Transactions on Engineering and Computing Sciences - Vol. 11, No. 6
Publication Date: December 25, 2023
DOI:10.14738/tecs.116.15504.
Wetoyi, A. O. (2023). The Beauty of Pointers. Transactions on Engineering and Computing Sciences, 11(6). 106-118.

Services for Science and Education

The Beauty of Pointers

Austin Owino Wetoyi
Dept. of Mathematics, Physics and Computing,
School of Sciences and Aerospace Studies,
Moi University, Eldoret, Kenya

ABSTRACT

Talk to any folk familiar with programming for an advice on pointers and the most likely advice you will get is avoid pointers at all costs. Very little is said about pointers that show them in good light. We try to show in this paper that pointers are probably the best feature that a programming language can offer. There are too many beautiful (and even ugly) things that can only be done with pointers. Pointers are the most powerful feature of languages that allow their use. Obviously extreme power requires extreme responsibility. Careless use of them is known to lead to the most difficult logical errors to fix. Pointers are the closest thing to hell before you understand them. Pointer bugs can cause a program crash in random ways which makes them more difficult to debug. Nonetheless, even with their problems, pointers are irresistibly powerful programming constructs. Many advanced programming and debugging problems only make sense with a complete understanding of pointers and memory. There's a lot of nice, tidy code you can write without knowing about pointers. But once you learn to use the power of pointers, you can never go back.

Keywords: Pointer, Reference, Bug, Memory

WHAT IS A POINTER?

Memory is typically laid out as a sequential series of equally sized usable memory locations; we usually refer to these locations as eight-bit bytes but actually the size of each space depends on the architecture of the particular machine and is usually called that machine's word size. Each of these slots can be uniquely distinguished from all other spaces by an integral numeric value known as its address. Think of the memory that all machines use as made up of bytes that have sequential addresses starting at zero and going up to however much memory you have in your computer. Essentially the memory is a one-dimensional array indexed by addresses. Figure 1 illustrates this view of the computer memory.

VARIABLES

A variable is a memory location that a program can access for inspection (reading or retrieving or examination) or for manipulation (writing or storing into). To access a variable, the program uses the variable's name (if it has any) or its address. A variable is also known an object. The number of bytes an object occupies is known as its size, and this size is determined by the type of the object. The address of an object is the address of the first byte of the bytes the object is sitting on or its base. A pointer is a memory address. An address of an object is known as a pointer to that object. If a variable has a name, the name is also often referred to as a variable.

A variable's content may also be a pointer (memory address). Such a variable is also often called a pointer (strictly a pointer variable).

Address	Content
0000	
0001	"m"
0002	
.	
.	
.	
n - 1	

Figure 1: A view of the computer memory showing the second byte (byte number one) storing the character "m"

Figure 2 illustrate three variables named letter, number and numPtr with the last one numPtr being a pointer to one of the first two variables.

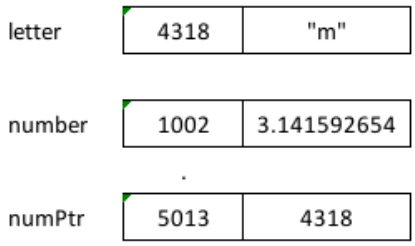


Figure 2

All languages have a means of creating variables. In a compiled language like C and C++, there are three ways in which a variable can be created:

- When a declaration statement is executed.
- When passing parameters by value.
- Manually requesting for space from the free memory area.

Whichever method used; some languages require that you must specify the type of information the variable is going to store. As stated above, the type may require more than one byte in this case the address of the variable is the address of the first byte the variable it occupies also known as its base address.

Declaration Statement

A simple means of creating a variable is a declaration statement. The syntax of a declaration statement is

data_type variableName [= expression];

For example

```
double pi = 3.14159265358979323846264338327795;
```

When the compiler encounters a variable declaration, it creates the variable being declared. It does the following:

- allocates memory for a value of the specified type
- associates the variable name with the address of that memory space.
- initializes that memory space with the value provided in the declaration if any.

In the above example the compiler will create a location in memory that can store double precision floating point values and assigns it the name pi. It stores the value 3.14159265358979323846264338327795 in that location. In programming we say that pi has been assigned the value 3.14159265358979323846264338327795 or pi has been initialised with the value 3.14159265358979323846264338327795. You don't have to assign a value to an object/variable during declaration. The above declaration would still have worked without any error as double pi;

Note that the value a variable acquires if it is not initialised in C and C++ is whatever the most recent tenant left there.

Passing Parameter by Value

When you call a function in C or C++, a variable is created for each of the formal parameters and each initialised with the value of the corresponding actual parameter from the caller using the assignment (=) operation. In the code of Listing 1, when square () is called, a variable is created and identified by num and assigned the value of x; The statement executed when Square is called is int num = x;

Formal parameters count as local variables with the only difference between parameters and local variables being that parameters start out with a value copied from the caller (argument or actual parameter) while local variables start with random initial values.

```
int square (int num)
{
        int result;
        result = num * num;
        return result;
}
int x = 3;
square(x);
int y = x;
```

Listing 1:

The value of y after the code in Listing 1 has executed is 3. The value of x is still 3 outside square (). The value it was working with is not the actual parameter it was passed but its own copy.

The memory for the locals continues to be allocated so long as the thread of control is within the owning function. Locals continue to exist even if the function temporarily passes off the thread of control by calling another function. The locals exist undisturbed through all of this. When the function finishes and exits, its locals are deallocated automatically.

Using the New Operator or the Malloc () Library Function

These two create variables from ***heap*** - the free memory area that lies between a program (and its permanent storage area) the stack. Heap memory, also known as "dynamic" memory, is an alternative to local stack memory. The two methods we have discussed so far make use of local stack memory. Local memory is automatic - it is allocated automatically on function call and it is deallocated automatically when a function exits. Figure 3 shows conceptually how a C++ program appears in memory.

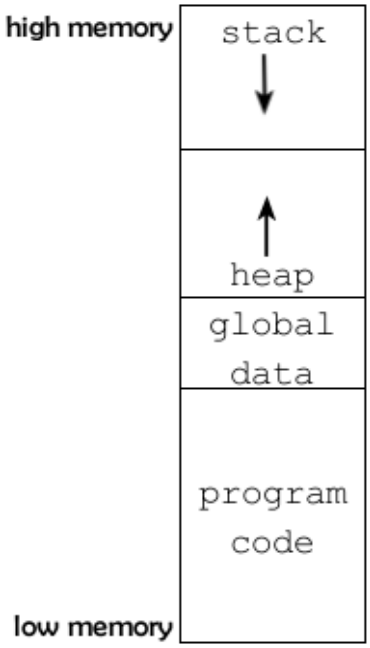


Figure 3

Heap memory is different in every way. The programmer explicitly requests the allocation of a memory "block" of a particular size, and the block continues to be allocated until the programmer explicitly requests that it be deallocated. Nothing happens automatically. This way, the programmer has much greater control of memory, but with greater responsibility since the memory must now be actively managed.

When you use either of these methods, you are making a request for a block of memory large enough to store a value of the specified type. As we see later, with malloc (), you make this request explicitly. What you get is a pointer to the base of the block of memory allocated. They return the starting/base address of the block if the request is successful. If you let this pointer, get lost, you end up with what is known as a memory leak - an object is stored in memory but cannot be accessed by the running code. This may or may not be a serious problem. The result

will be that the heap gradually fills up as there continue to be allocation requests, but no deallocation requests to return blocks for re-use. So, to use the variable, you must keep this pointer. Where? In another variable, a pointer (variable). The syntaxes of the two are

- new data_type[size];
- malloc (bytes);

Of the two, new is easier to use. It simply returns the base pointer of the allocated block. This pointer is a pointer to type specified in the request. If the object/variable to create is one, you could specify 1 for size or omit the subscript operator all together.

For malloc, it allocates a region of memory large enough to hold an object whose size is the number of bytes passed when it is called. To specify this number, typically the sizeof operator is used as

```
sizeof(data_type) * number of elements
```

For example, to allocate space to store an array of 23 integer variables, we make the call

```
malloc (sizeof(int) * 23)
```

The pointer malloc () returns, like the new operator, is a pointer to the first element of the region, but this pointer is guaranteed to be properly aligned for any data type. The caller may then use a cast operator to convert this pointer to another pointer type. The prototype for malloc () is

```
#include <stdlib.h>
void *malloc (size_t size);
```

Once a program is done using the allocated space on the heap, it should deallocate the heap block. To do so needs a pointer to the block. The two facilities used are the free () library function used to deallocate space previously allocated by malloc () library function and the delete operator used to free space previously allocated by the new operator. Once freed, this memory can be reallocated to different purposes by a subsequent allocation request. The prototypes for free is

```
#include <stdlib.h>
void free (void *ptr);
```

The free () function takes a pointer to a heap block and returns the bock to the free pool for later reuse. The pointer passed to free () must be exactly the pointer returned earlier by malloc (), not just a pointer to somewhere in the block.

The general form for delete is

```
delete pointer_var;
delete [] pointer_var; //frees a dynamically allocated array
```

The pointer passed to delete must be exactly the pointer returned earlier by new, not just a pointer to somewhere in the block. Both free () and delete turn their pointer argument/operand into what is known as a dangling pointer - a pointer that is pointing to a location that has been

returned back to the pool of reusable blocks (the variable it pointed to has been deallocated). The programmer must remember not to use the pointer after. A possible consequence is a crash of the program especially if you write to the target of the pointer.

REFERENCES

A reference is the name that is given to an existing variable or what you get back when you dereference a pointer. When you declare/create a reference, you must bind it to an existing variable. References are also known as aliases, perhaps because they are created using names of existing variables, but a reference can as well be bound to a unanimous variable. You create and bind a reference to variable using declaration statement that contains assignment operation (implicitly or explicitly) in which the left-hand side is the reference being created and the right-hand side is a lvalue such as a variable name.

A *lvalue* is an expression that refers (or returns a reference) to a region of storage that can be examined and stored into. Such a region is known as an *object*. For example, a variable or an array element are lvalues but the constant 42 and the expression i+1 is not. A lvalues is a value that has an address, meaning it is a variable or a dereferenced reference to a certain memory location.

The code below illustrates this:

```
double pi;
double &piRef = pi; //the & operator means piRef is a reference
piRef = 3.14159265358979323846264338327795;
```

After the above code has been executed, the value of pi is 3.14159265358979323846264338327795. The two names/identifiers pi and piRef are names of the same variable. In other words, when you create a reference, no variable is created.

The in Listing 2 is a version of the code in Listing 1. The only change is the character '&' preceding the formal parameter num. This turns num into a reference (local) variable.

```
int Square (int &num)
{
    int result;
    result = num * num;
    return result;
}

int x = 3;
Square(x);
int y = x;
```

Listing 2:

The value of y after the above code has executed is 9. The formal parameter num is now a reference variable and therefore in the above code refers to the same variable as x, the actual parameter in the call. Therefore, any changes made to num in square are really just changes to the x. This code illustrates what is known as *passing by reference*. When a variable is passed by reference, the changes made to the formal parameter are reflected at the caller. This is in contrast to *passing by value* in which the called function creates a copy of the actual parameter for itself before going to work (see section 2.2).

SO, WHAT ARE POINTERS FOR?

Pointers solve two common software problems. First, pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better. Second, pointers enable complex "linked" data structures like linked lists and binary trees.

Share Information

Pointers enable different sections of code (e.g., functions) to share information or to communicate. This is implemented by simulating *passing by reference*. The term *passing by pointers* has been used to refer to this simulation. To illustrate this, let us look at the classic example swap () function (used to illustrate reference parameters) which exchanges the values of two ints. We first show the version that accomplishes this using references.

References Version:

Below is version of this code that accomplishes the job using passing by reference.

```
void swap (int& a, int& b) {// The & declares pass by reference
{
    int temp;
    temp = a; // No *'s required -- the compiler takes care of it
    a = b;
    b = temp;
}

void SwapCaller ()
{
    int x = 1;
    int y = 2;
    Swap (x, y); // No &'s required -- the compiler takes care of it
}
```

Listing 3:

Pointers Version (Shallow copying):

Before looking at swap (), here is a quick information on retrieving and creating pointers. It should be clear now that every variable has an address (or a pointer to it). Here are some facts one must know about pointers:

- To obtain a pointer to a variable, you precede its reference/name with the ***address of***

operator, '&'.

- To obtain a reference to a variable if you know its address/pointer, precede the address/pointer with the ***dereference*** operator, '*'.
- To declare a pointer variable, use the same form as described in section 2.1 but precede its identifier with the '*' operator.

Back to the swap () function, listing 4 is how to implement it using pointers.

```
void Swap (int* a, int* b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
void SwapCaller ()
{
    int x = 1;
    int y = 2;
    Swap (&x, &y); // Use & to pass pointers to the int values
}
```

Listing 4: To Call Swap (), The Caller Must Pass Pointers to The Variables of Interest:

When Swap () is called, it is passed two parameters by value i.e., two variables are created identified by the formal parameters a and b and initialized by the value of the actual parameters. The values passed are pointers. So, after the call, the value of a is the same as the value of &x. Two pointers which both refer to a single variable are said to be "sharing". In this case a and &x are an example.

Remember that in this case, it is the address of x that is copied to the variable identified as a in Swap. It is not the object x that was copied but the pointer to x. Both the caller and Swap can access the value of interest x in the caller, but the value of interest itself, x is not copied. What is copied is its address. This communication is called "shallow" since instead of making and sending a (large) copy of the value of interest, a (small) pointer is sent and the value of interest is shared.

The recipient needs to understand that they have a shallow copy, so they know not to change or delete it since it is shared. The alternative where a complete copy is made and sent is known as a "deep" copy. Deep copies are simpler in a way, since each function can change their copy without interfering with the other copy, but deep copies run slower because of all the copying.

References Vs Pointers:

These two versions clearly show that the pointer version is untidier syntactically. When using references, the types of the various variables and parameters operate simply as they are declared (int in this case). The complicating layer of pointers required to implement the reference parameters is hidden. The reference parameter is translated into a pointer variable

and the corresponding argument into the address of the argument. The pointer parameter is then automatically dereferenced whenever it is used in the function, producing aliases (i.e., references) for the argument. The compiler takes care of all these without allowing the complication to disturb the types in the source code. Generally, whenever a reference can work, it is better to use them.

Linked Data Structures

Pointers enable complex "linked" data structures like linked lists and (binary) trees. This can only be done by pointers. You cannot create a liked structure using references because a reference is implemented as some kind of a constant pointer; you must initialize it at declaration point and you cannot change it (the object it points to) after this. If you were to use references to create a linked list for instance, the list can never grow (or shrink) in size. In a linked structure, an element (a variable or object) contains at least one link (a pointer) to another element. For a linked list, this link is a pointer to the next element in the chain of elements. See the figure below.

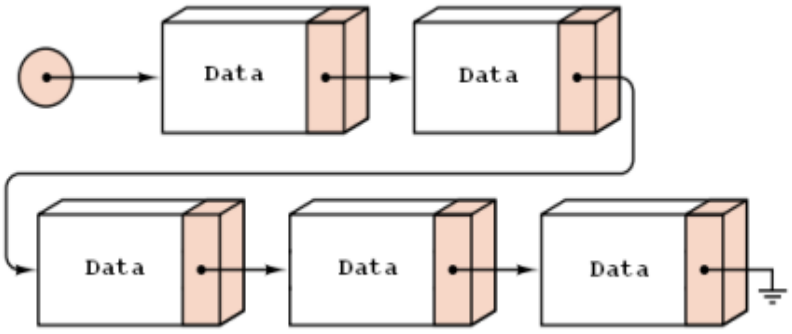


Figure 3: A linked list

ADOPTING OBJECTS

The elements of a linked structure don't have to be in any order in memory. The term element, node and member can be used to mean the same thing in this context. The most (probably the only) important thing is that they are all reachable/accessible (by passing from one node to another) because each node stores the address(es) of some other node(s) in the linked structure. Such a sequence is the most frequently used in implementation of a linked list which - a data structure composed of nodes, each is holding some information and a pointer to another node in the list. If a node has a link only to its successor in this sequence, the list is called a singly linked list.

An inherent problem with all linked structures is that they forbid direct access to their elements. This makes them slow data structures for algorithms that require random access to elements. To access an element of a linked structure, you must pass through other elements if the target element is not the first. A worst case is where the target is the last element in the search path, in which you visit every node in the path.

This does not have be though. Using pointers, you can access objects of the same type in any desired way, even randomly. You can view objects of same type as if they are members of structure diverse structures at the same time!

We will illustrate this using the simplest data structures; arrays and singly linked lists. Arrays are suitable data structure for algorithms that require random access to objects because they allow direct their members. The main problem of arrays arises from the property that members (the elements) of an array must be adjacent (the same distant from each other) in memory. This means that that to insert an element in an array, in any position other than the last, requires (some of) the existing elements to move. In the worst case, you are inserting an element in the first slot of the array, in which case all the other elements must move. This weakness is what a linked list addresses but as we have shown, introducing its own drawback.

Suppose you had a linked list whose nodes are each a structure with the following fields

- Name
- Account Number
- Account balance

And for some reason you wanted to see the list sorted by name or by account number or account balance. You could use an algorithm that the linked list as and sorts it! we use the exclamation mark to terminate the previous sentence because given the way a linked list elements are accessed and the number of elements movements expected even in a nearly sorted list, it is ridiculous to attempt sorted a linked list. An array would perform much better so you could shift or copy the data to an array before sorting but still you will have the problem of first moving data each time you wanted to sort. A solution that involves no movement of data both initially and during sorting would be the best. This we illustrate in the example that follows.

Example: Accessing Linked List Elements Randomly

This can be achieved by an array adopting the elements of the linked list. How? You create an array of pointers to the type of elements of the linked list. Then you set each element of the array to point to an element of the linked list.

Consider the linked list in Figure 4. Each node contains a serial number, content and a link to the next element. The diagonal line in the pointer part of the last node represents a null link and indicates that this list element has no successor.

To access the last element of the list i.e., the element containing 3333, you must visit the first six elements. If this structure had been an array, then only one operation would be necessary because we would know how far this element is from the first or in other words, we know its address. In an array once you know the address of what is equivalent to the *head* in a linked list, you can compute the address of any element because the elements are adjacent to each other and every element has the same size in bytes as the other.

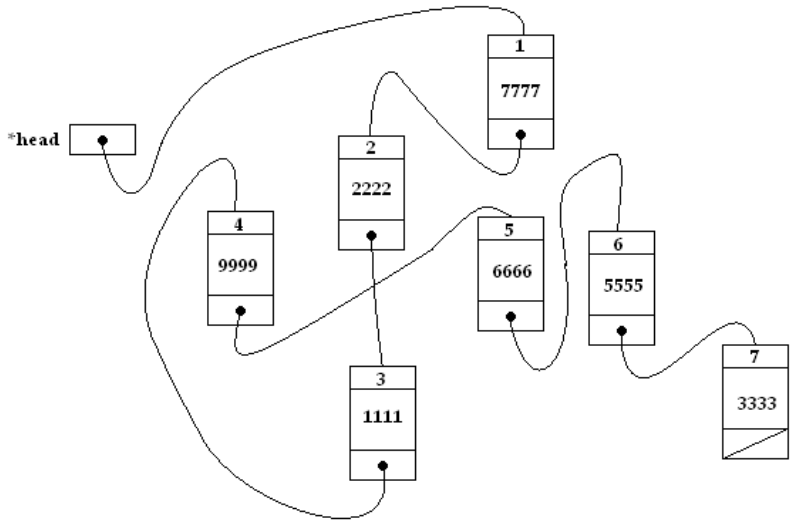


Figure 4

The diagram in Figure 5 shows an array that has adopted the elements of the above linked list.

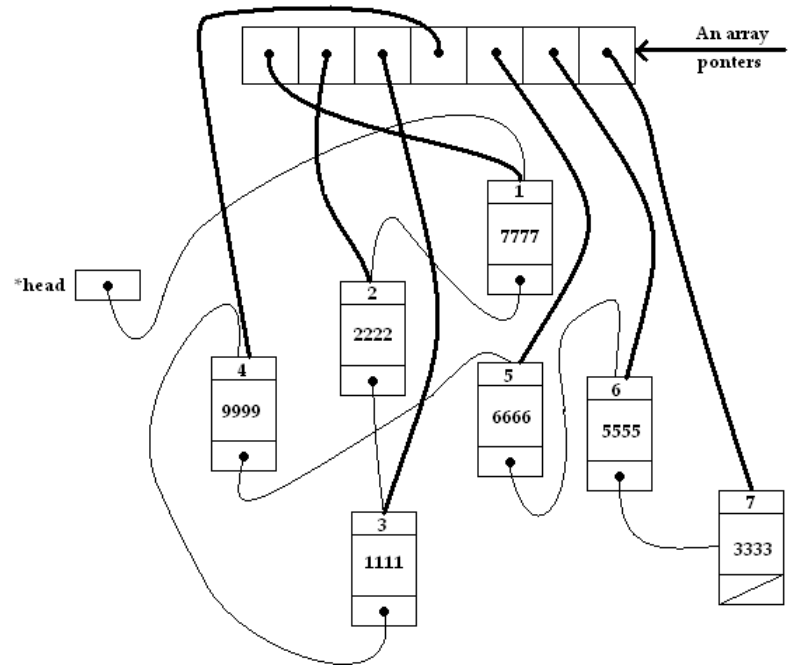


Figure 5

If the array name were nodes, then to access the last element would require just specifying the array name and 7 as a subscript to obtain the pointer to this node. What do you do to the pointer to access the node? In C/C++ you precede this pointer with the indirection operator. For example, to display the content of seventh element on the screen, you could write

std: cout << *nodes [7]. content;

assuming that a node is a structure with the field content storing the content/data/information of the node. To do the same via the linked list would be a frightening.

The two structures can exist coexist without either being aware of this fact. For example, the array can sort the nodes without disturbing the nodes. You make the i^{th} array member point to the i^{th} largest element. You disturb neither the linked list nor the objects. The diagram in Figure 6 shows the array having not only adopted the nodes but also sorted them.

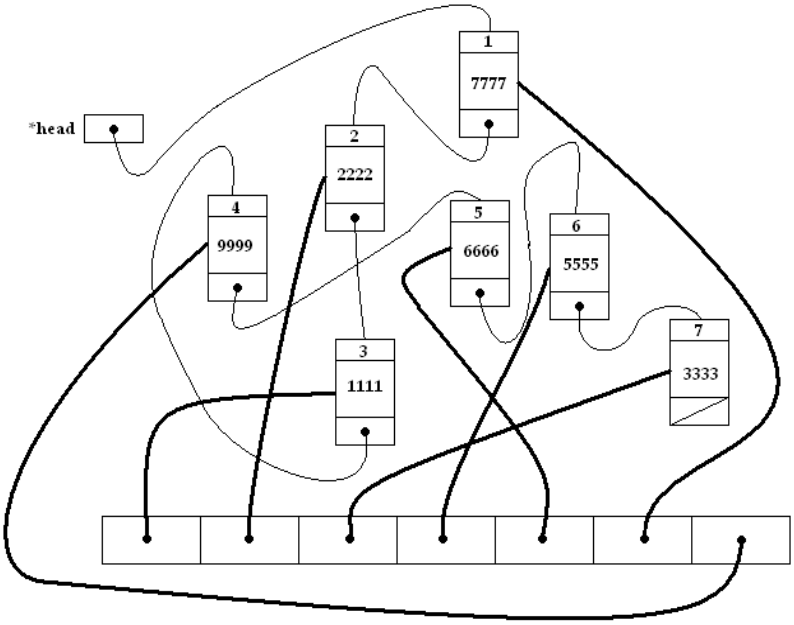


Figure 6

In the most flexible implementation, objects should not be part of a data structure. The data structure should instead only adopt them – elements of the structure are not containers of any useful information to the user but instead a pointer to an object of interest. This way, the linked list in the example above should adopt the nodes rather than the nodes being part of its fabric. This is done by having the nodes of the linked list being pointers to the type of objects it needs to adopt. The diagram of a linked list would be too murky to draw given that both the objects to be adopted and the nodes of the linked list can be anywhere in memory.

Using the adoption approach, the structure is not tightly bound to the objects it claims to arrange or organize. As illustrated above, a consequence of this is that the same objects can be adopted by (viewed as being elements of) multiple data structures at the same time. Another desirable consequence is that objects will survive (even without being aware) the demise of the structure that seems to own them and vice versa. The only problem with this is that a structure's nodes will contain wild pointers if another structure that also owned the objects decides to return some or all the objects to the free pool.

CONCLUSION

We have shown in this paper that implementing a data structure should not be with the elements as part of the fabric of the structure. This way a data structure can share or hand over its elements to another structure in the same program. Many useful applications need this ability. For instance, in the same application, you might require to present data sorted lexically on some key and numerically on another key. If you implement this as an actual sort, your application will be slow because actual sorts haul objects in memory. Using adoption where shadow copying takes place will be a huge improve but having multiple concurrent structures is even faster. This as we have illustrated can only be done with pointers.

References

[1] Samuel P. Harbison III, Guy L. Steele Jr. *C, A Reference Manual, 5th Edition*. (C) 2002 by Prentice Hall.

[2] Herbert Schildt. *C++ from the Ground Up, Third Edition*. McGraw-Hill/Osborne, 2600 Tenth Street, Berkeley, California 94710, U.S.A.

[3] Nick Parlante. *Pointers and Memory*. 2000