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 02: Variables and Data-Types

LaTeX document built February 1, 2022 on Linux.

Table of Contents

1 The Concept of a Variable

In the previous script, we discussed the concepts of *code* and *data*, treating them as a couple of distinct quantities that must be distinguished by the linker for correct construction of binaries. However, what is "data"? What is a "variable"? How do they interrelate with the previously discussed concepts of symbol tables and extended debugging information? Moreover, how do we define these symbols, and why do some languages stipulate restrictions on the form of data stored in certain types of variables? As a matter of fact, where are they stored, and how do we use variables to refer to other variables? What if we want to compartmentalise multiple pieces of data into one symbol, and how can attempting to access certain pieces of data cause our program to crash?

These are all extremely valid and important questions, the answers of which must be at the forefront of all low-level programmers' minds. Especially when working with C, which is fairly strongly typed[1] and also allows manual memory allocation on the heap, a strong foundational knowledge of variables and the allocations and uses thereof are absolutely essential. For this script, a basic knowledge of binary is assumed, including an elementary understanding of signed integer and floating-point arithmetic.

1.1 Variable Declaration and Manipulation

1.1.1 Variables & Datatypes: Declaration and Instantiation

The vast majority of any mildly technical cohort will have encountered, likely on a regular basis, the concept of a variable: the idea of assigning a symbolic

[1]The exact meaning of "strong typing" and "weak typing" will be discussed shortly, as will the definitions of the "stack" and the "heap". This passing remark was intended for advanced readers already familiar with typing.

name to a piece of data, the value of which may change over time. In C, and virtually all languages likely to be used by electronic engineers, all primitive variables are scalars[2]. It is often the case that a programmer may have a finite set of discrete values, in which case an n-dimensional array must be formed. Arrays are identified with a single symbol name, their members may be accessed individually through a process called "indexing".

As you may have expected, different languages have very different approaches to managing the use of variables: these can roughly be categorised into "strongly typed" and "weakly typed" languages, with C being a member of the former, and higher-level languages such as Python and JavaScript being members of the latter. In the case of strongly typed languages, a variable is declared, from the beginning, to hold only one specified type of data, and this may never be changed[3]; you can always be absolutely certain that the value contained within a variable is of a form acceptable to its prescribed type. With weakly typed languages, this is not the case: variables may be defined generically and assigned any value, the type of which is often guessed by the interpreter or virtual machine and changed on-the-fly. While the latter category may be perceived as easier by beginner programmers, seasoned developers will almost-ubiquitously support strong typing, as it removes the layer of abstraction and completely eliminates the prospect of working with unknowns. In Electronic Engineering and embedded programming, having full knowledge of the state of your program is critical[4], and strong typing is hence the preference of many C-like languages.

For declaring a variable in C, the premise is simple: the data-type followed by the symbol name, followed by an optional initial assignment. A set of qualifier keywords may also be used, to provide hints to the compiler regarding the intended use of the variable. Let's have a look at a few examples, and step through their meanings in sequence (Listing 1):

a) Declare `a` an integer with an initial state of `a = 2`.

b) Declare `b` as a character with the initial state of `b = 'A'`. Assuming the compiler conforms to the ASCII encoding (or some superset thereof), this will be stored as `b = 65`, as all characters must have an integer representation; the mapping of which is completely unknown to the compiler. The

[2]Throughout these lab scripts, we will begin to explore non-primitive data-types that may be defined by the programmer, such as `struct` or `union` types. These data structures are remarkably useful for the collation of variables of differing types. In object-orientated languages such as C++ or Java, `object` variables are instantiations of classes, which may contain data fields and methods (where methods are functions designed to operate on fields allowed by the scope and inheritance of their parent class). This can be emulated in C to some degree, using functions taking pointers to `struct` types, where the `struct` defines methods as members of a function pointer table. (In fact, early C++ compilers would generate C and delegate the actual compilation to a traditional C compiler.)

[3]It is always possible to *force* a piece of data to fit within a variable. This is achieved through a process called casting, in which a value is forcefully (and often crudely) converted into a state that conforms with the specified type of the assignee. For example, to cast a four-bit data field to an eight-bit data field would imply padding the four most-significant bits with zero, assuming little-endian.

[4]... lives may depend on it.

```c
int a = 2;               register char b = 'A';     float c = 0.01;
const int d = 3;         static short e = 8;        unsigned long f;
volatile double g;       long long h;               char * i = NULL;
```

Listing 1: Some increasingly complicated C variable declarations.

`register` keyword informs the compiler that it's probably a good idea to place the variable in a CPU register, however this is only a preference, and is not guaranteed[5].

c) Declare `c` as a floating-point number, with an initial state of `c = 0.01`. Floating-point arithmetic is a very complex area of Computer Science, and deserves an entire volume of lab scripts dedicated to nothing else. (In fact, the standard describing floating-point—known as IEEE 754—is an amusingly voluminous document [Com19].) Unless you're doing something deserving a great concern over decimal precision, you only need to know that the `float` and `double` types are used to represent rational numbers.

d) Declare a constant integer `d` of the state `d = 3`. If this value is changed at any point, the compiler should report an error (or **very** strong warning). Constants are useful for naming fixed values, such as the size of a memory page or a hard limit for an iterative algorithm[6]. Although a detailed analysis of compilers' treatment toward constants is beyond the scope of this document, it is worth knowing that only *in some cases* will a compiler perform a direct substitution. In the trivial case of "`const int d = 3`", every compiler would certainly do a straight substitution, as though it was a `#define`.

e) Declare `e = 8` as a `short` type, which is defined to be no larger than an `int`, but is allowed to be smaller; the size of two `short` values must also be greater or equal to the size of a single `char`. While this may be initially perceived as a useful optimisation, its use should generally be avoided in practice, unless the use of a smaller type is a hard requirement. These cautionary words are, again, due to performance concerns, which arise when dealing with data whose size is not a multiple of the size of a processor word[7]. Also note the `static` qualifier, which forces the persistence of the variable throughout the execution, even when the variable goes out-of-scope

[5]This is often used for variables that will be accessed regularly. As selection from main memory tends to be relatively slow, commonly accessed variables will be placed in CPU registers, which are (literally) inside the processing unit. However, modern compilers will typically detect these cases, and attempt to place variables in registers without the explicit `register` directive.

[6]Constants prove especially useful in embedded programming for defining unintuitive memory addresses defined by the hardware vendor, such as the famous `0x101f1000` UART0 TX memory address for VGA access in QEMU.

[7]CPUs are always fastest when dealing with their own words. A `short` may be smaller than an `int`, however in a typical 32-bit architecture, loading a 16-bit value requires one bus cycle to load thirty-two bits, and a further operation to discard the topmost sixteen. (Again,

and is not (easily) accessible to the rest of the program[8]. The concept of *scope* will be examined shortly.

f) Declare `f` as an `unsigned long` with no initial value, where a `long` is a value that may be larger than an `int`. Always be cautious with uninitialised values, as they may not always be equal to zero: they are equally likely to be filled with garbage, except in exceptional cases in which the memory arena has been zeroed prior to allocation (this is commonly used in cryptography and security applications to avoid private keys poncing around in memory). Also notice the `unsigned` qualifier, which instructs that arithmetic and bitwise operations not consider the existence of a sign bit, as is the case with one's and two's complement numbers, and to avoid performing sign-extension or shift-checking during numerical manipulation.

g) Declare `g` as a `double`, where a `double` is a `float` with potentially greater precision. The `volatile` keyword is very rarely used for user-space applications, however embedded engineers must be aware of its usages, since the programming of embedded devices typically mandates self-management of unpredictable memory arenas. When a variable is declared as `volatile`, the compiler knows that the value of the variable may be changed unexpectedly, likely by another process or other external force, of which the current process has no control or knowledge[9]. Volatile data is generally beyond the needs of most first-year undergraduates, however a general awareness of its presence is useful. (Infuriating bugs requiring disassembly await...)

h) Declare `h` as a `long long` type, with no initial value. We have met the tricky `long long` type before, albeit briefly in a footnote. Although it is occasionally useful to store very large integers, `long long` should generally be avoided, as it is a non-standard compiler extension defined by GNU as part of their GCC suite[10]. Unless you have complete assurance of the

this works under the assumption of a little-endian architecture. For a big-endian machine, the sixteen bottommost bits would be discarded.) With that said, `short` types are oft-useful for networking and file-storage protocols, in which packet/payload size (and hence transmission performance) usually prevails over processing performance.

[8]Like a few keywords in C, the `static` keyword has a double-meaning dependent upon the context. In the case of variables, `static` always denotes the persistence of the data, as described above.

[9]Despite the use of scheduler-aware interprocess communication mechanisms, data is still often changed by external execution. In some embedded cases, the sources for these changes includes memory-mapped hardware, the effects of which is impossible to predict in software. Multi-threaded applications may also fall afoul of this problem, as improper threading protection (such as mutexes and semaphores) may facilitate the corruption of relationships between thread-local and global variables. Processor interrupt routines, which will not be discussed at all, may also prove troublesome.

[10]If you need to do anything with very high precision or very large integers, arbitrary precision may be the best solution. There are a plethora of good libraries, such as GMP, enabling client programs to arbitrarily grow numerical types, at run-time, as the values contained within them require more bits. Those wishing to gain a deeper understanding of floating-point and signed arithmetic may wish to implement the basic features of one manually, although it is not an

toolchain under which your program will be built, non-standard extensions absent from the likes of ANSI or POSIX should be avoided, as they severely limit the portability of programs.

i) Declares `i` as a *pointer* to a `char`. Pointers tend to confuse novice programmers to no end, despite their concept being extremely simple and straight-forward: a variable containing a memory address, referring to another piece of data. While the data-type of the referenced data is expected to be `char`, the data-type of `char *` is a memory address, the size of which can be attained with "`sizeof(void *)`". Pointers to pointers ("double pointers") are also possible, as are triple, quad, or nth order pointers[11]. They will be discussed shortly, hopefully to the satisfaction of all readers.

1.1.2 Post-Declarative Manipulation & Shorthand

With the notable exception of the `const` type, all variables may be manipulated after their initial declaration and potential assignment. C follows (and generally established) many of the common rules for inputting expressions to programming languages; infix ordering is used, and the standard rules of BIDMAS precedence are respected. Parentheses are also supported. In additional to arithmetic operators, we also have bitwise operators, which work on the data at a bit-level. These are endlessly useful for otherwise-impossible optimisations. As a basic knowledge of binary is already assumed, descriptions will be sparse. All operators known to standards-compliant C compilers are listed in Table 1.

It is often useful to combine an operation on a variable with an assignment to the same variable, using its current value as the left operand. For all binary operators, C supports a shorthand notation for this purpose, such that an operator token immediately followed by an equals sign performs the described action. For example, considering some variables `x` and `y`, "`x <<= y`" is semantically identical to "`x = x << y`": both statements would shift `x` left by `y` places, assigning the new value back to `x`. Special-case shorthand notations are defined for the increment- and decrement-by-one operators: "`x++`" and "`y--`" are semantically identical to "`x += 1`" and "`y -= 1`", respectively[12].

1.1.3 Exercises: Variable Declaration and Manipulation

1. Declare an integer variable, `a`, with an initial value of zero. Using the most compact shorthand in three distinct statements, increment the integer by five, multiply it by two, and subtract seven.

easy task! (The ShockSoc Technical Officer has written a number of such libraries, and would be very pleased to discuss this topic with anyone interested.)

[11]If you're using pointers of any order greater than two, you're probably doing something wrong. In over ten years of C programming, the ShockSoc Technical Officer has used them twice in production code. In both instances, a double-pointer would have likely sufficed should the processes have been better-designed.

[12]These are particularly useful for defining iterative statements, which will be explored soon.

Operator	**Meaning**	**Example**	**Class**
`+`	Addition	`5 + 2 = 7`	Arithmetic
`-`	Subtraction	`5 - 2 = 3`	Arithmetic
`*`	Multiplication	`5 * 2 = 10`	Arithmetic
`/`	Integer Division	`5 / 2 = 2`	Arithmetic
`&`	Conjunction	`0xFF & 0x00 = 0x00`	Bitwise
`|`	Disjunction	`0xFF | 0x00 = 0xFF`	Bitwise
`^`	Exclusive Disj.	`0xFF ^ 0xFF = 0x00`	Bitwise
`~`	Inversion	`~(0xFF) = 0x00`	Bitwise
`<<`	Left Shift	`0 | ( 0xFF << 1 ) = 0xFF0`	Bitwise
`>>`	Right Shift	`0 | ( 0xFF >> 1 ) = 0x00F`	Bitwise

Table 1: A review of arithmetic and bitwise operators, including the unary (single-operand) inversion operator, for which no short-hand exists.

2. Declare an unsigned character variable, `b`. Using only the disjunction operator, set initial value to have only the first, third, fifth, and seventh bits set to one, such that its binary string representation is `0101 0101`. Working with pencil and paper may aid you here.

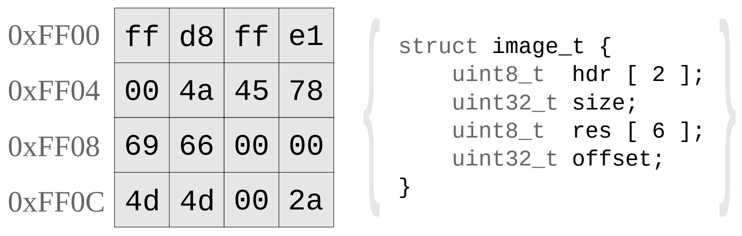
3. In a single statement, reassign `b` to be equal its is inverse, such that its binary string representation `1010 1010`.

4. Assuming `c` is an unsigned integer declared elsewhere, construct an assignment on `c` that sets the individual bit at position `n`, where `n` is an unsigned integer less than to the number of bits in the datatype of `c`. (You may also assume that `n` is defined elsewhere.)

5. Construct an assignment to clear the `n`th bit of `c`, again assuming `c` and `n` are defined elsewhere. Could you also construct a declaration of an integer, `d`, which stores either zero or one dependent upon the value of the `n`th bit in `c`?

These last couple of questions hint a very common optimisation in C, which essentially allows the storage of a large number of boolean values without using an individual variable for each one. For example, if you have a program that may take a maximum of eight toggle switches (in the form of command-line arguments), this method allows encoding all of their individually addressable values into a single eight-bit `char`, as opposed to eight thirty-two-bit `ints`[13].

[13]Toggling a bit at an arbitrary position is also occasionally useful, and can be achieved with an exclusive disjunction. Note that for wider data-types, the statement "`x ^= 1UL << n`" should be used, as the "`1UL`" syntax promotes the integer to a wider type, avoiding undefined behaviour (UB) when shifting on large `n`.



0xFF00	ff	d8	ff	e1
0xFF04	00	4a	45	78
0xFF08	69	66	00	00
0xFF0C	4d	4d	00	2a

```
struct image_t {
    uint8_t  hdr [ 2 ];
    uint32_t size;
    uint8_t  res [ 6 ];
    uint32_t offset;
}
```

Figure 1: Through the process of casting, an arbitrarily large chunk of data can be quickly "reinterpreted" in any fashion, such as a collection of fields in a compound `struct` datatype.

1.2 Additional Exercises: Casting

Earlier, a process called *casting* was mentioned. Casting is a method through which a variable of any type can be forced into a variable of a differing type. Casts should be rarely used in well-designed code, however on the rare occasions in which they do arise, they are exceptionally useful.

A classic example of casting goes as follows: you have a well-defined and well-packed structure in memory to represent the various fields of a file format, and you also have a large buffer filled with the bytes of a file corresponding to *exactly* the same format. By casting the data buffer to an instance of the `struct`, you have transformed the data into an accessible form for no performance penalty, since casting is a compile-time operation[14]. (Recall the important point that the CPU itself has no notion of a "type".) A simple example of such a case is illustrated in Figure 1.

1. In order to forge a familiarity with casting, its C syntax will not be described in this script. With programming, having information-seeking skills is equally important as remembering the syntax of a particular language. For this first exercise, your task is to research the C casting syntax.

2. Once you are comfortable with the basic concepts of casting, try performing a cast between an `int` and a `long`, and vise versa. Which one of these casting operations is more likely to cause an issue? In what circumstances would this not be dangerous (but still silly)?

3. A cast will *never* fail, as per the C standard. A compiler may issue warnings for dangerous operations involving signed and floating-point types, but the build system is bound to accept any order of ridiculous expressions. What types of undefined behaviour-inducing errors may occur when "forcing" data in this way?

To demonstrate the power of casting, some may enjoy reading around "The Fast Inverse Square Root", used for calculating an approximation of $1/\sqrt{x}$. Part of this algorithm is often hailed as "the greatest line of C ever written".

[14]Just be careful to avoid falling afoul of C's peculiar strict aliasing rules!

1.3 The Stack and the Heap

In order to fully understand the idea of a 'variable', you must also comprehend the idea of placement. Where *is* data stored, and how should the programmer determine which types of data should be stored where? For our purposes, there are two major abstract locations: the **stack** and the **heap**. The actual organisation of these areas is rather complicated, and often varies by processor architecture and implementation, however an idea of the abstract interfaces used to confer with these areas is immensely useful and commonly necessary.

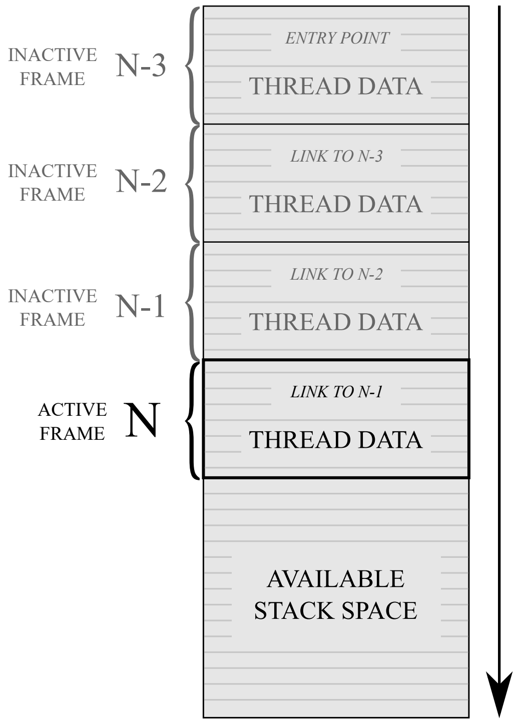


Figure 2: A standard downward-growing stack arrangement, with marked active and inactive frames.

First, we shall motivate the stack, which is the region used for standard variable declarations[15], such as those enumerated above.

In the field of Computer Science, a "stack" is a common word to describe an abstract datatype, such that elements are "pushed" (added) onto the top of the stack, and "popped" (taken) from the top; this model is commonly referred to as a *last-in-first-out* (LIFO) arrangement, for obvious reasons. In our context of SMBA (stack-based memory allocation), the stack is split into frames, typically with one frame-per-thread[16]. With the stack typically growing downwards from its origin (i.e., the active frame N is the child of the inactive frame $N-1$), it usually stands as the most time-efficient method of data storage due to its LIFO allocation structure. Frames are

[15]Although the stack is usually manipulated through direct variable instantiation and assignment, there is a little-known standard library function, `alloca`, which dynamically allocates space on the stack, which is automatically freed when the frame becomes redundant. For more information regarding this obscure and curious function, confer with `man alloca`. It is usually a bad idea to use this function, as it has poor error-reporting, and will not inform the caller if the stack cannot be extended. Also, the conditions under which the data is freed can often prove ambiguous, and should not have a place in robust user-space code.

[16]Do not worry if the meaning of a CPU 'thread' is not immediately obvious. For this hugely oversimplified context, these statements can be interpreted as one frame-per-function, with each frame N containing the return address, which corresponds to the caller responsible for frame $N-1$.

also automatically reclaimed once their data becomes redundant due to an explicit operation, such as a function returning to the address of its caller[17].

Conversely, we have the heap. Sometimes called the "free store", the heap is a far less elegant structure, often being implemented as a large pool of memory granted to the process, by the scheduler, to use as it wishes; this is heap-based memory allocation (HBMA, Figure 3). Whilst being significantly slower and drastically more complex on an interface-implementation-level, it does permit storing *very* large amounts of data in memory, limited only by the amount of the installed physical memory and addressing capabilities of the CPU; this often grows into the tens of gigabytes for modern personal computers, games consoles, and servers[18]. Unlike with SBMA, HBMA must be performed through an explicit standard interface, which is typically implemented through the `malloc` and `free` functions, to allocate memory of an arbitrary size and return it to the free store, respectively.

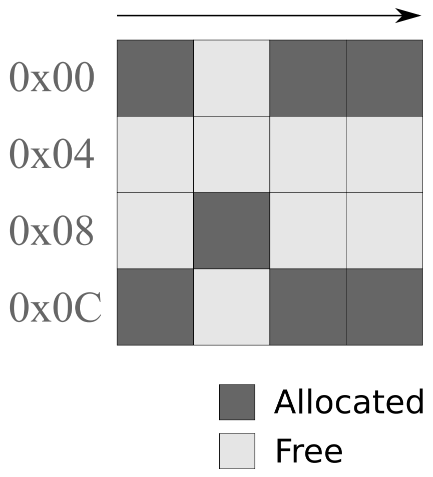


Figure 3: A free store pool region, with large amounts of external fragmentation.

In general, such dynamic HBMA should only be employed when static SBMA is not an option, as all allocations must be tracked and appropriately freed; doing this incorrectly often leads to *memory leaks*, which occur when processes allocate memory and do not hand it back to the OS before exiting, thus depriving other processes of using the extraneous blocks[19]. Other implementation-specific issues are rife, and often form very difficult problems in Computer Science [Knu97]. One such issue is that of arena fragmentation, in which repeated allocation

[17]Although remarkably useful, the automatic reclaiming of stack frames is the cause of a very common bug, in which a function returns a pointer to some local data, which the caller then attempts to dereference. This will often work without issue, as frames are not immediately cleared, however this is a prime candidate for a race condition as the integrity of the referenced data cannot be guaranteed.

[18]RAM capabilities of embedded systems with on-board memory vary greatly, however they will be vastly smaller than most non-embedded configurations.

[19]This is not strictly true: any decent operating system will automatically free ("clean up") any `malloc`'d data upon a process returning from its entry point, however it is terrible practice to write "leaky code", as it is affectionately termed. Dynamic allocation tools such as *valgrind* are invaluable for tracking such leaks; these tools run a client process under a virtual machine, which can explicitly track each `malloc` and `free` call and quickly detect leaks. Coupled with unit testing, this type of analysis can develop impenetrable programs. Modern languages, such as Rust, have been carefully developed to allow complete memory safety by using strong compiler hints to enable state-of-the-art static analysis. Furthermore, garbage collection can be implemented without a significant performance penalty due to clever manipulation of the fetch-execute CPU pipeline and branch predictor with out-of-order execution.

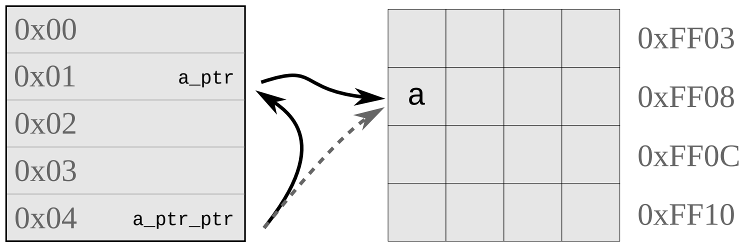


Figure 4: A stack pointer referencing another stack pointer, which ultimately references some data on the heap. This sort of arrangement becomes exceptionally useful for handling dynamic allocation of single data regions across functions.

and deallocation result in the creation of many small spaces contiguous blocks, greatly inhibiting the ability of the allocator to quickly find a set of consecutive blocks to accommodate the requested region size. The same issue occurs on permanent storage, and some environments (such as the Java Virtual Machine) may perform occasional run-time memory defragmentation. This is an extremely costly operation, however it may obviate future delays on HBMA calls.

2 Pointers

2.1 An Overview: Pointers are Seldom Difficult

Pointers have occasionally been mentioned throughout this document, however they have yet to be rigorously defined. The mere concept tends to confuse and discourage novice programmers, however such confusion tends to prove unwarranted. In their simplest form, a "pointer" is an integer containing a memory address to refer to another variable, which may or may not be a pointer itself. While pointers may exist on the stack, their referenced data typically exists on the heap, although it may exist anywhere, including memory mapped by peripherals. See Figure 4 for a graphical example.

In C, pointers are used extensively, to the point of it being reasonably impossible to compute anything awfully useful without employing pointers to some extent, even if their use is not explicitly known by the programmer. As previously discussed, they are declared with an asterisk, and also dereferenced with the same character. A preceding ampersand is used to retrieve the address of a variable, which is occasionally useful when assigning to a pointer type. This is illustrated in Listing 2. Functions, especially those concerning HBMA, also use pointers exclusively in their return type and arguments to refer to heap memory.

As C is strongly typed, its pointers must conform to all of the same rules. Just as with any other variable declaration, the datatype of **the data to which the pointer refers** is specified. Regardless of the size of the referenced data, the pointer itself will always be **an integer of a fixed size** (c.f. `sizeof(ptr_t)`), which is typically a multiple of the CPU word size; see Listing 3.

```c
#include <stdio.h>

/* Output: a = ff, ptr_a = 0x7ffdf3acf838, b = ff */

int main ( )
{
        int a = 0xFF;
        int * ptr_a = &a;
        int b = *ptr_a; /* b = a */

        printf ( "a = %x, ptr_a = %p, b = %x\n", a, ptr_a, b );
        return 0;
}
```

Listing 2: A simple C program to create an integer and manipulate its pointer. For now, the `printf` line can be considered as a (very useful) black box for pretty-printing and debugging.

```c
#include <stdio.h>

struct my_big_struct {
        long a, b, c, d, e, f;
};

/* Output: my_big_struct: 48; my_little_pointer: 8 */

int main ( )
{
        struct my_big_struct my_big_struct;
        struct my_big_struct * my_little_pointer = &my_big_struct;

        printf ( "my_big_struct: %ld; my_little_pointer: %ld\n",
                        sizeof ( my_big_struct ),
                        sizeof ( my_little_pointer ) );

        return 0;
}
```

Listing 3: Even in the case of a larger data type, the size of its pointer remains constant. Notice the size of the referenced type is forty-eight bytes, whereas the pointer size remains eight bytes.

Now we have motivated the basic idea of a first-order pointer, it is worth mentioning second-order pointers, such as the one illustrated in Figure 4. Pointers may point to other pointers, which tends to be extremely handy for cases where slightly more complex memory management and abstraction is required; such examples will appear very often in larger codebases. The syntax involved with second-order pointers is intuitive: one more asterisk in the declaration indicates an additional pointer order, and one more asterisk in the dereferencing statement means “dereference this to get a pointer, and then dereference that”. A simple example is shown in Listing 4. $n_{\in\mathbb{N}}$-th order pointers are allowed by the C syntax and compilers, however any situation in which $n > 2$ would be worryingly atypical.

```c
#include <stdio.h>

/* Output: A A A */

int main ( )
{
        char     a          = 'A';
        char   * a_ptr      = &a;
        char  ** a_ptr_ptr  = &a_ptr;

        printf ( "%c %c %c\n", a, *a_ptr, **a_ptr_ptr );
        return 0;
}
```

Listing 4: In this example, a second-order pointer is used to refer to the same character three times, through increasing levels of (de)referencing.

Once we start to examine functions in greater detail, pointers shall be covered again extensively in the context of function arguments and return types (where they are most useful), however you should now have a basic understanding of referencing in C.

2.2 Arrays

'Arrays' being a subsection of 'Pointers' may surprise some, however the C standard dictates an intricate relationship between the two, the importance of which cannot be understated. This seems initially unintuitive, but begins to make perfect sense upon realising that the stack and heap *are* arrays by their very nature. Thus, when declaring an array on the stack, the new variable is merely interpreted as a pointer to the array's first element, with the compiler ensuring the stack is adequately extended to accommodate the requested size. As such, the first element of an array can be accessed with the asterisk dereferencing operator, and further elements can be accessed through a process called *indexing*, in which the position of the desired item, counted from zero, is typed in brackets after the variable name. Listing 5 contains an exemplar of a simple array assignment and indexing, coupled with an example of a multi-dimensional array[20]. Although the latter array may seem initially daunting, its syntax is a simple and logical extension of the single-dimension case, with index specifiers for each "row" and "column"[21]. When arrays are passed to or returned from functions, they are

[20]Some may notice that in `str`, the size is not explicitly given. For standards-compliant compilers, the size of the first dimension may be omitted. Further dimensions must always be given explicit bounds.

[21]Two-dimensional arrays can be visualised as tables, and three-dimensional arrays can be similarly visualised as tables such that every cell has a "depth", made up of additional cells.

implicitly converted to a pointers, thus creating a situation of *pass-by-reference*, as opposed to the default *pass-by-value*. There are subtle differences between the placement of arrays and pointers, however they are beyond the scope of this document, and of primary concern to compiler developers.

2.3 Exercises: Pointers & Arrays

For these exercises, you may find it useful to investigate the purpose of the `printf` function, and the methods through which its various format specifiers can aid your understanding of variables and pointers. Once you have included `stdio.h` and placed some `printf` calls inside the `main` function, you can now print any (primitive) variables that you declare: the `%p` specifier is especially useful for printing memory addresses.

1. Declare some variables, and declare their pointers. Try using the `printf` function to print the value of the original variable, the contents of the pointer, and the dereferenced pointer.

2. Investigate the meaning of a *void pointer* as the C generic type. These cannot be dereferenced directly, so why are they useful? Can you dereference some of your previously defined variables through dereferencing some cast void pointers? What about double void pointers? How many casts would you need?

3. The `stdlib.h` header defines `NULL` to be "zero, cast to a void pointer": i.e., `#define NULL ( ( void * ) 0 )`. Try assigning this value of `NULL` to a local pointer, and then try to dereference it. What happens? Uninitialised memory does exist at `0x00`, so why might the operating system and scheduler prevent client programs from accessing memory outside of well-defined regions? Do you get a similar result when trying to index beyond the bounds of an array?

4. What are the intricate differences between an array and a pointer? Answering this question fully may mandate some independent research.

5. Experiment with the `malloc` and `free` functions by including `stdlib.h`. The POSIX/Linux programmers' manuals are fantastic for these purposes, and can be accessed by executing `man` *`function`*. In later sessions, we will begin to clearly illustrate memory leaks with the *valgrind* and *memchk* dynamic analysis tools.

6. **(Optional.)** Those eager to explore next week's content may be interested in the concept of a "function pointer". Consider how these arrangements may be useful when implementing a simple four-operation two-operand arithmetic calculator.

Higher dimensions, as in Euclidean geometry, become greatly harder to visualise, and are rarely used in real code. In fact, strings in C are implemented as character arrays, and all string functions in `string.h` operate on types of `const char *`.

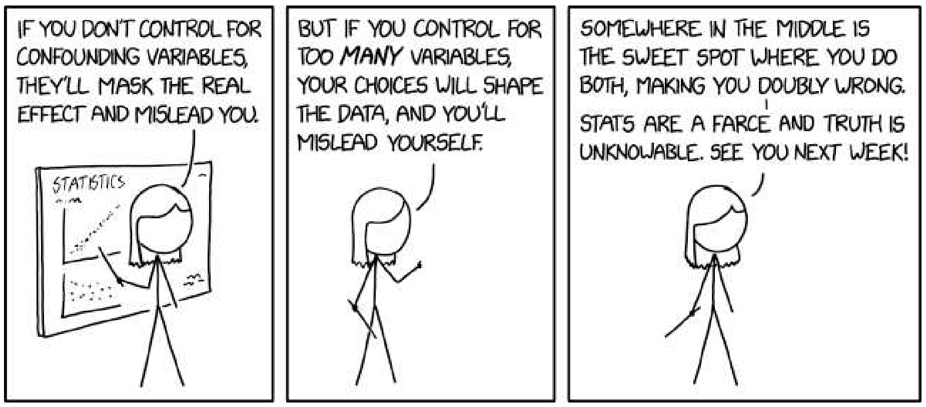
```c
#include <stdio.h>

/* Output:

   H e l l o
   0 1 2 3
   4 5 6 7
   8 9 10 11
*/

int main ( )
{
        /* The 'str' array may be alternatively defined as
         * char * str = "Hello", which would append a
         * NULL-terminator ('\0') as the final element.
         * String-handling functions in <string.h> distinct
         * themselves from memory-handling functions due to
         * their special treatment of NULL-terminators. */

        char str [ ] = { 'H', 'e', 'l', 'l', 'o' };

        printf ( "%c %c %c %c %c\n",
                        str [ 0 ],
                        str [ 1 ],
                        str [ 2 ],
                        str [ 3 ],
                        str [ 4 ] );

        int matrix [ 3 ] [ 4 ]  = {
                {  0,  1,  2,  3 },
                {  4,  5,  6,  7 },
                {  8,  9, 10, 11 }
        };

        printf ( "%d %d %d %d\n%d %d %d %d\n"\
                 "%d %d %d %d\n",
                        matrix [ 0 ] [ 0 ],
                        matrix [ 0 ] [ 1 ],
                        matrix [ 0 ] [ 2 ],
                        matrix [ 0 ] [ 3 ],
                        matrix [ 1 ] [ 0 ],
                        matrix [ 1 ] [ 1 ],
                        matrix [ 1 ] [ 2 ],
                        matrix [ 1 ] [ 3 ],
                        matrix [ 2 ] [ 0 ],
                        matrix [ 2 ] [ 1 ],
                        matrix [ 2 ] [ 2 ],
                        matrix [ 2 ] [ 3 ] );

        return 0;
}
```

Listing 5: A program showing the assignment and dereferencing/indexing of various arrays, the latter of which is multi-dimensional. As mentioned, the line dereferencing the first element of `str` may be rewritten as `*str`.

3 Next Week...

Next week, we shall finally examine functions, the concepts around scoping, and the vital skill of commenting. This is when things start to get interesting, as we can write modular code across various files and directory structures, allowing our code to become superbly structured and maintainable.

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


IF YOU DON'T CONTROL FOR CONFOUNDING VARIABLES, THEY'LL MASK THE REAL EFFECT AND MISLEAD YOU.
STATISTICS
BUT IF YOU CONTROL FOR TOO MANY VARIABLES, YOUR CHOICES WILL SHAPE THE DATA, AND YOU'LL MISLEAD YOURSELF.
SOMEWHERE IN THE MIDDLE IS THE SWEET SPOT WHERE YOU DO BOTH, MAKING YOU DOUBLY WRONG.
STATS ARE A FARCE AND TRUTH IS UNKNOWABLE. SEE YOU NEXT WEEK!

XKCD #2560: "Confounding Variables". `https://xkcd.com/2560/`

PSS script dedicated to *M. Q.*

Referenced Works

[Com19] IEEE Standards Committee. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std. 754-2019 (Revision of IEEE 754-2008)* (2019). DOI: `10.1109/IEEESTD.2019.8766229`.

[Knu97] Donald Knuth. "Fundamental Algorithms". In: 3rd ed. Vol. 1. The Art of Computer Programming. Addison-Wesley, 1997. Chap. Dynamic Storage Allocation, pp. 435–456.