

The Public Domain licencing notice is removed on inspection copies.

# ShockSoc C Programming Support Sessions

Oliver Dixon  
ShockSoc Technical Officer  
`od641@york.ac.uk`

P/T/401 2pm–5pm  
Spring Term, 2021–2022

## Lab Script 03: Functions and Headers

INTERNAL SHOCKSOC INSPECTION COPY

*Please do not disseminate. Direct any comments to Oliver Dixon.*

## Table of Contents

<b>1</b>	<b>Functions</b>	<b>2</b>
1.1	Return Values & Arguments . . . . .	3
1.1.1	Exercises: Return Values & Arguments . . . . .	5
1.1.2	Bonus: Variadic Arguments . . . . .	5
1.2	Function Qualifiers . . . . .	7
1.3	Commenting Function Declarations . . . . .	7
1.4	The Calling and Placement of Functions . . . . .	9
1.4.1	Exercises: The Calling and Placement of Functions . . . .	11
1.5	Pointers to Functions . . . . .	11
1.5.1	Exercises: Pointers to Functions . . . . .	13
<b>2</b>	<b>Header Files and Multiple Translation Units</b>	<b>13</b>
2.1	... without Header Files . . . . .	14
2.2	... with Header Files . . . . .	14
2.2.1	Bonus: <code>#include</code> Guards . . . . .	16
2.3	Exercises: Header Files & Multiple Translation Units . . . . .	16
<b>3</b>	<b>Next Week...</b>	<b>17</b>

## 1 Functions

In this lab session, we will explore the concept of a fundamental C structure: the *function*. At least one function must be declared in every C program, as executable code must be contained within its scope. For most user-space code, this function is the “entry point”<sup>1</sup>, typically termed `main`, the notion of which was previously discussed in the first lab script. With that in mind, the vast majority of useful C programs contain many functions—often hundreds or thousands. As such, attaining a deep understanding of function declaration and usage is an integral endeavour for any prospective programmer.

Unlike many things in software development, there is a hard-and-fast rule for functions, which all programmers should religiously follow: a function must do one thing, and do it well. If any function in a program is exceeding thirty to forty lines of code, or contains statements deeper than three levels of indentation, then it is doing more than one thing, and should be split into auxiliary (or “helper”) functions. Code should also be wrapped on eighty columns, as longer lines tend to cause interoperability and readability issues across different displays and programmers. First, take some time to closely examine the function syntax in Figure 1.

Heavy use of small and modular functions brings many advantages, which become increasingly helpful as the size of a codebase grows. Software becomes

<sup>1</sup>All programs need not contain an entry point. Sometimes, such as in the case of a library, having an entry point does not make sense, as the program would never be run directly, but only by a client program to which it was linked. The client program would call certain functions explicitly as required.

```
qualifiers return-type name ( arg-type arg-name, ... )
```

Figure 1: The general function declaration syntax. Qualifier keywords and arguments are optional, however the function return type and symbol name are mandatory. For these syntax declaration figures, grey text indicates the expected presence of a keyword, as opposed to free-form symbol names. Optional fields are set in italics.

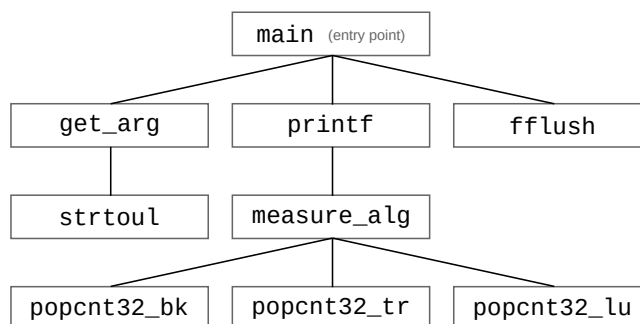


Figure 2: A partial call tree of a simple program to benchmark various Hamming Weight/**popcount** algorithms, with a simple binary alphabet of  $\Sigma = \{0, 1\}$ . A full copy of the program under inspection can be found at <https://www-users.york.ac.uk/~od641/misc-c/popcnt.c>.

*substantially* easier to manage, read, debug, distribute, and reuse. The calling card of an experienced programmer is often presented by their ability to take an algorithm and split it into distinct chunks capable of working together in a tree-like structure (see Figure 2), as functions begin to call other functions<sup>2</sup>.

## 1.1 Return Values & Arguments

The constant mention of modularisation begs the important question: how do functions actually work together? Like any useful algorithm, functions have inputs and outputs, and these are typically referred to as the arguments and return value, respectively. They may take any (reasonable) number of arguments, delimited with commas, and may return a single value. These may be of any type, as stated in the function declaration, with an arbitrary name by which the symbol shall be referred within the scope of the function. By default, C passes arguments by value, such that unless explicitly told otherwise—through the use of pointers—a function will take a copy of its arguments for its own use; any

<sup>2</sup>An extreme example of this is known as recursive programming, in which a function calls itself, iteratively, until the task completes and all functions return to the original instance of the caller. This is often a bad practice, and often leads to a processor error known as a stack overflow, such that the call stack, as shown in the previous lab script, cannot grow large enough to accommodate each function's frame data and return link address.

changes made to an argument in a function will not affect the original symbol as recognised by the caller. The same is true with return values.

These concepts can be tricky to grasp at first. Listing 1 shows an example function designed to perform a multiplication; the operands are passed as arguments `op1` and `op2`, and the result is returned. To ease into function declarations, recall the outstanding consistency and versatility of C syntax; all of the techniques in variable type declaration demonstrated in PSS script #2 are used in exactly the same form for defining function arguments and return values. In later lab scripts, we will begin to explore methods of passing compound types across function hierarchies, with pointers to `struct` types.

```
1 int multiply ( int op1, int op2 )
2 {
3     return op1 * op2;
4 }
```

Listing 1: A simple function that returns the product of its two arguments. As a bonus exercise, consider in what situations this function could be dangerous, invoking a strongly worded letter from the ALU. Think about the width of the `int` type.

In some cases, a function may have no reason to return a value, as it may modify data elsewhere; an exemplar of this is the `free` function, which never fails. To declare such a function, state the return type as `void`<sup>3</sup>. A `void` function will always return to its caller, as do all functions, however a value is not attached to the return address. In these instances, an explicit `return` keyword is not needed at the end of execution, as the compiler will automatically add a branch-to-the-caller instruction (“`bx lr`” on ARM) once the function goes out of scope. However, a `return;` statement may be used to exit a `void` function early; Listing 2 shows an example of this.

```
1 #include <stdio.h>
2
3 void print_if_odd ( int val )
4 {
5     if ( val & 1 == 0 )
6         return; /* The argument is even. Exit early. */
7
8     puts ( "The argument is odd!" );
9 }
```

Listing 2: A function to print “The argument is odd!” to `stdout` in appropriate situations. If the argument is even, this function produces no visible output. The `if` conditional syntax will be discussed in the next lab script.

---

<sup>3</sup>Note that `void` is **not** the same as `void *`. The latter, which is a void pointer, is a value equal to a memory address which points to data of an undetermined type.

```
1 int * dodgy_function ( )
2 {
3     int i = 0xFF;
4     return &i;
5 }
```

Listing 3: A mysteriously devious function, containing an unobvious mistake made by many inexperienced programmers.

### 1.1.1 Exercises: Return Values & Arguments

1. Write analogues to Listing 1 for the other three basic arithmetic operations. Given that addition and subtraction are closed under  $\mathbb{Z}$ , the implementation of these will be relatively painless. Aside from the potential of a divisor equal to zero, what sort of unexpected behaviour could be caused when implementing division? Perhaps a return type other than `int` may be helpful? What about the arguments' type?
2. While these sessions encourage forging a strong familiarity with defining function signatures, becoming comfortable with reading them is an equally important skill. At a Linux terminal, try typing `man`, followed by a space, followed by a standard library function. This will display the manual page for the given function through the system pager<sup>4</sup>, usually headed with the relevant signatures. Try `man malloc`, which describes the signatures for `malloc` and `free`, amongst various other related functions. What are the types of the return values and arguments?
3. Write a function that modifies the original value of its argument. Pointers, and the dereferencing thereof, may be helpful.
4. Write a function that, given a fixed-width integer from `stdint.h`, returns a copy of its argument with the  $n^{\text{th}}$  bit set, where `n` is also an argument. Recalling your answers from exercises in PSS #2 may allow you to complete this exercise quicker.
5. Consider the function shown in Listing 3. Why might this lead to a *race condition* bug, in which unexpected behaviour appears only some of the time?

### 1.1.2 Bonus: Variadic Arguments

The above sections discuss functions that take a fixed number of arguments, however the C standard does stipulate the existence of *variadic* arguments, in

<sup>4</sup>The system pager is typically defined as an environment variable, `$PAGER`, storing the command through which `man` pipes its output. If manual pages are not scrollable, try executing `export PAGER=less`, and retyping your `man` command. If this produces satisfactory results, append this line to your `~/.bashrc` file.

```
1 #include <stdarg.h>
2
3 int va_multiply ( unsigned int n, ... )
4 {
5     int product = 1;
6     va_list operands;
7
8     /* Populate 'operands' with the passed arguments, of
9      * which there are 'n'. */
10    va_start ( operands, n );
11
12    for ( unsigned int i = 0; i < n; i++ )
13        product *= va_arg ( operands, int );
14
15    /* Destroy the argument processor instance created by
16     * the 'va_start' function. */
17    va_end ( operands );
18
19    return product;
20 }
```

Listing 4: A function to multiply an arbitrary number of operands. The iterative `for` construct, along with a few others, will be examined in the next lab script.

which a function may take a fixed number of arguments (at least one), immediately followed by an arbitrary number of arguments, denoted with a literal ellipses (“...”). These notes are included for completeness, but variadic arguments should be generally avoided wherever possible, and are rarely used in real code, due to their awkwardness. `printf` is a perfect example of such a construct.

To use variadic arguments, include the standard `stdarg.h` header, which defines the `va_list` type, alongside various helper functions, the most important of which are `va_start`, `va_arg`, and `va_end`. Variadic argument processing must be explicitly initialised and destroyed, owing to the first and last functions shown above. `va_arg` is used to retrieve an argument of the next index in the `va_list` vector<sup>5</sup>.

Listing 4 shows an analogue of Listing 1 for an arbitrary number of operands. For those interested in the many peculiarities of the variadic argument functions, try running `man stdarg`. Advanced students already familiar with C may wish to delve into the implementation of these functions in one of the common standard library implementations, such as *glibc* or *musl* to determine the strange methods that enable `va_arg` to return a general non-pointer type.

---

<sup>5</sup>As some may have suspected, the entire variadic argument interface is commonly implemented as a state machine working with globals, hence the requirements to explicitly initialise and destroy an argument vector, coupled with the lack of support to use `va_arg` to select an argument of an arbitrary index. In general, unless absolutely required with no sensible alternative, this interface should never be used in robust code.

## 1.2 Function Qualifiers

Figure 1 mentions qualifiers as the first potential modifier to a function declaration. There are only a couple of such (important) qualifiers in C applicable to functions; these are `inline` and `static`<sup>6</sup>. By declaring a function as `static`, the programmer is stipulating that the following function may only be called from within the same translation unit, and may not be exposed, typically via a header, to other source files; attempting to do will cause the compiler to throw an error. The concepts of *headers* and sharing declarations across files will become clearer after reading the second half of this script.

The `inline` directive suggests that a program is likely to perform in less time, if each function call is replaced with the function itself. This idea may seem confusing at first. Recall that every function call corresponds to, at the very minimum, a few instructions on the CPU. In performance critical applications, or situations in which a function is being called millions of times in a loop, it often proves advantageous to completely remove the overhead of a function call and stack realignment. A high-quality modern compiler, such as `clang`, will automatically inline functions where a clear performance advantage exists; the `inline` keyword, which may not always be obeyed, is a mere suggestion to the compiler from the knowledgeable and context-aware programmer. (This can only be excepted with compiler-specific extensions and attributes, which are used in some later examples. In general, such constructs should be avoided in portable code.)

To demonstrate the potential performance gains enabled with `inline`, consider the program shown in Listing 5. The `popcnt` function, which is not inlined<sup>7</sup>, calculates and returns the number of set bits in its unsigned 32-bit argument, with the aid of a sixteen-value lookup table. On an Intel<sup>®</sup> Core<sup>™</sup> 2 P8400 CPU @ 2.26GHz, this program typically executes in  $36.086 \pm 0.5$  seconds. Now consider the replacement `popcnt` function signature in Listing 6. Supposing this function is executed by the same caller, under the same conditions, the program completes in  $27.851 \pm 0.5$  seconds, which clearly demonstrates the potentially significant overhead of a basic function call with minimal arguments.

## 1.3 Commenting Function Declarations

As with all constructs in C, effective and consistent commenting is a requirement. Unlike with certain languages enjoying de-facto documentation-generation systems, such as Java with JavaDoc, there is no well-defined rule for commenting C functions. However, competent programmers should ensure that *every* function declaration or implementation is preceded with a comment block ex-

---

<sup>6</sup>The `extern` keyword can also be applied to functions, however every compiler adds it as an implicit keyword, and can only be overrode by the complementing `static` qualifier.

<sup>7</sup>The ugly “`__attribute__ ( ( noinline ) )`” syntax prevents the compiler from ever inlining the function. It should not be generally used, aside from in cases such as this one, in which we *want* to create a poorly performing binary. The `always_inline` attribute does the exact opposite, regardless of the specified optimisation level. All examples here were compiled with `gcc -O3`.

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 /* This function will never be inlined. */
5 static __attribute__((noinline))
6     uint8_t popcnt ( uint32_t v )
7 {
8     static const unsigned char tbl [ ] =
9         { 0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4 };
10
11     asm ( "" );
12
13     return  tbl [ v          & 0x0F ] +
14            tbl [ v >> 0x04 & 0x0F ] +
15            tbl [ v >> 0x08 & 0x0F ] +
16            tbl [ v >> 0x0C & 0x0F ] +
17            tbl [ v >> 0x10 & 0x0F ] +
18            tbl [ v >> 0x14 & 0x0F ] +
19            tbl [ v >> 0x18 & 0x0F ] +
20            tbl [ v >> 0x1C          ];
21 }
22
23 int main ( )
24 {
25
26     register uint8_t c;
27     uint64_t i = 0;
28
29     for ( ; i <= 0xFFFFFFFF; i++ )
30         c = popcnt ( i );
31
32     printf ( "0x%X has %d set bits.\n",
33             ( uint32_t ) ( i - 1 ), c );
34
35     return 0;
36 }

```

Listing 5: A simple LUT Hamming Weight implementation for 32-bit values. This function could be substantially improved with a larger lookup table; a 256-valued table would halve the required number of indexes, as eight bits, as opposed to four, could be examined at once. The empty `asm` call is used to prevent the function from being “optimised” away by some unintelligent compilers; readers needn’t worry about its strange presence.

```

1 /* This function will always be inlined. */
2 static inline __attribute__((always_inline))
3     uint8_t popcnt ( uint32_t v )
4 {
5     /* ... */
6 }

```

Listing 6: A replacement function signature, suggesting—and using a compiler extension attribute to force—the inlining of the function body.



```
1 /* multiply: calculate and return the product of its integer
2  * operands 'op1' and 'op2'. This function does not check
3  * for cases of overflow or underflow. */
4
5 int multiply ( int op1, int op2 )
6 {
7     return op1 * op2;
8 }
```

Listing 7: A simple multiplication function with an equally simple preceding synopsis.

plaining its purpose, return value, arguments, and any quirks which may affect the caller (such as a function that modifies its pointer arguments). Even simple functions, such as the earlier two-operand arithmetic operation examples, should have a short synopsis comment. Eager readers may wish to take every function in this script and write a summarising comment to appear above its declaration; the PSS Author would be pleased to review any such attempts and offer advice for improvement. An example for the earlier `multiply` example is shown in Listing 7.

For some workflows, a well-defined and easy-to-parse commenting schema may be used, which can then be scanned by an external program to generate documentation in a variety of formats such as HTML or PDF. There is no such standard for C, but common software includes *Doxygen*, *Sphinx*, and *Natural Docs*. In such cases, defined tokens embedded into comments are used to enumerate various properties and characteristics of a function, such as the arguments, the return value, potential errors, and any additional notes.

## 1.4 The Calling and Placement of Functions

Functions only become useful once they are called! In fact, most compilers with any reasonable level of intelligence will remove any unused functions during the first stages of optimisation, once the AST generation and traversal (discussed in PSS #1) has been completed. Function calls have been used implicitly throughout this document, however a slightly explicit discussion will prove helpful.

To call a function from another function, type its full symbolic name followed by parentheses. Any arguments expected by the function, as defined by its signature, should be entered between the parentheses, delimiting them with commas. Unless the function call is part of a larger statement, it should be terminated with a semicolon. For non-void functions, the function call can be assigned to a value, such that the symbol on the LHS of the assignment operator takes the function's return value. Some standard library functions, such as those marked with the GCC `warn_unused_result` attribute extension, will generate a warning if the caller does not check or assign the return value; this is because errors are commonly indicated through a `return` statement<sup>8</sup>.

<sup>8</sup>An example of `warn_unused_result` proving useful is shown with the `malloc` function. It

On the traditional architectures, the process of calling a function consisted of various stages. The following is a simplified overview of this process:

1. The call stack is enlarged by an appropriate amount to accommodate the storage of all arguments. This is typically achieved by moving a value into the stack pointer register.
2. All arguments are pushed onto the stack in the newly defined region, typically termed as the *stack frame*<sup>9</sup>. This stack frame will also contain any local data for the function, and will grow (and potentially shrink) at runtime while new function-local variables are declared; this local data includes the address of the caller, allowing the `return` statement to be translated into an unconditional branch by the compiler. The address of the most recent caller will usually be stored in a register, often termed the link register<sup>10</sup>.
3. The caller will jump to the address of the function, as calculated at build-time by the linker (assuming static linking). The function will then pop the arguments from the call stack, ideally onto registers, and begin to execute its defined instructions, following normal control sequences.
4. Upon returning, the callee will branch back to its caller, which will then shrink the stack size to its pre-call state. **Data local to the callee could still exist and be accessible, but may be overwritten at any point.** (A potentially troublesome function designed to illustrate this point is shown in Listing 3.)

The placement of functions is also important, and their declaration across various translation units will be discussed further in the coming sections, however it is useful to know a few core concepts. A function must always be *declared* before the point at which it is first called, however its *implementation* may be anywhere after its declaration, including within a different file. To illustrate the strong manner in which these actions can be decoupled, see Listing 8. Notice, that although the function implementation must specify the symbolic names of each argument, the function declaration syntax makes this extraneous typing optional. As compilers traditionally read translation units top-to-bottom, local argument names are of no interest to the caller<sup>11</sup>, and thus may be omitted<sup>12</sup>.

---

is completely useless to call `malloc` without assigning its return address, since memory leaks will always occur if the process does not know the addresses of its own heap memory. On GCC, attempting to discard the return value of marked function generates a `-Wunused-result` warning; `clang` optimises the call away entirely, even with the notoriously aggressive `-O0 -g -Weverything` flags.

<sup>9</sup>The call stack was discussed and illustrated in PSS #2, complete with a diagram containing stack frames, thread data, and link address information.

<sup>10</sup>Compiler targeting ARM processors will often translate a `return` keyword to the “`bx lr`” native instruction, where `bx` is a branch opcode and `lr` represents the link register.

<sup>11</sup>Although local argument names are optional, many programmers choose to include them as a matter of style, purporting their uses in increasing readability without having to locate

```
1 #include <stdio.h>
2
3 static int multiply ( int, int );
4
5 int main ( )
6 {
7     const int test_op1 = 5, test_op2 = 3;
8
9     printf ( "%d * %d = %d\n", test_op1, test_op2,
10             multiply ( test_op1, test_op2 ) );
11
12     return 0;
13 }
14
15 static int multiply ( int op1, int op2 )
16 {
17     return op1 * op2;
18 }
```

Listing 8: An example of the `multiply` function having its declaration and implementation decoupled. As described, a function declaration must precede its first call, however the implementation may be anywhere else. This program could be rewritten to declare and implement `multiply` above `main`, in which case no decoupling would be required.

#### 1.4.1 Exercises: The Calling and Placement of Functions

1. Replicate the decoupling shown in Listing 8 for each of the arithmetic `add`, `multiply`, `subtract`, and `divide` functions. Under what circumstances may this level of separation be seen as appropriate? (This is subjective.)
2. Research other architectures and their stack models. The management of functions and threads is certainly not a ubiquitously defined process across all modern processors. The von-Neumann, Harvard, and the “hybrid” modified Harvard architectures have wildly differing approaches to SBMA and code-data separation principles.

### 1.5 Pointers to Functions

The C syntax—as shown in Figure 3—allows the declaration and usage of *function pointers*, which store the address of a function, and may be used to indirectly call a subroutine. The value of these pointers, and thus the functions to which they refer, may change over time, rendering them exceptionally useful for

---

the function implementation. Both styles of inclusion and omission are common.

<sup>12</sup>The only other noteworthy differing factor between function implementations and declarations, aside the termination of declarations with a semicolon, concerns functions taking no parameters. In the implementation case, specifying `void` between the parentheses is allowed, but not recommended. For declarations, in the interests of avoiding ambiguity, the `void` keyword is mandatory.

```
return-type ( * qualifiers name ) ( arg-type arg-name, ... )
    = &f-name;
```

Figure 3: The function pointer declaration syntax. As with all other variables, functions may or may not be initialised at the point of their declaration, but are certainly not guaranteed to be allocated in zeroed stack space. When referring to function addresses, the ampersand is optional (for reasons best known to the standards authorities).

```
1 #include <stdlib.h> /* Included for 'malloc'. */
2
3 /* A constant array of function pointers, named
4  * 'arithmetic', that contains functions returning a single
5  * integer and accepting a couple of integer arguments. In
6  * this case, it is initialised as a standard array with the
7  * optional ampersand. */
8
9 int ( * const arithmetic [ ] ) ( int, int )
10    = { &add, &multiply, &subtract, &divide };
11
12 /* A function pointer, named 'allocator', that refers to a
13  * function returning a void pointer and taking a single
14  * argument of type 'size_t'. This is initialised to the
15  * standard 'malloc' function, omitting the optional
16  * ampersand. This pointer is non-constant, and may be
17  * changed over time. */
18
19 void * ( * allocator ) ( size_t ) = malloc;
```

Listing 9: Various function pointer declaration and initialisation examples.

the iterative calling of similar functions. Function pointers are a secondary concept, however their importance cannot be understated; they often reduce code size significantly, with only a minute performance hit caused by the additional dereferencing operation. Modern compilers capable of strong static analysis may possess the ability to reduce a constant function pointer to a real call, obviating the dereferencing overhead entirely.

A function pointer declaration must contain compile-time information about the class of potential functions for which it will be a pointer; namely, the ordered list of argument types and the return value type. The exact same rules of standard function declaration apply to pointer declarations, with the notable addition of pointer qualifiers and a pointer name, both of which pertain to the function pointer, and not the referenced function. Example pointer qualifiers may include any keywords attachable to a normal datatype, such as `const`, `volatile`, or `static`. Arrays of function pointers are also supported, and would be declared in the usual manner: succeeding the pointer name with a series of brackets. A couple of examples are presented in Listing 9.

### 1.5.1 Exercises: Pointers to Functions

1. Try the function pointer syntax for yourself. Copy the declarations in Listing 9 and test them in a real program, using `allocator` to put aside some heap memory<sup>13</sup>, and the various indexes of `arithmetic` to perform some basic calculations. You will need to include local implementations for the `add`, `multiply`, `subtract`, and `divide` functions, with the latter subroutine performing integer-only division. Akin to the optional ampersand in function referencing, calling pointers can be achieved with or without an explicit dereference; “`( * fptr ) ( )`” and “`fptr ( )`” are semantically identical.
2. Declare and initialise a function pointer to your earlier `divide` function, taking a couple of `double` arguments and returning a single `double`. Mark this pointer to remain on the stack regardless of the active frame.
3. Declare a constant pointer to a hypothetical function that takes a function pointer of any class, and returns a `void` pointer. Clearly breaking declarations of this complexity over multiple lines, potentially with short comments accompanying particularly difficult instantiations, is a useful and common tactic. The importance of this will become especially obvious in later sessions, in which we study compound datatypes such as `structs`.
4. Think about multi-dimensional function pointer arrays. Do you dare to declare and initialise one? Although these are used *extremely* rarely in real production code, considering their syntax is useful for building a common familiarity with the versatile and oft-surprisingly consistent syntax of C.
5. Read the manual page for the `qsort` function in the Linux Programmer’s Manual. Do you find anything interesting about this function’s signature, particularly within its argument list? How might a similar construct be created for a function returning a function pointer?

## 2 Header Files and Multiple Translation Units

The concept of “modularisation” has been often mentioned as a desirable design strategy, but non-trivial programs may easily reach thousands of lines in a relatively small amount of time, and become intractable and difficult to manage. Modern IDEs and build chains ease the burden significantly, however the manual splitting and sharing of data and code between multiple translation units remains an invaluable skill for any modern high-level programmer. As mentioned in PSS #1, the primary purpose of the linker concerns symbol resolution, and any linker written in the past half-century will natively support an arbitrary number of C source files. When we have such powerful build tools, we should learn to exploit them to the fullest.

---

<sup>13</sup>Remember to `free` all `malloc`’d memory!

These symbol declarations may be “glued” together and shared across source files with special translation units called “header files”, the contents of which may be prepended to other files with the `#include` preprocessor directive<sup>14</sup>. These files always have the `.h` filename extension, just as source files always have the `.c` filename extension. We have already encountered these, many times, in the form of pre-defined standard headers, containing variable, function, type, and macro declarations for POSIX and *libc* functions; such examples include `stdio.h`, `stdlib.h`, and `string.h`.

## 2.1 ... without Header Files

First, we shall place the notion of a header file to one side, as they are not strictly required to interlink data and code between multiple translation units. Consider the programs illustrated in Listing 10, where two source files `printer.c` and `multiply.c` exist. The “printer” code calls a function in the “multiply” file, but since a valid function definition has been provided for the relevant function, the compiler assumes the symbol will be resolved during the linking stage, and can continue without error<sup>15</sup>.

Similar logic can also be applied to variable definitions, thanks to the `extern` keyword. Much like a function declaration lacking an immediately defined implementation, a marking a variable as `extern` promises the compiler that a variable of a given type and symbol name will be declared elsewhere, typically in another file, and should not be absolutely resolved until the linker stage. See Listing 11 for an example of `extern(al)` variable resolution.

## 2.2 ... with Header Files

In virtually every real-world situation, more than two source files will exist, and it is desirable to share (“export”) certain chosen symbols between them all. This can be achieved with header files, which is a type of translation unit typically containing only function, variable, preprocessor, and type declarations. In later lab sessions, we will examine some elaborate headers containing `struct`, `enum`, and `typedef` statements, but for these introductory purposes only function and variable declarations will be shown.

To include a header file, place an `#include` directive at the top of a file<sup>16</sup>, and with the relative path typed between double quotes; to include a header

---

<sup>14</sup>It is possible to `#include` any file on the filesystem, regardless of its extension. However, if you are telling the preprocessor to include anything other than an explicit header file, you have done something very wrong. **Redesign your code or throw it away.**

<sup>15</sup>In the unfortunate event of a signature existing without a corresponding implementation, the compiler will raise no complaint, however the linker will fail, reporting an “unresolved reference” error, or some variation thereof. Any functions used in this decentralised manner must also not be marked as `static`, as they can *only* be used by the translations units in which they are declared and implemented.

<sup>16</sup>Like all preprocessor statements, the `#include` directive can be placed at any point in the translation unit, however it is remarkably poor practice to include an external file after any non-preprocessor statements are issued.

```
1  /* file: printer.c */
2
3  #include <stdio.h>
4
5  int multiply ( int, int );
6
7  void print_multiply_output ( int op1, int op2 )
8  {
9      printf ( "%d * %d = %d\n", op1, op2,
10              multiply ( op1, op2 ) );
11  }
12
13  /* file: multiply.c */
14
15  int multiply ( int op1, int op2 )
16  {
17      return op1 * op2;
18  }
```

Listing 10: A function declaration and implementation can be decoupled across multiple files, providing that the declaration appears before any corresponding calls. To compile this code, one would specify all source files at the command line: “gcc printer.c multiply.c”. For this example, no entry point (such as main) is defined, so the linker will not be able to construct a valid binary.

```
1  /* file: read.c */
2
3  #include <stdio.h>
4  #include <stdint.h>
5
6  extern uint16_t var;
7
8  int main ( )
9  {
10     printf ( "var is equal to 0x%X\n", var );
11     return 0;
12 }
13
14 /* file: set.c */
15
16 #include <stdint.h>
17
18 uint32_t var = 0xDEADBEEF;
```

Listing 11: Similar concepts may be applied to variables, where the variable declaration, marked as `extern`, is separated from its actual instantiation. At the time of declaration, the compiler must know its prospective symbolic name and datatype; the linker will then resolve real memory offsets in the resultant binary. To compile and run this example, try executing “gcc read.c set.c -o vartest && ./vartest”.

```
1  /* file: printer.c */
2
3  /* The preprocessor will copy the content of 'decls.h' into
4   * 'printer.c', thereby exposing the declaration for the
5   * 'multiply' function, which is then implemented in
6   * 'multiply.c'. */
7  #include "decls.h"
8
9  int main ( )
10 {
11     printf ( "%d * %d = %d\n", op1, op2, multiply ( op1, op2 )
12 );
13     return 0;
14 }
15
16 /* file: decls.h */
17 int multiply ( int, int );
18
19 /* file: multiply.c */
20
21 int multiply ( int op1, int op2 )
22 {
23     return op1 * op2;
24 }
```

Listing 12: An example header file linking two source translation units.

./globals.h, the directive `#include "globals.h"` would be used. See Listing 12.

### 2.2.1 Bonus: #include Guards

Occasionally, when headers include other headers, which then include a tree of various other headers, identical code can be superfluously (and often erroneously) copied into the final program, increasing build times and substantially introducing the risk of linker complaints. To prevent this, a de-facto standard has been agreed upon, using the `#ifndef`, `#define`, and `#endif` preprocessor directives. With this method, each header exports a unique preprocessor symbol after its first inclusion, which can then be checked by all subsequent `#include` directives. If the symbol has not yet been defined, then define it, and include the header. Otherwise, do nothing, as the header has already been included. A minimal example is shown in Listing 13.

## 2.3 Exercises: Header Files & Multiple Translation Units

Relevant exercises will be provided by the PSS Author and published on-line shortly.



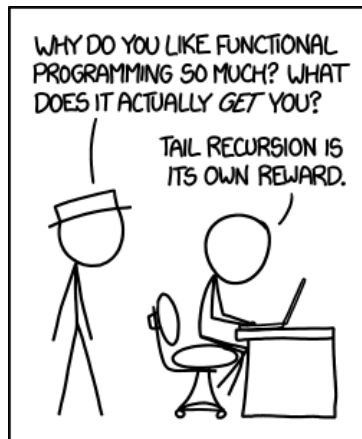
```
1 /* file: my_header.h */
2
3 #ifndef _MY_HEADER_H
4 #define _MY_HEADER_H
5
6 /* ... header content ... */
7
8 #endif /* _MY_HEADER_H */
```

Listing 13: A skeleton header file, `my_header.h`, surrounded by an `#include` guard. Exported preprocessor symbols used for guarding should always derive from the filename of the header file which they guard, with a single leading underscore to reduce the chances of ambiguity and naming clashes.

### 3 Next Week...

Next week, we will begin to explore various control sequences. These structures principally include conditionals (`if` and `else`) and iterations (`for`, `while`, and `do ... while`). We should also begin to examine compound datatypes, such as `structs`, and the ways in which they can be used to transport large and complex data structures around programs with very little effort.

See you then, I hope you enjoyed the third PSS script.  
PSS Author, Oliver Dixon



XKCD #1270: “Functional”. <https://xkcd.com/1270/>

PSS script dedicated to *M. Q.*