Syntactic sugar and obfuscated code: Why learning C is challenging for novice programmers

Michael Wirth
School of Computer Science
University of Guelph

Sally sold C shells by the C shore.
But should she have?

C is an inherently complex language, not from the perspective of the language, because it is a small language in comparison to the likes of Ada or C++. However there is an inherent complexity to learning the language because of its assembly-like heritage. This white-paper overviews some of the more challenging aspects of C from the perspective of the novice programmer, and offers alternatives from other languages, where the usability may be improved.

June 2018

1. Introduction

The debate over the choice of programming language use in CS1 (the core introductory programming course) has been going on as long as languages have been used to teach programming. Fundamentally CS1 has three basic goals. The first involves disseminating the basic principles of programming, such as decision statements, repetition, modularity, testing, readability etc. The second involves teaching how the basic principles are implemented in a specific programming language. The third involves showing how programming can be used to solve problems. It is the latter which spurs interest in programming - taking a problem, and deriving a solution which can then be turned into a program. The reality is that people write code to solve problems, not just for the sake of writing code. To facilitate this breadth of knowledge, the language used in CS1 must help bind the three goals together - too much time spent on language syntax, or higher level notions such as object-oriented concepts detracts from this.

In the years following the evolution of first generation programming languages, the languages used for teaching paralleled those used in industry. Fortran was one of the fore-runners in CS1 courses, followed shortly afterwards by Algol. The 1970s heralded languages which were true teaching languages - the likes of Pascal, and Modula-2. This is in part due to the sea change which occurred in the early 1970s when structured programming emerged. The 1960s rendition of the Fortran standard, Fortran 66, had yet to fully immerse itself in structured programming, and languages which embraced the notion were to prosper more as instructional languages.

The C programming language, first introduced in 1971, has been used to teach introductory programming for more than three decades. However it was never designed for novice programmers, it is more of a transitionary language for programmers with an existing understanding of the fundamental concepts of programming, and how memory works. But why is this the case? Early reviews of the language sometimes concluded that the syntax was "irregular and messy" (**Ande90**), suggesting that "the designers and implementers of C have a lax attitude to syntax and semantics" (**Ande90**).

2. C - Language Synopsis

C was designed and implemented by Dennis Ritchie at Bell Labs in 1972. It was designed to implement the Unix operating system running on a DEC PDP-11 series machine. C evolved from the likes of B (**John73**) and BCPL (**Rich69**) (amongst tidbits from CPL, Fortran, Algol68, Assembler and PL/I). BCPL was typically used for implementing operating systems, and language processors. Both B and BCPL are typeless, hence the major advance of C was the addition of typed variables.

C is a systems programming language considered by some to be a "high-level, portable assembly language" because the language is really just a thin veneer above the underlying machine architecture, effectively designed to be "close to the metal". It is one of the few languages where the programmer can obtain an understanding of where any data object in the language is stored.

However this is somewhat of a double-edged sword because in order to user the low-level aspects of the language, one has to comprehensively understand the stack, heap, and other regions of memory allocation. C was designed as a very permissive language, to allow a wide range of applicability. The basic language was designed to be small in comparison to contemporary languages, with features such as input/output, math, and string processing being omitted (they are provided as standard libraries).

C has its advocates and detractors. Some of the basic issues with C concern its advanced nature. Mody (**Mody91**) put this quite aptly - "A concept is advanced when the tools for its assimilation and use have not been previously developed". For example having to deal with memory issues such as the stack and heap when never having encountered them before is advanced. Mody also had issues with C's progeny (**Mody91**): "I am appalled at the monstrous messes that computer scientists can produce under the name of `improvements'. It is to efforts such as C++ that I here refer. These artefacts are filled with frills and features but lack coherence, simplicity, understandability and implementability. If computer scientists could see that art is at the root of the best science, such ugly creatures could never take birth." There is nothing inherently wrong with low-level programming, however when novice programming are introduced to the craft they should be concentrating more on solving the problem, deriving an algorithm, and translating that algorithm to code, not on how and where the machine stores its bits.

2.1 Language problems

McIver and Conway (**McIv96**) identify language problems in the form of "grammatical traps", an example of which is the *syntactic synonym*, whereby two or more syntaxes are available to specify a single construct. This is a common occurrence in C and leads to a substantial increase in the learning curve, since the underlying concept becomes obscured in the students mind since it is no longer associated with a single distinct syntax. A classic example in C is the four ways to increment a variable by a factor of one: ++x, x++, x=x+1, x+=1 or perhaps the extracting elements from an array:

```
array[1]          *(array+1)        1[array]
```

A second difficult grammatical construct is the *elision* (**McIv96**), which implies the omission of a syntactic component, e.g. switch fall-through in C. C also suffers from failing to "enforce" some of it's syntactic rules, for instance the rule that every variable that appears in a **scanf()** must be preceded by an ampersand. If an ampersand is accidentally omitted, most C compilers will proceed to compile the code. Running the program will result in a segmentation fault, or a garbage value being stored.

3. Datatypes and memory

Datatypes are usually one of the first concepts introduced to novice programmers, and problems with learning C generally start here.

3.1 Basic memory

Nearly everything done in C requires some basic understanding of memory. Anyone can write a "Hello World" program, the real problems start when the input function **scanf()** is used for the first time. Consider the following **scanf()** example:

```
int x;
scanf("%d",&x);
```

A discussion of the relevance of **&** and the format specifier cannot be avoided. Other languages overload I/O functions, which generally make them easier to learn. So the novice programmer must learn the basics of memory, and how (and where) variables are stored. Compare this to the simplicity of Ada, which simply uses the **get()** function: **get(x)**.

3.2 Datatypes

All compiled and interpreted languages have datatypes of one sort or another. Part of the initial problem may lie with the abbreviated use of **int** and **char** to represent integer and character types respectively, in comparison to languages such as Fortran which use the full word. However beyond that it is the number of different types available (on a 128-bit MacBook Pro):

data type	memory (bytes)	range
short	2	-32768..32767
unsigned short	2	0..65535
int	4	-2147483648..2147483647
unsigned int	4	0..4294967295
long	4	-9223372036854775808.. 9223372036854775807
unsigned long	4	0..18446744073709551615
long long	8	-9223372036854775808.. 9223372036854775807
unsigned long long	8	0..18446744073709551615
char	1	-128..127
unsigned char	1	0..255
float	4	1.17549e-38..3.40282e+38
double	8	2.22507e-308..1.79769e+308
long double	16	system dependent

This many datatypes can become unwieldy for the novice user, especially when many of the ranges of values yielded by these data types are system independent. For example consider **long** vs **long long**. The C language specifies them as different types, however it is up to a particular platform to decide what size of integral number to use to support them, .i.e. types in C are not exactly portable. On many systems **long** and **long long** are equivalent in range. We haven't even mentioned the type qualifiers **register**, **const**, **volatile**, and **static**... and we won't. C also provides three real types: **float**, **double**, and **long double**.

Novice programmers are almost better learning to program using a language which has dynamic types for this reason alone. For instance, although Julia does dynamic typing, it does include types of the form **Int8**, **Int16**, **Int32**, **Int64**, which have an explicit width, and are standard over all platforms. Newer renditions of C, do provide similar constructs, for example **int8_t**, however they do require further nuances to make them work. For example to print an 8-bit integer requires:

```
int8_t x;

x = 24;
printf("%" PRIi8 "",x);
```

Note in this case, an **unsigned char** could also be used, as it has the values 0-255 when **unsigned** (the fact that the **char** datatype is really an integer does lead to some confusion). There is also the issue of **void**. The datatype **void** means no type, and is most commonly used in function declarations where the function does not return a type. A **void** function is essentially a procedure. In many other languages, the procedure and function are easily distinguished entities. Here are some details about **void**:

- a **void** is a type of function that is "not a function".
- the type of a pointer that is semi-valid
- the type of the parameters of a function, that takes no parameters

3.3 Typing

C is not considered to be a language with strong typing. Some of this has to do with the languages treatment of implicit conversions. In some languages such as Pascal, the only implicit conversions allowed are those that do not result in any loss of information (otherwise explicit transfer functions are required). C allows implicit conversion between any of its basic types or pointers, which can lead to unintended consequences for the novice programmer who does not understand what has happened. Examples include **float** to **int**, which causes truncation, and **double** to **float** which causes rounding. A classic example of a problematic equation is:

```
double pi, area, radius;
pi = 22 / 7;
area = pi * radius * radius;
```

If the novice programmer does not have a clear understanding of type conversions, they will not understand why the **area** calculated it incorrect. They may fail to understand that **22/7** is performed using integer division, which results in **pi** having a value of **3.0**.

4. Symbolic Confusion

Operators can be confusing in C, in part because some of them are used in different places to facilitate different things. A case in point is the **&** operator, which is used in association with pointers to have the compiler choose the address attribute of a variable, and as a bitwise AND operator.

4.1 Assignment

Programming languages which evolved from Algol (e.g. Pascal, Ada), use **:=** for assignment and = for comparison. C conversely uses = for assignment and == for comparison. The problem with this is that novice programmers often get the two confused and end up inadvertently writing an assignment where they intended a comparison. For example:

```
if (x = 0.001)
   y = x;
```

It is easy for a novice to make such as mistake, using = where they should be using ==. The act of assigning 0.001 to **x** will make the statement nonzero (true) every time.

4.2 Bitwise vs. logical

On a similar vein, it is just as easy to confuse & and &&, or | and || both of which do different things. A novice programmer who uses them in the wrong context, might not consider there to be a problem, because the program might actually work. One set (&, |, ^) are bitwise operators, which treat their operands as a sequence of bits. The other set (&&, ||, !) are logical operators (boolean, i.e. true or false).

Here is a simple piece of code which illustrates this (**Koen89**):

```
i = 0;
while (i<size && arr[i] != x)
   i = i + 1;
```

All the code does is search the array **arr** for an element that has the same value as **x**. It terminates if the element is not found. Replace the **&&** with **&**, and the code actually works as intended. This is more by sheer luck than anything else - both comparisons yield a value of 0 or 1, so the bitwise operator works. Were one to yield a non-zero value other than 1, the code would break. The novice programmer may get lucky, and this initial win lures them into a sense of complacency. Next time they use **&**, it may not function as intended.

4.3 The semicolon

Even the use of the semicolon can cause confusion to the novice programmer. In the simplest case a misplaced semicolon will cause a null statement, or an error message from the compiler, however in certain circumstances misuse will elicit no compiler issues. For example:

```
if (x > max);
   max = x;
```

The semicolon on the first line will not cause the compiler any issues. This code essentially implies that "if **x** is greater than **max**", an empty statement will be activated. The second line of code is activated regardless of the outcome of the **if** statement because it is not associated with the statement. This is not an uncommon situation amongst novice programmers. A better construct is to actually avoid the use of terminators completely, as is the case in Fortran and Julia.

4.4 Evaluating expressions

C won't allow the use of range expressions such as 0 < x < 100, which are quite valid in mathematics.

```
int x=101;
if (0 < x <= 100);
   printf("in range");
```

This expression is valid from the perspective of the compiler, but when the program is executed, the statement "in range" *will* be printed. This can confuse the novice programmer, as they may not have a complete understanding that **0<101** is evaluated first, which evaluates to 1 (true), and then **1<=100** is evaluated, which is also true.

5. Permissiveness

C is an extremely permissive programming language, which means it will compile almost anything that is syntactically correct, even if logically it does not appear as being correct. This sort of behaviour is okay for experts, but for novices it tends to hinder their ability to code a algorithm. It may be as simple as adding a spurious semicolon, but it can lead to hours of frustration. Examples are given throughout this paper. A case in point is how C allows an assignment statement within a conditional (see Section 4.1). There is also the problem of failure to check the bounds of arrays (see Section 8).

Other languages such as Ada, Fortran, or Swift are not as overindulgent when it comes to issues with code.

6. Incoherent control structures

6.1 Braces

C provides a series of control-structures which are seemingly incoherent to the novice programmer. Firstly there is the problem of consistent use of the block encapsulating braces, { }. This can lead to coding snafu's of this form:

```
if (x > 0.001)
   y = sqrt(x);
   z = x * log10(y);
```

Failure to enforce the use of { } to create a block structure means that the second statement will always be activated, regardless of the outcome of the conditional, because only **y=sqrt(x);** is associated with the conditional (if true). This is because C makes the use of braces optional if only one statement exists to be executed in a conditional or loop. If C were forced the use of { } (like Swift) less of these type of logic errors would occur. An alternative is to use control-structure terminators, such as those used in languages such as Fortran, Ada, or Julia. These reduce the requirement to remember specific format for specific control-structures.

6.2 Dangling else

The **dangling else** problem is a syntactic ambiguity that was first noticed in the original Algol 60 report in 1960 (**Naur60**). It was first published in 1963 (**Kauf63**) and there was a preponderance of papers (**Abra66**) that attempted to deal with the issue in the 1960s, to no avail, it remains one of the lasting contributions of Algol 60 to the programming community.

Consider the following program snippet:

```
if (x == 0)
   if (y == 0)
      printf("error");
else
   z = x + y;
```

The novice programmer might assume that this piece of code deals with the case when **x** equals zero, and when **x** does not equal zero. However C has a rule that an **else** is always associated with the closest unmatched **if**. Below the code shows what really happens (left) and what the programmer intended to happen (right).

```
if (x == 0){
   if (y == 0)
      printf("error");
   else
      z = x + y;
}
```

```
if (x == 0){
   if (y == 0)
      printf("error");
}
else
   z = x + y;
```

For the novice programmer it is easy to accidentally implement a dangling-else ambiguity in C, because C does not force the use of a block construct. Pascal, C++, Java all have similar issues. Other languages such as Fortran, Ada, and Julia avoid the problem by using **if** terminators, and Swift avoids it by forcing the use of braces, { }, regardless of the circumstance.

6.3 Switch fall-through

The **switch** statement in C is problematic because it is somewhat redundant. There is very little that the **switch** statement provides that cannot be similarly provided by a nested **if** statement. There are two distinct problems with **switch**. The first is fall-through. Fall-through occurs when one case is chosen, and its associated code executed, after which control passes to the code associated with the next case. Here is a classic example:

```
switch (level) {
   case 1: printf("Ground floor.");
   case 2: printf("Second floor.");
   case 3: printf("Third floor.");
   case 4: printf("Fourth floor.");
   default: printf("That floor does not exist.");
}
```

Here if the value of **level** is 2, what will be printed is:

```
Second floor.Third floor.Fourth floor.That floor does not exist.
```

This is because in languages like C, the default situation is fall-through. The fixed code looks like this (addition of the **break** statement stops fall-though occurring):

```
switch (level) {
   case 1: printf("Ground floor."); break;
   case 2: printf("Second floor."); break;
   case 3: printf("Third floor."); break;
   case 4: printf("Fourth floor."); break;
   default: printf("That floor does not exist.");
}
```

Ignoring the **break** statement to impede fall-through is easy to do. In other languages, only the code associated with the matching case is executed. For example Pascal:

```
switch x of
   1: write(x+1);
   2: write(x+2);
end
```

The second issue is the inability of **switch** to do ranges of values. This is a case where fall-through is beneficial. Here is an example

```
switch (p) {
   case 1: ;
   case 2: ;
   case 3: ;
   case 4: ;
   case 5: printf("values are between 1 and 5"); break;
}
```

Of course this kind of code is somewhat superfluous, and would be better represented as:

```
if (p>=1 && p<=5)
   printf("values are between 1 and 5");
```

6.4 Loop-d-loop

C provides three forms of loop: the **for** loop for numbered loops, the **while** for for indeterminate loops, and the **do-while** for loops that must iterate at least once. The most problematic of these loops is the **for** loop.

The problem with the **for** loop is two-fold. The first issue relates to the complexity of the syntax, which is also what makes it a powerful construct. Consider the following example:

```
for (i=1; i<=100; i=i+1)
   sum = sum + i;
```

The loop has an initialization condition, **i=1**, a conditional, **i<100**, and a loop index modifier, **i=i+1**. Often C programs substitute **++i** or **++i** instead of **i=i+1**, which causes the novice programmer to assume they do the same as **i=i+1**. In this context they do, however the novice programmer will easily transfer this knowledge to situations where it will fail.

The initialization is performed exactly once, when the loop is first initiated. The conditional component is performed every time before the loop body is entered. The loop modifier is applied after the loop body. This effectively makes the loop somewhat of a hybrid structure containing a branching instruction. This syntax is far more complex than that of say Fortran, Julia or Ada:

Fortran	Julia	Ada

`do i=1,10`
 	
`end do`
	`for i=1:100`
	
	`end`
		`for i in 1..100 loop`
		
		`end loop;`

The loop syntax is also more complex because it allows for multiple loops. Consider the following code, which includes two loop conditions:

```
int p = 0, q = 5;
for(i = 1, j = 1; i < p, j < q; i++, j++){
   printf("oops\n");
}
```

Because of a lack of understanding the programmer has used a comma operator instead of the **&&** (and) operator. The comma operator evaluates the expression on the left of the comma, discards it then evaluates the expression on the right and returns it. So technically this piece of code will print out "oops" five times (instead of only once if **,** is replaced with **&&**).

With reference to the **while** and **do-while** loops, there is very little difference between the two except for that the **while** loop will iterate 0 or more times, and the **do-while** will iterate at least once. The problem with these is syntactic inconsistencies (see below).

7. Lack of consistency

The way C operates in certain circumstances is very inconsistent. Such inconsistencies can result in the novice programmer second-guessing themselves. A case in point is the use of the address-of operator in **scanf()**. In the case of a normal variable, a **&** is required:

```
int x;
scanf("%d", &x);
```

However in the case of a string, (which in C is just a special character array) the **scanf()** is different, and does not require an ampersand.

```
char str[20];
scanf("%s", str);
```

This is because using the string name, **str**, tells **scanf()** to store the input string at the pointer associated with the string. However students will attempt to do this with a numeric array as well, in which case it will fail, as numeric arrays must be input element-by-element.

This is just one case of inconsistency. Other forms of inconsistency have to do with how control structures deal with blocks of code. For example, each of **for**, **while**, and **if-else** statements, allow parentheses to be ignored if there is only one statement to be activated.

`for (i=1; i<100; i=i+1)` `  sum = sum + i;`	`while (i<100)` `  sum = sum + i++;`	`if (i>1 && i<100)` `  prod = prod * i;`

If more than one statement is required, then block delineators, in the form of the braces, (or curly brackets, or even squiggly brackets) { and } must be used. For example:

```
i = 1;
while (i < 100) {
   sum = sum + i;
   i = i + 1;
}
```

However, when using the **do-while** loop, the brace-block construct is required.

```
i = 1;
do {
   sum = sum + i;
   i = i + 1;
} while (i < 100);
```

All other control structures also do not require a terminating semicolon, when a block construct is used, however **do-while** does. This lack of consistency can confuse the novice programmer.

8. Arrays

8.1 0-based Indexing

Arrays are tricky in C, and it's not only because they are indexed starting from 0. An array index that start from zero is the proverbial elephant in the room. Before B, few if any languages used zero-based indexing of arrays. Understanding why it is done in C-based languages does not make the use of 0-based indexing easier to understand for novice programmers. An array which has N elements is easily indexed with elements 1 to N. Moving to 0-based indexing means this shifts to 0..N-1, which can lead to issues with "off-by-one" errors in constructs such as loops. There are intensive debates on the value of zero-based indexing, but if arrays are modeled on the concept of matrices and vectors, where indices are naturally 1-based. Languages which use 0-bases are not really using indices, but rather *offsets*.

There are instances where the math is easier in 0-based indexing. One example is converting the elements of a 2D array into a 1D equivalent. Consider a 10x10 2D array:

```
0-based: index = 10 * r + c
1-based: index = 10 * (r-1) + c
```

Many languages, e.g. Fortran, Ada, Julia use 1-based indexing, indeed some languages such as Ada and Fortran take this a step further and allow arbitrary indexing (which is extremely effective from a problem-solving viewpoint, because the language can be moulded to the problem, rather than forcing the problem to be moulded to the language.

The other unintended artifact of 0-based indexing is that when coupled with **for** loops, novice programmers have a tendency not to index arrays properly. For example, a novice programmer may write code of the form:

```
int i, arr[100];

for (i=1; i<=100; i=i+1)
   arr[i] = i;
```

This code will compile without issues, but may end up assigning the wrong values: **arr[0]** will not get a value (or be set to 0), and **arr[100]** will overwrite a non-existent element. The novice programmer is again lured into having a false sense of security. (See Memory)

8.2 Multiple Array Indexing Syntax

Array syntax also exists in many forms, which can become confusing. Consider the following example:

```
int x, offset=1, arr[12];

x = arr[offset];
x = *(arr + offset);
x = *(offset + arr);
x = offset[arr];
```

Here **arr** is an array, i.e. a contiguous block of memory large enough to hold 12 **int** values. All four pieces of syntax are equivalent. This is because **[ ]** is used as an "add and dereference" operator, essentially implying that **arr[1]** is the same as ***(arr+1)**. The name of the array, **arr**, gives an address.

8.3 Arrays ≠ Pointers

C suffers from a confusion in the relationship between arrays and pointers, which Ritchie acknowledged this as a stumbling block for beginners (**Ritc93**). Arrays are not pointers.

8.4 Bounds checking

C has issues with overstepping array boundaries, or rather a failure to indicate when this occurs. It provides no run-time bounds checking. For example the following code will compile, and run properly, even though the storage associated with element 15 (**arr[14]**), has not been allocated. At some point a segmentation fault will occur, but C has no qualms about potentially overwriting a memory address.

```
int arr[12];

arr[14] = 1;
printf("%d", arr[14]);
```

This then becomes a critical safety issue. In C this is another significant source of undetected inconsistency, which can result in obscure failures. For the novice programmer, a small deflection from the array bounds may cause no immediate issues, and yet at some point critical data may be overwritten.

Most other languages provide some form of basic bounds checking at run-time. For instance attempting to add a sixth element to a 5-element integer array will result in the following error messages:

Julia	`ERROR: LoadError: BoundsError: attempt to access 5-element Array{Float64,1} at index [6]`
Ada	`raised CONSTRAINT_ERROR : bounds.adb:13 index check failed`
Fortran	`Fortran runtime error: Index '6' of dimension 1 of array 'x' above upper bound of 5`

8.5 Static vs. dynamic arrays

Arrays are also challenging because simple arrays are created on the stack, if the array is large enough, running the program will result in a *segmentation fault*, caused by not having sufficient memory int he stack. For example:

```
double arr[10000000];
```

The novice programmer is required to have an understanding of the size of the stack, and how the situation can be rectified. To move beyond the constraints of the stack requires knowledge of dynamic memory (i.e. the heap), and how to allocate it. This requires code of the form:

```
double *arr;
arr = malloc(N * sizeof(double));
```

It requires an understanding of how to allocate heap memory, and use functions like **malloc()**, (which allocates heap ,memory) and **free()** (which deallocates the memory). All this adds to the cognitive load of the novice programmer, distracting them from the task at hand. C does not have automatic garbage collection, partially because of overhead in performance. Failure to deallocate memory can result in memory leaks.

8.6 Strings

Strings in C are just character arrays, but a special case, because of the existence of the '\0' character to terminate them. Strings also have their own format character, **%s**, which allows them to be input and output as a complete entity, unlike numeric arrays, where the elements have to be input/output independently. This again leads to a lack of consistency. The problem with strings starts when they are declared:

```
char str[10];
```

This declares str to be a string, with, wait for it - the ability to store 9 characters, in elements 0 to 8, because the final element, **str[9]** in this case, is the string terminator '\0'.

There are more issues with the input of strings, see Section 13.

9. Pointers and memory

9.1 Memory

Some of C's most powerful features, are also its *Achilles Heel* when it comes to novice programmers. Top of the list are memory and pointers. To understand how to effectively program in C, a novice programmer has to understand how memory works, and more

importantly how pointers work. If pointers were an advanced topic it would be one thing, but they are essentially introduced the first time a **scanf**() function is used to read in a value.

```
int x;
scanf("%d", &x);
```

Most other programming languages perform a read from the standard input in a much more transparent manner, however C requires the programmer to state the data type being used in a formatting string, and use the address-of operator, **&** when specifying the variable where the value is to be stored. This is essentially specifying a function with a pass-by-reference parameter.

9.2 Pointers

C pointers are a low-level mechanism that should not be the concern of novice programmers, such constructs should be transparent to them. Even simple pointer declarations can be misconstrued, for example:

```
int* x, y;
```

This does not mean both **x** and **y** are pointers, but rather only that **x** is a pointer. The correct declaration is:

```
int *x, *y;
```

The actual placement of the ***** is often of concern as well, although all the following are equivalent:

```
int* x;     int *x;      int * x;      int*x;
```

9.3 Void

Further to this is the idea of **void***. It is hard enough to try and describe the concept of **void**, however void* is a complete other kettle of fish. Joyner (**Joyn96**) suggested it was the equivalent of an oxymoron, and that "A pointer to void suggests some sort of semantic nonsense, a dangling pointer perhaps?". It is also the complete opposite of **void** - void means no object of any type. **void*** means any object of any type.

9.4 Dynamic versus static memory

Most of the real issues begin when programmers start to use memory allocation functions like **malloc**(), or **calloc**(). Once novice programmers begin to meddle with memory, a new series of problems arise, often of a more subtle nature:

- Memory leaks: failing to free memory that was previously allocated.
- Invalid references: Continuing to use memory after it's been freed.
- Invalid array indices: Referencing outside the bounds of the array.

Once the novice programmer attempts to explore memory, they are then almost obligated to begin to understand the use of tools like **lint** (which flags programming errors, bugs, stylistic errors, and suspicious constructs), and **valgrind** (memory debugging and detects memory leaks).

9.5 Limited error messages

When a run-time issue does occur, the novice programmer is sometimes confronted with a message of the form: "**Segmentation fault**". Runtime errors, are not compiler messages per se, yet the functionality of the program can be attributed to the behaviors seeded to them by the compiler. Many of these messages are triggered by some lower level operation. In reality very few messages in compilers provide error messages which are self-descriptive. A message such as "segmentation fault" does indicate to the user that there is a problem with some aspect of memory management within the program, but does not give any indication of what actions should be taken. The only guidance such a message provides is to tell the user to interrogate regions of the program associated with manipulating memory.

10. Functions and parameter passing

10.1 Functions versus procedures

Functions in C are the only form of modularization. To turn them into procedures, i.e. subprograms that don't "return" a value, it is necessary to set the return type to **void**. No other language uses the term void, which is more likened to a black hole than anything else.

10.2 Parameter passing

There is also the issue of passing values to and from functions. In many languages this is done relatively transparently and therefore functions are easier to understand. In C, types must be explicitly declared. For example, consider the following function to calculate x^y. The programmer is required to specify both the type of the parameter being passed, and whether or not it is just being passed into the function, or a value is also being passed back.

```
double power(double x, int y)
{
   int i;
   double p=x;
   for (i=2; i<=y; i=i+1)
      p = p * x;
   return p;
}
```

Here both **x** and **y** are pass-by-value (i.e. their values go *in* to the function. Single variables must then be declared as pass-by-value, or alternatively if pass-by-reference if the value within the variable is to be modified.

Arrays passed to function are always passed as a pointer to the first element. This has the effect of reducing memory required because the whole array does not have to be copied. But it does break with the consistency convention, because it is essentially a special case, all other variables are passed-by-value by default.

Other languages allow either parameters which are more easily specified, or dispose of types for parameters altogether. This is inherently the value of dynamically typed languages. Here are some examples:

Statically typed	Dynamically typed

C
 	
`double betterGuess(double b, double x)`
`{`
`  return ((x + b/x) / 2.0);`
`}`
	Julia
	
	`function betterGuess(b,x)`
	`  return ((x + b/x) / 2.0)`
	`end`

Fortran
 	
`real function betterGuess(b,x)`
`  real, intent(in) :: b, x`
 	
`  betterGuess = ((x + b/x) / 2.0)`
`end function betterGuess`
	Python
	
	`def betterGuess(b,x):`
	`  return ((x + b/x) / 2.0)`

Ada
 	
`function betterGuess(b: in float; x: in float)`
`return float is`
`begin`
`  return ((x + b/x) / 2.0);`
`end betterGuess;`	

Obviously removing the requirements of parameter datatypes means that subprograms in dynamically typed languages become somewhat generic (parameter passing is handled transparently).

10.3 Pass by reference (sort-of)

The other issue with parameter passing in C, obviously has to do with pass-by-reference, which requires an understanding of pointers. Technically all parameters are passed in C using pass-by-value. Pass-by-reference is actually simulated by passing the address of a variable into a function, this is copied by the function, which then modifies the values at those addresses.

Consider the **power()** function modified to return the power as a parameter:

```
void power(double x, int y, double *p)
{
   int i;
   *p = x;
   for (i=2; i<=y; i=i+1)
```

```
        *p = *p * x;
}
```

Here you can see the function has effectively been turned into a procedure, with the return type modified to **void**. The variable ***p** is a pass-by-reference variable where ***** denotes the dereference operator.

10.4 Passing structures

It is also impossible to return structures like arrays using return (or for that matter returning multiple items like you can in Python and Julia), unless they are dynamic structures. Consider the two pieces of code shown below. The one on the left will not work because the function is trying to pass back a static array, which is not allowed in C. The program on the right allocates memory on the heap, and returns a pointer.

Won't work	Will work

`#include <stdio.h>`
 	
`int wontwork(int n)`
`{`
`  int i, x[4];`
`  for (i=0; i<n; i=i+1)`
`  x[i] = i;`
`  return x;`
`}`
 	
`int main(void)`
`{`
`  int arr[4];`
`  arr = wontwork(4);`
`  printf("%d\n", arr[2]);`
 	
`  return 0;`
`}`
	`#include <stdio.h>`
	`#include <stdlib.h>`
	
	`int* willwork(int n)`
	`{`
	`  int i, *x;`
	`  x = malloc(sizeof(int)*n);`
	`  for (i=0; i<n; i=i+1)`
	`  x[i] = i;`
	`  return x;`
	`}`
	
	`int main(void)`
	`{`
	`  int *arr;`
	`  arr = willwork(4);`
	`  printf("%d\n", arr[2]);`
	
	`  return 0;`
	`}`

The program on the left below does work, using a **struct** containing an array. This is a case where the most obvious method does not work. Finally the option on the right uses a more traditional approach in C, which passes the array as a parameter).

Will also work	Traditional solution

```c
#include <stdio.h>

typedef struct {
   int element[4];
} arrayRettype;

arrayRettype willwork(int n)
{
   int i;
   arrayRettype x;
   for (i=0; i<n; i=i+1)
      x.element[i] = i;
   return x;
}

int main(void)
{
   arrayRettype arr;
   arr = willwork(4);
   printf("%d\n", arr.element[2]);

   return 0;
}
```

```c
#include <stdio.h>

void willwork(int n, int x[])
{
   int i;
   for (i=0; i<n; i=i+1)
      x[i] = i;
}

int main(void)
{
   int arr[4];
   willwork(4,arr);
   printf("%d\n", arr[2]);

   return 0;
}
```

11. Failure to build on existing knowledge

C uses [ ] to index arrays, { } to encapsulate blocks of code, ( ) for functions calls. The use of square brackets for array indexing is consistent with what exists in fields such as mathematics where [ ] are used to specify matrices. Parentheses, or curved brackets, ( ), are classically used in mathematics to indicate an order of operations, for which they are also used in C. However C also uses them to indicate arguments in functions, which again makes this consistent with its mathematical underpinnings. Curly brackets are rarely used outside of programming. In prose they may be used to mark words that should be grouped together, and in mathematics they delimit sets, however they aren't commonly used, and therefore few people have experience with them. C is not the only language that uses curly brackets, the syntax originated with BCPL in 1966, and has been used consistently by languages that have been heavily influenced by, or descended from C. Using begin-end to denote a block is more logical for the novice programmer, because they understand clearly what they mean.

Another example is the exponentiation operator in C. There isn't one. So a programmer wishing to program x^y in a calculation has to explicitly use the math function **pow()**[1]. This is contrary to many other programming languages which use operators such as ****** (Fortran, Ada, Python) or **^** (Julia). This builds on existing mathematical knowledge, where **^** can be used to express exponentiation. Another issue is the use of **log()** and **log10()** in C, to denote the mathematical functions **ln()** and **log()** respectively. This confuses the novice programmer who reads a mathematical equation containing **log**, and interprets it as the **log()** function, rather than **log10()**.

One of the more complex examples involves indexing of arrays. As arrays in C use 0-based indexing, they fail to build on existing knowledge of indexing in mathematics - that of matrices

[1] C requires the use of the **math.h** library, which unlike any other library generally requires the use of the **-lm** flag when compiling.

and vectors. In the real-world, groups of similar items are counted starting from 1. When you count the number of books on a shelf, you start with book 1, etc.

In mathematics, a clause like **1≤x≤100** is quite understandable - **x** has the values 1 to 100. The novice programmer may be tempted to right something similar in C:

```
if (1 <= x <= 100)
```

However this conditional statement could produce either a correct or incorrect result, and it is all predicated on how C evaluates the conditional. If **x** has the values 1 to 100, C will first evaluate **1<=x**, which will be true (1), and then evaluate **1<=100**, which will always be true. If however the value of **x** is 101, the end result will also be true. This makes the expression "semantically valid, although pragmatically useless", because conditional expressions like this are evaluated as **int**'s not booleans (this is termed semantic non-redundancy (**Mody91**)).

To make this work in C requires use of two separate conditions logically anded:

```
if (1 <= x && x <= 100)
```

This requires the novice to rethink the way the clause is written. The code would be much simpler if written in the original manner, as the novice could easily transfer their existing knowledge. This is done in Julia:

```
if (1 <= x <= 100) then
```

12. Code brevity and condensation

Sometimes C code becomes too compact for any programmer to even understand, let alone a novice (a relic of its quasi-assembler like behaviour). This is precipitated by the use of compact operators such as ++ (which means increment a variable by 1. The increment and decrement (--) operators were designed as a high-level assembler for the DEC PDP machines C was implemented on.

The increment operator can be used in two forms: ++x and x++. The first means increment x, then use its value, the second means use its value then increment x.

```
int x, y=6;
x = y++; // x is assigned the value 6, and y is incremented to 7
x = ++y; // y is incremented to 8, and x is assigned the value 8
```

In the first instance, the value of x is 6, while in the second instance the value of x is 8. This sort of code can be extremely perplexing for novice programmers. It can also lead to side effects because of misuse, e.g. writing **x=x++** instead of simply **x++**.

Here is another cryptically condensed piece of code which swaps two integers without using a temporary variable:

```
x^=y^=x^=y;
```

The use of these operators means that the novice programmer is consumed with attempting to understand how they work, rather than the implementation aspects of their algorithm. Tricky, obfuscated code is always problematic for novice programmers.

Novice programmers often see loops written in the manner where the loop variable modification is performed using i++, for example:

```
for (i=1; i<100; i++)
   sum = sum + i;
```

They then go on to use this in other code, for example:

```
int x[100], y[100];
i = 0;
while (i<100)
   y[i++] = x[i];
```

This will of course not work as intended: In the first iteration of the **while** loop, **i** will be used, i.e. **y[0]** and then incremented to **1**, so **y[0]** is assigned the value of **x[1]**. The final loop will see **y[99] = x[100]**, which does not exist, so a garbage value will be assigned. The challenge is that many C compilers will post a warning with code such as **arr[i++] = i,** of the form:

```
warning: operation on 'i' may be undefined
```

Here is further example:

```
int i, arr[100];

for (i=1; i!=100; i=i+1){
   arr[i] = i++;
```

A horrific piece of code which actually increments the loop variable within the loop. The novice programmer had intended to set each array element to 1 more than its index, e.g. **a[1]=2**, however inadvertently increments **i**, which has the effect of bypassing the loop condition (**i** will increment as 1, 3, ..., 99, 101 etc. bypassing 100, and eventually crashing or causing a segmentation fault. This is in addition to starting to index the array at 1. If the initial loop value were changed to 0, the loop would terminate, but the array contents would still be wrong. The correct loop would look like:

```
int i, arr[100];

for (i=0; i!=100; i=i+1){
   arr[i] = i+1;
```

This shows how easy it is to make mistakes in C syntax that are accepted by the compiler as valid syntax (because they are).

Here is a piece of complex code:

```
/* Loosely based on OpenBSD source code */
char* strcpy(char *to, const char *from)
{
   char *save = to;
   for (; (*to = *from) != '\0'; ++from, ++to);
   return save;
}
```

The problem with this function which copies the contents of one string into another start from the time it is used. Convention would have it that copying string **A** to **B** would be achieved using the syntax **strcpy(A,B)**, however in C, this is **strcpy(B,A)**. However the novice programmer who dares to view the actual code will be confronted with an assignment within a loop conditional, and a loop without an initial condition. Additionally it returns the copied string via parameters *and* a return statement.

In many respects the operators in C also reflect the underlying hardware of the original design. However allowing things like assignments in expressions encourages side-effects, and makes programs less reliable. For example consider the following pieces of code which read characters into an array until end-of-file (**Feue82**):

C	Pascal

`while((s[i++] = getchar()) != EOF);`	`while not eof do`
	`begin`
	`  s[i] := getchar();`
	`  i := i + 1`
	`end`

13. Input snaffus

Some of the simplest issues with C have to do with the use of input functions, and more specifically, the cases where the **<return>** key causes a problem. This is predominantly because C stores keyboard input in a "buffer" (a piece of memory dedicated to standard input), and sometimes when the wrong combination or inputs occurs, an errant <return> is read where it shouldn't be, causing confusion for the novice programmer. Take the simplest case, where two integers are read from standard input:

```
int x, y;
scanf("%d %d", &x, &y);
printf("x = %d, y = %d\n", x, y);
```

When this code is run, the user can enter one integer, then another and there will be no issues (even if there are 20 spaces between the two integers). Even the following code works with an integer followed by a string:

```
int x;
char str[100];
scanf("%d", &x);
scanf("%s", str);
printf("x = %d, str = %s", x, str);
```

However when the **scanf()** to read the string is replaced by a call to **fgets()**, as in:

```
int x;
char str[100];
scanf("%d", &x);
fgets(str, 100, stdin);
printf("x = %d, str = %s", x, str);
```

When the following is entered:

```
12 <return>
```

the program "skips" the ability to enter a string. This is because the string that is "read in" is the <return> because it is a character (albeit an invisible one). However if the following is entered:

```
12 hoodwinked <return>
```

then **x** is 12, and **str** is " hoodwinked" (with a leading space).

Confused? Now imagine a novice programmer dealing with this. Again it related to the programmer having a clear understanding of the underlying structure of the language, i.e. that it uses a buffer to store standard input, and that buffer is cleared or not, depending on how input is input. This sort of nonsense usually rears its ugly head when a novice programmer attempts to make a menu of the form:

```
char c;
printf("q to quit\n");
do
{
   printf("Enter a character\n");
   scanf("%c", &c);
   printf("%c\n", c);
}
while (c != 'q');
```

When the code is run the following happens:

```
q to quit
Enter a character
e
e
Enter a character


Enter a character
a
```

```
a
```

It skips a character, where the user entered **e<return>**, and the **<return>** is read from the buffer and output.

One of the other issues has to deal with the fact that there is more than one way to perform input. Take the example of strings. There are at least three possibilities here: **scanf()**, **gets()**, and **fgets()**. Each of these has subtleties that will cause problems for the novice programmer. The use of **scanf()** is fine, except that it tends to read in more characters than allowed (well that and it breaks the language consistency, because although it is an array, it gets special treatment using the **%s** format specifier, and requires no **&**). Here is a sample piece of code:

```
char str[10];
scanf("%s", str);
printf("str = %s, length = %d\n", str, strlen(str));
```

Now consider the following inputs and outputs:

input	output
elephant	str = elephant, length = 8
deterministic	str = deterministic, length = 13
the cat in the hat	str = the, length = 3
supercalifragilisticexpialidocious	str = supercalifragilisticexpialidocious, length = 34 Segmentation fault: 11

The first input works fine. The second allows a string 13 characters in length to be stored in a container that should store 9 at most. The third input only reads in up to the blank, which shows a failure in C to really identify what a string is. Is a blank space not a character? Is so, why is the string input terminated when **scanf()** hits the blank? Dark magic? The final case shows that there are limits to what **scanf()** will read in, although technically it *does* read in the 34 character word, store it in **str**, print it out, and then barf.

Magic right? But it can be fixed. With **gets()** right? Wrong. **gets()** is a security risk and should not be used. In fact when the novice programmer runs a program containing gets(), it will produce a warning of the form: "**warning: this program uses gets(), which is unsafe**". Not during compile mind you, which would likely be better. (Why not make **gets()** obsolete?)

Finally we get to **fgets()**, which admirably fixes most of the problems with string input - except that it is a function for file input, and needs the file set to "standard input" (**stdin**). Here is a code snippet:

```
char str[10];
fgets(str, 10, stdin);
```

```
printf("str = %s, length = %d\n", str, strlen(str));
```

It allows at most 9 characters to be read in. Here are some sample runs:

input	output
elephant	str = elephant , length = 9
deterministic	str = determini, length = 9
the cat in the hat	str = the cat i, length = 9
supercalifragilisticexpialidocious	str = supercali, length = 9

In most cases it worked as intended. The one spurious output is the first input, where because the word was less than 9 in length, **fgets**() also stored the <return>, hence the break to a newline in the output. So, if a novice programmer would have to strip the <return> and move the end-of-string pointer to use this string in further processing.

Strings suffer from the same issues suffered by other constructs in C - multiplicity - many ways of doing the same thing, which tends to cause confusion for the novice programmer.

14. Hoodwinking the compiler

It is also easy to hoodwink the C compiler (although they have become somewhat wiser over the years). Here is a case in point, a function that calculates the Greatest Common Divisor of two numbers:

```
#include <stdio.h>

int gcd(int x, int y)
{
   int r = y % x;
   if (x == y)
      return x;
   else if (r != 0)
      gcd(r, x);
}

int main(void)
{
   int x,y,z=0;
   scanf("%d%d",&x,&y);
   z = gcd(x,y);
   printf("%d\n", z);
   return 0;
}
```

Now technically there is an issue with the function **gcd()** only containing one return statement, but this is only acknowledged if the **-Wall** compiler flag is used during compilation. Here is the message returned by the gcc compiler used to compile the code (gcc 4.9.3):

```
gcd.c:10:1: warning: control reaches end of non-void function
```

But the code *does* compile. When the executable is run, the function actually returns the correct value, even though there is no explicit **return** statement - and it is the value of **r** that is returned.

Sometimes C just allows crazy things to compile. Consider the following code:

```
int i;
scanf("%s", i);
```

This will compile without issue (only **-Wall** produces a warning), and crashes at runtime (with a **Segmentation fault**).

15. C code is just plain crazy

Perfect examples of the looniness of C code can be found in the 2nd edition of K&R, (Kernighan and Ritchie, The C Programming Language). You need to be what some have termed a *syntactician* in order to figure it out. Here is one derived from section 5.12 of the book, aptly entitled "Complicated Declarations". The program includes an "array of pointers to functions", a function that returns a pointer, and a pointer to a function.

```
#include <stdio.h>

// Function returning a char
char x1() { return 'm'; }
// Array of pointers to functions returning char
char (*x2[])() = {&x1};
// Function returning a pointer to the above
char (*(*x())[])() { return &x2; }

int main(void)
{
   // Pointer to a function returning char
   char (*x3)() = **x();

   printf("This is the value: %c\n", x3());
   return 0;
}
```

Is it any wonder novice programmers find programming in C challenging?

16. Conclusion

C is a powerful language, and although this paper has documented its idiosyncrasies, they are predominantly associated with teaching programming to a novice, i.e. someone who does not have the existing knowledge to leverage the use of C in an effective manner. Some are inherent design flaws in the language. There are many things about C which make it powerful, for example its ability to perform low-level programming, and direct memory manipulation, however these are not for the novice. One thing C does not suffer from is what is aptly termed "creeping featuritis" - languages that are too complex: C++ with its 50 distinct operators with 17 levels of precedence, and 62 reserved words (C only has 32); or languages which are object-oriented, but have elements of procedural programming, again like C++. synonym

References

(**Abra66**) Abrahams, P.W., "A final solution to the dangling else of ALGOL 60 and related languages", *Communications of the ACM*, 9(9), pp.679-682 (1966)

(**Ande90**) Anderson, B., "Type syntax in the language 'C': an object lesson in syntactic innovation", ACM SIGPLAN Notices, 15(3), pp.21-27 (1980)

(**Feue82**) Feuer, A.R., Gehani, N.H., "A comparison of the programming languages C and Pascal", Computing Surveys, 14(1), pp.73-92 (1982)

(**John73**) Johnson, S.C., Kernighan, B.W., "The programming language B" (1973)

(**Joyn96**) Joyner, I., "A critique of C++", (1996)

(**Kauf63**) Kaufe, A., "A note on the dangling else in ALGOL 60", *Communications of the ACM*, 6, pp.460-462 (1963)

(**Koen89**) Koenig, A., C Traps and Pitfalls, Addison Wesley (1989)

(**McIv96**) McIver, L., Conway, D., "Seven deadly sins of introductory programming language design", IEEE Software Engineering: Education and Practice, pp.309-316 (1996)

(**Mody91**) Mody, R.P. "C in education and software engineering", SIGCSE Bulletin, 23(3), pp. 45-56 (1991)

(**Naur60**) Naur, P. (ed), "Report on the algorithmic language ALGOL 60", *Communications of the ACM*, 3, pp.299-314 (1960)

(**Rich69**) Richards, M., "BCPL A tool for compiler writing and systems programming", in Proc. AFIPS Spring Jt. Computer Conference, 34, pp.557-566 (1969)

(**Ritc93**) Ritchie, D., "The development of the C language", https://www.bell-labs.com/usr/dmr/www/chist.html (1993)

TECHNICAL REPORT

June 2018