Why C has Pointers

"What?" & "How?" are the two questions normally answered by C programming articles. This article is on the "Why?". The reason why something as confusing as pointers are so commonly used in C is rarely mentioned but is very helpful in understanding how to use them. However, just in case you have not yet come across pointers or are not familiar with C at all, I will start with a brief description of what they are (definitions) and how they are used (syntax). Don't expect lots of that here, there is plenty of that in those standard programming books, just enough make sense of what follows.

Quick Introduction to Pointers

Definition: A pointer is a variable containing the address of another variable.

A variable is just a labelled place to store some data. For example one can make variable, with the unimaginative name of '`Variable1`', to store an integer in C with the command

```
int Variable1;
```

, store the number '96' in it with

```
Variable1=96;
```

and print it out with

```
printf("%d",Variable1);
```

. Instead of referring to this data store by name, one can refer to it by its address in the computer memory. This address can itself be put in another variable in C, such a variable being called a 'pointer' because its value points to where the value of the original variable is rather than being such a value itself.

To do this one puts a '`&`' infront of a variable to mean "give me the address of " and a '`*`' infront of a pointer to mean "give me whatever is that the address ". So one could make point a pointer, unimaginatively called '`Pointer1`', to the above '`Variable1`' by the command

```
Pointer1=&Variable1;
```

which stores the address of '`Variable1`' in the pointer variable '`Pointer1`'. To copy the value of '`Variable1`' to another variable, '`Variable2`', one can use

```
Variable2=*Pointer1;
```

which would have the same effect as

```
Variable2=Variable1;
```

or print it out with

```
printf("%d",*Pointer1);
```

Of course, one could make things really confusing with lots of pointers to pointers (or even pointers to pointers to functions returning pointers to arrays of pointers to ... !) etc., and many books do just to show off, but the basic syntax above is almost enough for the current purposes. One confusing bit does need to be explained if one is to use pointers, that is the syntax of creating a pointer variable in the first place. For '`Pointer1`' above, it would be

```
int *Pointer1;
```

whereas one might expect an '`&`' in making a pointer. The reason for the reverse, an '`*`', is because the '`int` ... ;' command specifies that the thing mentioned is an integer and C works out from the fact that '`*Pointer1`' is an integer that '`Pointer1`' itself must be a pointer to a thing which is an integer. The general method for creating such pointery variables is to work from the inside out: take the name of the variable; add whatever '`*`'s, brackets or whatnot would convert it into a simple type, like an integer, for which there is command to create; and use that command on the name with all the added bits left in place. In this case it was simply taking '`Pointer1`', adding a '`*`' to convert it to an integer & using 'int' on the combination.

That's enough about what they are and how to use them in C, but why are they there in the language at all?

Why?

C was developed when computers were much less powerful than they are today and being very efficient with speed and memory usage was often not just desirable but vital. The raw ability to work with particular memory locations was obviously a useful option to have. A few tasks these days, such as programming microcontrollers, still need this. However most modern programmers do not need such fine control and the complications of using pointers make programs less clear to understand and add to the ways in which they can be go wrong. So why are pointers still used so much in C & its successor, C++?

The reason is that pointers are used to bodge into C some vital features which are missing from the original language: arrays, strings, & writeable function parameters. They can also be used to optimize a program to run faster or use less memory that it would otherwise.

One of the complications when reading C programs is that a pointer could be being used for any, several or all of these different reasons with little or no distinction in the language so, unless the programmer has put in helpful comments, one has to follow through the program to see what each pointer is used for in order to work out why it is there instead of a plain simple variable.

I will cover different the uses individually to keep it simple.

Pointers used as Arrays

Arrays are very useful but C does not have array variables. This might seem incorrect as there is a C syntax for working with arrays. For example one can create an array, unimaginatively called '`Array1`', of 9 integers by

```
int Array1[9];
```

, store the number '96' in the 8th element (elements being numbered from 0 not from 1) with

```
Array1[7]=96;
```

and print it that element with

```
printf("%d", Array1[7]);
```

but this is really just working with pointers with an alternative syntax.

In C, an array variable is just a pointer to a chunk of memory where the elements are stored in order and expressions like "`Thing1[Thing2]`" are treated as "`*(Thing1+Thing2)`", i.e. take pointer '`Thing1`', advance by '`Thing2`' locations and use whatever is found at that place in memory.

Therefore the line

```
Array1[7]=96;
```

is identical to

```
*(Array1+7)=96;
```

because 'Array1' is just a pointer to the start of the memory chunk and the '[7]' moves it along 7 positions then uses what is at that position there. A particularly confusing special case is that "*Array1" can be used, of course, as a short way of typing "Array1[0]" but looks nothing like an array at all!

Besides being misleading, it has other problems including: the program having to remember the length of the array separately; there being no facility to pass a copy of an array as a function parameter like a normal variable (just the pointer gets copied instead); and manipulating an array requires the programmer to write functions that manipulate the array's elements individually.

How did this primitive implementation of arrays survive? It was probably because more dedicated array implementations in other languages were not complete and were less easy to bodge work-arounds into. For example FORTRAN-77 has a dedicated array system but it allocates a fixed amount of storage for its array during compilation so one cannot use them for arrays if the length depends on the data when the program is run later. In C one can bodge up an such an array by bypassing the array allocation line, explicitly creating a pointer and a chunk of memory of the correct size and using that instead. For example, if the length desired is in a variable called 'Length1',

```
int *Array1;
```

will create the pointer and

```
Array1=(int*)malloc(Length1*sizeof(int));
```

will use 'malloc' (meaning "memory allocate") for reserving a chunk of memory based on the number of bytes needed for an integer as found by 'sizeof'. Not that the '*' before 'sizeof' is not a "give me whatever is at address " instruction but a merely a normal multiplication. (Anyway, don't bother learning the syntax of C's 'malloc' because it has now been superseded by C++'s 'new' which does the same thing neater; the equivalent line being "Array1=new int[Length1];".)

The pointer 'Array1' could then be used just as if was created as an array because there is no difference in C. All one has to do different is remember, that as the memory was explicitly reserved, it must be explicitly released using the 'free' command (use 'delete' instead if one used 'new' instead of 'malloc') when the program is finished with it. Otherwise the memory gets filled up with leftovers which the computer does not know are no longer needed.

Pointers used as Strings

Similarly, C appears to have text string variables which can be created like

```
char *String1;
```

set like

```
String1="Some text";
```

and printed out like

```
printf("%s",String1);
```

but, like arrays, they are really just pointers. In this case, this is not even hidden in the syntax of the command to initially create the string where it is blatantly created as pointer to a variable of type 'char'.

Indeed, the second line of the above example, "`String1="Some text";`", could be a bug because it does not do what one would expect it to do were it a real string variable. It does not load the characters 'Some text' into some storage created for that string in the first line.

A normal string in C is nothing but a chunk of memory containing the characters as bytes and ending with a byte of value zero. The thing which is used as a string variable is simply a pointer to the first of those bytes. C treats text in double quotes as a string in the same fashion, storing it in the compiled program with a zero byte at the end, so the second line merely points the record of the start of the string (the pointer created in the first line) at the first byte of that.

This becomes apparent if one tries to tries to modify the string into 'Some test' by replacing a letter:

```
String1[7]='s';
```

This will probably cause a compiler error or memory access crash when run in modern systems that don't allow self-modifying executable (some older systems allowed it but it is now generally considered a security risk) or if the executable is unalterable in firmware. This is because the bytes of 'Some text' aren't being copied into some mutable real string variable but are just sitting there in the program executable (ignoring misc. behind the scenes optimisation from the compiler) so that line tries to alter a byte in the executable itself!

This is not to say that strings cannot be modified. As long as the memory pointed to can be modified, the string can be. For example the following is valid:

```
char String2[18]="Some text";
String2[7]='s';
printf("%s",String2);
```

works. This is because the first line is a C shorthand for

```
char String2[18]={'S','o','m','e',' ','t','e','x','t','\0'};
```

i.e. make allocate an array of 18 '`char`'s on the stack (alterable memory allocated automatically) and initialise it with the bytes of that string. Note that 18 bytes only gives space for strings up to 17 characters long due to the zero byte on the end. Also note that it assumes one byte per character, increase the number for multibyte characters.

Similarly one could do it with memory on the heap (alterable memory allocated by explicit programmer command):

```
char *String3;
String3=(char*)malloc(18*sizeof(char));
String3[0]='S';
String3[1]='o';
[...]
String3[9]='\0';
String3[7]='s';
printf("%c", String1[7]);
```

There are lots more programming complications arising from C's raw pointer bodged-up strings (& the speed/memory advantage these days mainly only of interest to microcontroller & other firmware programmers) and this is a pointers article not a strings article so I'll quickly list a few more major 'gotchas' then move on:

- The string cannot contain the ASCII null (code zero) character as that would look like a premature string end. If that is really needed then an alternative string representation, in which one stores the length in a separate variable, is sometimes used and interconversions are required.
- This restriction on zero bytes also causes problems with some encodings of multibyte characters such as UTF-16 (UTF-8 is specifically designed with backward compatibility but instead has the

problem of variable numbers of bytes per character making allocation sizing difficult and jumping to individual characters inefficient).
- One cannot simply concatenate strings or even copy from one string to another (using '`=`' would instead make the two pointers point to the same memory rather than a duplicate) without writing routines to process the strings a character at a time or using library functions.
- It is easy to forget the maximum length one has allocated to a string or forget to keep a zero byte on the end allowing the program to run off the end of the string into arbitrary memory. This is the classic "buffer overrun" bug that has so often been exploited to break into networked programs.
- As with arrays (of course) strings can be made to any length in C with '`malloc`' (or '`new`') and the string length updated whilst reading strings of initially unknown length but it is hassle to have to create a new string, copy the characters across one by one, release the memory of the old string and change the pointer to point at the new string just to change the length of a string. Hence programmers have very frequently just guessed a size they have thought ample and hoped for the best (sometimes not even checking for overflow) storing up limitations & vulnerabilities as problems for the future.

Pointers used as Writeable Function Parameters

When a function or subroutine is called in C, a copy of its parameters are passed to the function. For example if

```
int Variable1=96;
Subroutine1(Variable1);
printf("%d",Variable1);
```

is called on this function

```
void Subroutine1(int Parameter1)
{ printf("%d",Parameter1);
  Parameter1=Parameter1+1;}
```

then only '9696' will be printed , not '9697', because only the local copy, '`Parameter1`', of the value of '`Variable1`' in the subroutine, not the original variable itself, has been altered. In some languages, like FORTRAN-77, it is the variable itself which is as the parameter so the equivalent of this subroutine would work.

Sometimes a copy is what is desired (particularly if one wants no risk of a subroutine from someone else changing the variable or if the routine is to be used recursively) and sometimes the variable itself is desired (particularly if the original variable needs to be altered). C allows the choice. Unfortunately it does not offer the choice in a clear manner like Java where they are '`in:`', '`out:`' & '`inout:`' as per the directions the data goes (the C in-only method being equivalent to '`in:`', the FORTRAN-77 both-ways one equivalent to '`inout:`'). C instead, as you have now come to expect, bodges it up with pointers.

If one has a pointer to the variable instead of the variable itself as the parameter then, even though that pointer gets copied, one can use its value, the memory address, to directly access the memory where the value of the original variable is stored.

So for example altering the above example to

```
int Variable1=96;
Subroutine1(&Variable1);
printf("%d",Variable1);
```

and

```
void Subroutine1(int *Parameter1)
{ printf("%d",*Parameter1);
```

```
    *Parameter1=*Parameter1+1;}
```

will produce '9697'.

This is a rather messy thing to do in that requires altering not just the function parameter list & the calling line but every reference to the parameter in the function but that is the way C does it.

C++ has the option of passing variables themselves not copies but the old method has stuck from tradition and because it makes it clear when calling functions as to which variables it is safe to assume won't be altered by the function. The syntax is also horridly confusing: to specify that a variable is to be passed directly not as a copy, one precedes its name in the function's parameter list with a '`&`' - exactly the same symbol as used to me "give the address of " but with a different meaning!

Pointers used for Optimization

The use of pointers for passing alterable variables to subroutines has an another use - it can save memory and run faster because it does not have to duplicate the data. For the above example, there would be no saving because an integer is as small in bytes as the pointer which is being copied instead. However, for more complicated variables in programs that must run very fast, it can be a vital trick.

In C, the biggest variables are 'structures'. Structures are user-defined variable types which are used to keep a set of different variables associated with a single thing together. Structure variable types are user-defined with the '`struct`' command, e.g.

```
struct UserType1 {int Part1;char* Part2};
```

defines '`UserType1`' as a being a variable type that is a group of an integer variable and a string variable labelled '`Part1`' & '`Part2`' respectively, and created just like other variables except that 'struct' precedes the name of the variable type, e.g.

```
struct UserType1 Variable3;
```

creates '`Variable3`' as a variable of this new type, '`UserType1`'. The constituent variables within a structure variable are accessed by putting a '`.`' and the appropriate label after the structure variable name, e.g.

```
Variable3.Part2="Some text";
printf("%s", Variable3.Part2);
```

would set the string part of the variable to 'Some Text' and print it out.

The use of pointers with structures is so common that there is an alternative syntax available to make such program lines look less messy. This is "`Thing3->Thing4`" which means the same as "`(*Thing3).Thing4`", i.e. "get the structure pointed to by Thing3 and use the constituent labelled Thing4". For example, if one had only a pointer to '`Variable3`'

```
struct UserType1 *Pointer2;
Pointer2=&Variable3;
```

then one could use

```
Pointer2->Part1=96;
```

to set the the '`Part1`' constituent of '`Variable3`' to 96 instead of

```
(*Pointer2).Part1=96;
```

which is not much difference in this case (only one character shorter) but saves a lot of nested parentheses when structures contain pointers themselves, e.g. "`Pointer3->Part11->Part12->Part13`" instead of “`(*(*(*Pointer3).Part11).Part12).Part13`”.

Once again the ‘`->`’ structure system is just, like the ‘`[]`’ array system, pointers in disguise.

Summary

They look complicated because it is not obvious what they are doing and they can be combined many levels deep but normally they are just being used to bodge in several of C’s missing features.

It can get very, unnecessarily, confusing if these different uses of pointer are combined. For example, even if one is just passing an alterable array of strings to a subroutine, one has to work with a pointer to pointers to pointers to characters with the pointers being used in 3 different ways at once, none of them being the raw memory manipulation pointers were essentially designed for!
