct variables

The fields of a struct may be individually accessed using *variable.field* notation. You can use a field expression in any way that you would use a variable of the same type. In particular, you can assign and calculate with numeric fields as in the following example.

```
yan.age = 17;
yan.weight = 82.1;
yan.gender = 'm';
ali.age = yan.age – 2;
ali.weight = 79.2;
ali.gender = 'm';
if (ali.gender == yan.gender && ali.gender == 'm')
    printf ("They are brothers\n");
```

[1] For more information about C scoping rules, please refer to other information sources.

Initialising a struct variable

C allows you to statically initialise the contents of a struct variable at the same time as you declare the variable. For example:

```
struct person yan = { 17, 82.1, 'm' };
struct person ali = { 15, 79.2, 'm' };
```

The values in braces are stored in the fields of the structure in the same order as the fields appear in the struct definition. In this example, that order is `age`, `weight`, `gender` so `yan`'s age is 17, weight is 82.1 and gender is `'m'`.

This type of initialisation is particularly efficient for global variables, file scope variables and static local variables. In all these cases, the variable is only ever initialised once when the program is loaded. The variable is placed in memory at a location chosen by the compiler, and the compiler can preload the initialisation values into the variable. This means that no program code is required to initialise the variable. In contrast, initialising a structure by assigning values to each field individually (as explained in the section *Manipulating struct variables*) requires program code that copies the values into the structure during program execution. Procedure local variables are not placed in memory by the compiler. Instead, they are dynamically created on the stack so they must be initialised using program code at run time to set the fields to their values.

Copying struct variables

You can copy the values from one struct variable to another using the assignment operator. The two variables must have exactly the same type. For example:

```
petra = donna;
```

The C compiler generates program code that copies all the fields of the source struct into the destination struct. For large structures, this could take many machine instructions and would be much slower than copying a simple integer variable.

Passing and returning struct variables

You can pass a struct variable as a parameter to a procedure or function. Because C passes parameters by value, this means that the program has to make a temporary copy of the structure for the called procedure or function to use. As with copying structs, this can be inefficient for large structs.

You can also write a function that returns a struct function value. This feature is not commonly used.

A struct containing another struct

Here is a structure definition that contains two `struct person` fields and some other information.

```
struct parents {
    struct person mother, father;
    double income;
};
```

A struct containing an array

Here is a structure definition that contains an array of 3 floating point numbers.

```
struct labmarks {
    float mark[3];
};
```

An array of struct

Here is a declaration of an array of four `struct person` elements, and some code that manipulates the values of fields of individual elements of the array. Code that initialises the array has been omitted.

```
struct person family[4];
…
family[0].age += 1; // Had a birthday
family[2].weight -= 3.0; // Lost 3 kg
```

An introduction to formatted output with printf

Every programming language provides some means for the programmer to produce neatly formatted text output. This includes converting integers and floating-point numbers to text so that they can be displayed on the screen, printed or exported to other programs.

In the C programming language, formatting output is the responsibility of the function `printf` and its friends[2]. There are two main `printf` functions.

- `printf` outputs the formatted text to standard output. Normally, this is the display terminal screen, but Unix IO redirection can be used to send standard output to a disk file or to another command via a pipe.
- `fprintf` outputs the formatted text to a C FILE. You can open the file using `fopen`, write to it with `fprintf` and other functions, and close it with `fclose`. For more information, see the Unix manual pages.

The `printf` function is unusual because it can have different numbers and types of parameters depending on what you are asking it to do. The first parameter is always a format string. The `printf` function scans through the format string looking for formatting commands which begin with the `%` symbol. Everything that is not part of a formatting command is simply printed directly to standard output. This means that `printf` can be used with a simple formatting string to print ordinary text. For example:

```
printf ("Hello everybody\n");
printf ("I am feeling good today!\n");
```

However, if you just want to print arbitrary text strings, then you should use another function such as `puts()` which does not try to interpret formatting instructions within the string.

Whenever `printf` encounters a % symbol in the format string, it interprets the following part of the format string as a formatting instruction. The formatting instruction typically corresponds to an extra parameter value that is to be converted to text and inserted at that point in the output. Here are some format specifications:

[2] Scanning formatted text is provided by `scanf` and its friends.

Format specification	**Interpretation**
`%d`	Integer (int or smaller) value as decimal e.g. 17
`%f`	Floating-point (float or double) value as decimal e.g. 5.123456
`%s`	Insert character string (null-terminated array of char, or pointer to null-terminated array of char)
`%c`	Insert a single character (char)
`%%`	Literal % character in output

For example, consider the following C code:

```
# include <stdio.h>

int age = 17;
float weight = 75.2;
char name[] = "Annette";
printf ("%s is %d years old and weighs %f kg\n",
          name, age, weight);
```

The code formats the variable `name` with `%s`, the variable `age` with `%d` and `weight` with `%f`. It would print:

```
Annette is 17 years old and weighs 75.200000 kg
```

Note that `printf` requires you to include `<stdio.h>` at the top of your program.

Header files: `<stdio.h>` is a C library header file that defines the `printf` function and its friends, and also defines various constants and variables that are useful for doing I/O in C. You don't really need to know what is inside `<stdio.h>`. However, when you are using C library functions, you must include the system header files that are needed for each function. The Unix manual pages for each function will show you what header files are needed. Although the header files are reasonably standardised, there can be differences from one system to another. In particular, C programs for Windows often use different header files than corresponding C programs for Linux.

There are other format specifiers that are useful is more specialised situations. Here are some that you might need.

Format specification	**Interpretation**
`%u`	Unsigned 32-bit int or smaller, as decimal
`%x`	Int or unsigned int as hexadecimal
`%o`	Int or unsigned int as octal

You can also specify a length modifier that tells `printf` the size of the argument to be converted. The most important reason for using the length modifier is for formatting `long int` and `long unsigned` values. Sometimes, you need a length specifier for shorter values also. The following table shows some examples of using length modifiers. For more information see `man 3 printf`.

Format specification	Interpretation
`%ld`	Long int as decimal
`%lu`	Long unsigned int as decimal
`%lx`	Long Int or long unsigned int as hexadecimal
`%lo`	Long Int or long unsigned int as octal
`%hx`	Short int as hexadecimal
`%hho`	Char as octal

The `printf` format string also gives you the ability to specify formatting parameters to achieve the particular effect that you need. For all format specifiers, you can specify the field width – the minimum amount of space to be provided. If the conversion requires less than the specified width then `printf` will insert additional white space, which is useful for lining things up in neat columns. For floating-point numbers, you can use precision to specify the number of decimal places printed after the decimal point. For details of these features and more information about `printf`, see the Linux man page `man 3 printf`.

The main function and command line parameters

Every program has to know where to start execution. In C, programs start by executing the `main()` function. You write the `main()` function, of course, and from it you call all the other functions and procedures that are part of your program.

Here, again, is the `hello.c` program that was developed in *Compile, Run, Make C Programs on Linux*.

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

2. int main (int argc, char **argv) {
3.     printf ("Hello, %s!\n", argv[1]);
4.     return 0;
5. }
```

This program prints Hello to whatever name is given as the first command line argument. Let's analyse the program line by line.

1. The program uses `printf` which is one of a collection of functions provided in the standard C library. The *header* file (or *include* file) `stdio.h` defines prototypes for these functions, related data types (such as `FILE`), global variables (such as `stderr`) and defined constants (such as `EOF`). By including `<stdio.h>`, the C compiler becomes aware of all these things and permits us to use them appropriately in our program.
2. The function header for the `main` function.
    a. The `main` function is executed when the program starts, and the program terminates when the main function returns.
    b. The `main` function returns an integer value, so the type of the function is `int`. The return value is the exit status of the program. A zero value means that the program was successful. A non-zero value means that some error or unusual condition was detected – different values can be used to indicate different exit conditions. The meaning of the exit status is determined by each individual program. Unless otherwise specified for a particular purpose, programmers can use any small non-

zero values that they wish as exit status values to indicate various errors. The exit status must be a value that fits in a single byte.

c. The `main` function has parameters. The first parameter, `argc`, is the count of command line arguments provided to the program. Each command line argument is a text string, and the name of the program is the very first command line argument, so `argc` is always at least 1. If the user who started the program typed arguments after the command name[3], then `argc` will be 1 more than the number of arguments provided.

d. The second parameter, `argv`, is a pointer to an array of argument strings. In C, this means that it is a pointer to pointers to characters. To be more technically accurate, `argv` is an array of pointers, each of which points to the first character of one of the command line arguments. The type of `argv` may be given as `char *argv[]` or `char **argv` – both are exactly equivalent as we will learn in lectures when we discuss pointers. The command name and the arguments that the user provided are the values of the strings in `argv`. So, `argv[0]` is the name of the command, `argv[1]` is the first argument, `argv[2]` is the second argument and so on up to `argv[argc-1]` which is the last argument. The array `argv` has one more entry `argv[argc]` which is set to the NULL pointer – this makes it easy to scan the argument list in a loop, stopping at the null pointer.

3. The `printf` statement uses the format string `"Hello, %s!\n"` to print the value of `argv[1]`. Strictly speaking, this is not legitimate if `argc` is less than 2 – in that case, there is no command line argument `argv[1]`. In fact, if you compile this program with `gcc` on Linux and run it without any command line parameters, it will print “`Hello, (null)`” which is the GNU library’s way of saying that `argv[1]` was the null pointer. On other systems, the same program might crash.
4. The `return` statement returns the value 0. As discussed above, this is the exit status of the main program, and it indicates normal successful completion. If this program line is omitted, the program can still compile and run successfully, but the exit status will be whatever happens to be in a particular CPU register, and that value is unlikely to be zero so other programs will think that your program failed. It is very important to explicitly return the appropriate exit status.

An improvement to this program would be to check whether argv[1] actually exists, and do something different if it does not. There are many ways to achieve this, but here is one example.

```
# include <stdio.h>

int main (int argc, char argv[][]) {
    if (argc > 1)
        printf ("Hello, %s!\n", argv[1]);
    else
        printf ("Hello, world!\n");
    return 0;
}
```

[3] Typically, arguments are separated by white space in the shell. You can include space characters in the argument by enclosing it in quotation marks. For more details, see the manual page for the shell – e.g. `man bash`.

This version of the hello program reverts to the original behaviour of printing “Hello, world!” when the user does not specify a command line argument. You might like to consider how you could further extend the program so that it could handle two, three or an arbitrary number of arguments, printing something like “Hello, Joshua Bradley Smithson” if there were three arguments. To do that, you would want to know about loops in C.

Pointers

A pointer is a variable that tells you where to find another piece of information. Like a directional sign to the beach or a person pointing to a nearby bus stop, the pointer variable tells you where to find a piece of information of a particular type. To be precise, a pointer variable contains the memory address of another data item. Pointers can point to other variables, to arrays or array elements, to structures or structure elements, or to arbitrary data in memory. Pointers can also point to functions. In short, anything that is in the program’s memory can have a pointer to it.

Pointers are a very important concept in COMP2100. Please refer to lectures for a discussion of pointers.

Sorting and memory allocation

The Unix manual pages contain information about the `qsort` library routine, `malloc` and `free`. Please refer to the Unix `man` pages using the man command as follows:

```
$ man 3 qsort
$ man 3 malloc
```

malloc and free

`malloc` and `free` are C library functions that can be used to create and destroy objects in memory, specifically in the memory *heap*. `malloc` is C’s equivalent of `new` in Java. `free` is C’s way of disposing of memory objects that you no longer need. In Java, the system automatically detects objects that are no longer being used (this is called *garbage collection*) but in C you must call `free` for each object that you create with `malloc`. If you do not free some allocated memory, then your program will hold on to the allocated memory as long as it continues to run. This is called a *memory leak* – the memory is not serving any useful purpose but it cannot be used again until the program exits. Programs that are intended to run for long periods of time (such as operating systems and network servers) will crash unpredictably if they have memory leaks because they will consume increasing amounts of memory over time and eventually there will not be sufficient memory for them to continue running properly.

The Unix man page for `malloc`, shows that `malloc` requires the C library header file `<stdlib.h>`. It also shows that the prototype of the `malloc` function is as follows.

```
void *malloc(size_t size);
```

The `malloc` function accepts one parameter which specifies the size, in bytes, of the data object that you want to create. Commonly, the `sizeof` operator is used to compute the size of the object based on its type name. For example, we can allocate space to hold a single integer by passing `sizeof(int)` as the parameter to `malloc`. As another example, we can allocate space to hold an array of `n` characters by passing `n*sizeof(char)` as the parameter to `malloc`. Finally, we can allocate space for an instance of `struct person` by passing `sizeof(struct person)` as the parameter to `malloc`.

The parameter size is of type size_t which is a system-defined type that is large enough to represent the size of the largest possible memory object on the computer that you are working on. You don't need to worry about what it really is, but typically it would be unsigned int or unsigned long. Each system defines size_t appropriately so that programs are portable from one system to another. In order to write portable code, you should use variables of type size_t if you are computing with object sizes.

malloc returns a void * pointer. This is a generic pointer – it points to memory without specifying what type of information is stored there. Usually, we assign this pointer to a pointer variable of the correct type so that we can access the memory that has been allocated for us. For example:

```
int *intp = malloc (sizeof(int));
int n = 20;
char *stringp = malloc (n*sizeof(char));
```

In the above example, we allocate memory that can hold one integer, and point to it with the pointer intp. Then we allocate an array of 20 characters to hold a short text string, and point to it with the pointer stringp. The allocated memory is not initialised, so it may contain arbitrary data left over from previous work done in the current program.

There is a potential problem. Memory allocation can fail when the operating system refuses to allow our program to use any more memory. When that happens, malloc will return the NULL pointer. Before using the memory that we believe has been allocated, we should always check for memory allocation failure. For example:

```
if (intp == NULL || stringp == NULL) {
    fprintf (stderr, "Memory exhausted\n");
    exit (1); // Exit with failure
}
```

After allocating memory, we access it through the pointer in the normal way. For example:

```
*intp = 15; // Set the allocated integer to 15.
strncpy (stringp, "hello", 6); // Copy a string including nul
printf ("%s %d\n", stringp, *intp);
```

After using the memory, we free it.

```
free (intp);
free (stringp);
```

Sample program

The above code snippets have been combined into a working program which you can find at /home/unit/group/comp2100/alloc.c. Here is what happens when you compile and run that program.

```
$ gcc -o alloc alloc.c
$ ./alloc
hello 15
$
```

qsort

qsort is the C library sorting utility. It is described in the man page. Please read the man page first. The man page includes an illustrative example of sorting its command line arguments. That example is a little complex as your first example, so here is a simpler example.

This example program prints out an array of integers, sorts them and prints them out again. The array is statically initialised.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int cmpints(const void *p1, const void *p2)
{
    /* The actual arguments to this function are "pointers to
       int", but we need to access the integers, so first we
       cast the pointers to the actual type and then we
       dereference them to obtain the integer values */

    int *intp1 = (int *) p1;
    int *intp2 = (int *) p2;

    /* Compare and return comparison result
     * The result is >0 if the first int is greater than the
     * second.
     * The result is <0 if the first int is less than the
     * second.
     * The result is 0 if the two integers are equal.
     * We don't subtract the second int from the first because
     * there can be overflow that produces an incorrect return
     * value. */
    if (*intp1 > *intp2) return 1;
    if (*intp1 < *intp2) return -1;
    return 0;
}

# define ASIZE 10
static int a[ASIZE] = { 1, 3, 8, 7, 2, 4, 6, 5, 9, 0 };

int main(int argc, char *argv[])
{
    int j;
    // Print the array before sorting
    for (j = 0; j < ASIZE; j++)
        printf ("%d ", a[j]);
    printf ("\n");

    // Sort the array
    qsort(a, ASIZE, sizeof(int), cmpints);
```

```
        // Print the array after sorting
        for (j = 0; j < ASIZE; j++)
            printf ("%d ", a[j]);
        printf ("\n");

        // Successful termination of program
        return 0;
    }
```

Here is what happens when you compile and run this program (you can find a copy of the source code in /home/unit/group/comp2100/qsint.c).

```
$ gcc -o qsint qsint.c
$ ./qsint
1 3 8 7 2 4 6 5 9 0
0 1 2 3 4 5 6 7 8 9
$
```

Understanding the qsort call

The loops that print the array are quite standard C code, but the call to sort the array is what we are interested in. You should refer to the unix manual page for qsort and read it together with this tutorial explanation.

According to the qsort man page, the function prototype is:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

This prototype declares four parameters, as follows:

- base is a pointer to the start of the array of items to be sorted. In C, pointers are normally declared to point to items of some specific type, but when we want to write code that can work with pointers to any data type, we declare the pointer as void *. In the example, we wish to sort the array a, so we give the array name which C actually passes as a pointer to the array.
- nmemb is the number of items (members) in the array that is to be sorted. It is of type size_t which is an integer type that is suitable for representing the size of the largest possible array on the target computer. In the example, we wish to sort the ASIZE elements of the array a. Following good C programming practice, we have used a defined constant ASIZE for the size of the array a. The C compiler will automatically convert the constant ASIZE to the type size_t and pass it to qsort.
- size is the size of each element of the data array. Commonly, this is obtained using the C sizeof operator which returns the size of a data item, which may either be a type specification or a named variable of the desired type. In the example, we are sorting int elements, so we specify sizeof(int). By using sizeof we ensure that our code is portable to architectures where int may be a different size than on our current system. It is good practice in C coding to ensure that the code will work on other architectures where the primitive data types may have different sizes.
- compar is a pointer to a function. It has the most complex declaration, so let's split it into parts.

- `int (* compar)(…)` indicates that `compar` is a pointer to a function, and the function itself returns an integer. In C, we can use function pointers to make calls to a function using the function pointer instead of the actual function name. A function pointer is a type of variable that tells you where to find a function, so that you can call it. It is, in fact, implemented as the memory address of the program code that implements the function – in other words, it is a pointer to the machine code. In the example, the `qsort` library function needs to call a function that we have written (the `cmpints` function) to compare two integers together. Any time we use `qsort` to sort some particular type of data, we also need to provide a function that compared two instances of that data type and returns a value to indicate which one should be placed first in the sorted array. The `qsort` algorithm does not understand our data, but it knows how to sort data when we tell it how to compare data items.
- `(const void *, const void *)` indicates to the compiler that the `compar` function accepts two generic pointer parameters. The purpose of the `compar` function is to compare two data instances, so these pointers will point to the two instances that `qsort` wants our function to compare. In the example, we are sorting integers, so these are actually pointers to integers. It would be convenient if our function `cmpints` could be written with integer pointers (`int *`) parameters, but then the C compiler would find non-matching parameter types. So, we have to declare our function with generic pointers as required by the `qsort` prototype, and then within our function we use type casting to convert the pointers to integer pointers.

In the example, the comparison function that we want to use with `qsort` is our `cmpints` function, so we pass `cmpints` as the parameter `compar`. When we specify a function name as a constant in this way, the C compiler interprets it a pointer to the function, similar to the way that the name of an array is interpreted as a pointer to the array. The C compiler passes the pointer to `cmpints` into `qsort`, so `qsort` can call our function when it needs to. This is commonly called a *call back* – `cmpints` is the call back function that we pass to `qsort`.

Understanding the cmpints function

When `qsort` calls our comparison function, it passes generic `void *` pointers to each of the two items to be compared. `qsort` does not know (and does not need to know) what data is held in the items that we are sorting. Instead, it requires that our function returns a negative integer if the first item should be sorted *after* the second item, zero if the items are the same, and a positive integer if the first item should be sorted *before* the second item. (This is exactly what `strcmp` does, so you can sort an array of strings easily by passing `strcmp` as the comparison function to `qsort` – this is how the example in the `qsort` unix `man` page sorts the command line arguments.)

Our `cmpints` function first casts the pointers from `void *` to `int *` pointers, because we know that the pointers are actually pointing to integers. To make the code clear for novice programmers, we have used local variables to hold the type cast pointers. Then we explicitly dereference each pointer when performing the comparisons. There are a few different ways to write the comparison – we have chosen a simple but verbose solution. There are other solutions that are faster in C.

Len Hamey 2020