medium.com /@hazemu/tricky-pointer-basics-explained-ba87656c9a9a

Tricky Pointer Basics Explained

hamid

25-31 minutes

Working with pointers can be confusing if one does not possess a solid understanding of what they are and how they work. In this post, I present an attempt to explain some of the intricate details of dealing with pointers in simple terms. This article does not require any special background, only preliminary knowledge of elementary C/C++ concepts like variables, pointers, and functions.

In essence, pointers are just another type of variables. The only difference of course is that they are intended for pointing to other variables. But before we go any further, let's try to imagine how variables are represented in memory.

We can imagine computer memory as a set of slots, each containing an address where some data can be stored.

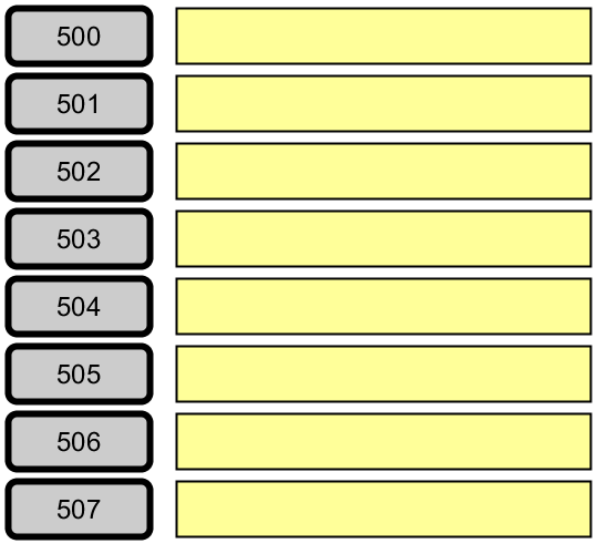

Fig.1 An imagination of how computer memory looks like

The grey rounded rectangles in Fig. 1 carry the addresses of the different memory slots, while the yellow rectangles represent the actual slots where data is stored.

When you declare and initialize a simple variable, e.g.

```
int x = 10;
```

what happens is that the compiler translates this code into machine code that:

1. reserves a memory slot
2. records that you want to refer to this slot as '*x*'
3. stores the value 10 into it

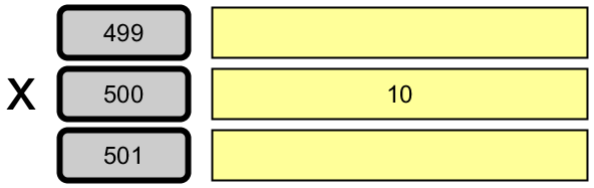

Fig.2 Computer memory after running code that declares an integer x and initializes it to 10.

From this point onward in your program, whenever you reference the variable *x*, the compiler knows you're referring to the value in slot #500. You can read this value, increment it, decrement it, add another value to it … etc. Alright, this doesn't sound difficult. But why does it sound all confusing when we start using pointers? Well, let's see.

When you declare and initialize a pointer, e.g.

```
int* p = NULL;
```

the compiler generates code that:

1. reserves a memory slot
2. records that you want to refer to this slot as '*p*'
3. stores the value NULL into it

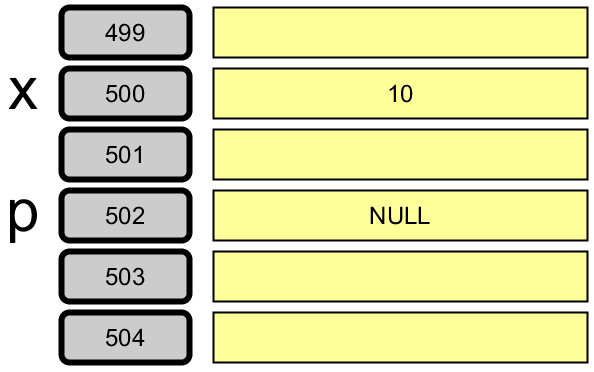

Fig.3 Computer memory after running code that declares a pointer to integer p and initializes it to NULL.

The result is illustrated in Fig. 3. So far, pointers do not look any different. Let's have our pointer *p* point to something, like the integer *x* we introduced earlier.

How do we do that? Well, it's pretty simple:

```
p = &x;//
// Doing it this way would cause a compilation error
// p = x;
```

This line of code instructs the computer to take the address of the slot/variable called *x* and place it into the other slot/variable called *p*. This means storing the value 500 in slot #502.

Attempting to assign *x* to *p* directly — as shown in the commented code — would cause a compilation error because *p* and *x* have different types. *p* is an *int*;* a pointer to a slot capable of holding other integers' addresses. It is illegal to store anything in *p* that is not an address of an integer slot, and the compiler knows that *x* is an integer, which explains why it deemed the assignment of *x* to *p* invalid. It simply says: you cannot store integers in a place designated for storing addresses of integer slots.

This explains why we have to use the address-of operator &, to tell the compiler that we want to store the "*address*" of slot *x*, i.e. 500, not *x*'s contents, i.e. 10, into our newly created slot *p*.

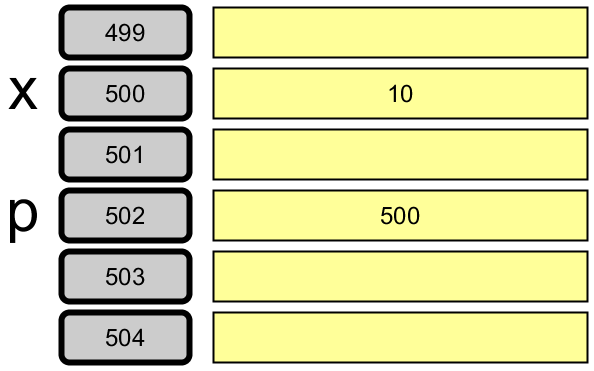

Fig.4 Computer memory after assigning p to point to the address of x.

The only difference we have discovered between pointer and non-pointer slots/variables so far is that we use the former to store data and the latter to store addresses of other slots/variables.

This difference also explains why we have special syntax to change the contents of the slot/variable a pointer is pointing to, e.g.

```
p = NULL;   // p is pointing to NULLp = &x;     // p is now pointing to x
*p = 20;    // p is still pointing to x, but x is now equal to 20int y = 0;
p = &y;     // p is now pointing to y
*p = -1;    // p is still pointing to y, but y is now equal to
-1printf("%d", x);  // prints 20
```

Passing variables around

Let's see what happens when we pass variables to other functions. Let's say we write the following code:

```
void process(int n)
{
    // do something with n
}int main()
{
    int x = 10;
    process(x);
    printf("%d", x);    // prints 10    return 0;
}
```

When *process()* is called, the code copies the contents of the slot/variable *x* into a new slot *n* that *process()* uses.

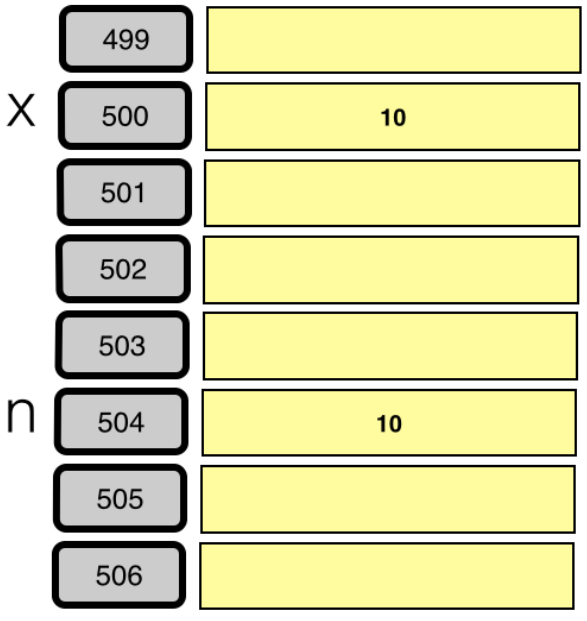

Fig.5 Computer memory after invoking function process()

This is commonly referred to as passing variables by value. And because variables are copied, assigning any value to *n* within *process()* has no effect on *x* in *main(); n* and *x* are simply two different slots/variables.

But what happens if we pass a pointer to a function?

Well, let's change our function *process()* to have it expect a pointer:

```
void process(int* j)
{
    // do something with j
}int main()
{
    int x = 10;
    int* p = &x;    process(p);    return 0;
}
```

Now, *process()* expects a pointer, *main()* declares a pointer *p* that points to *x,* and calls *process()* with *p*.

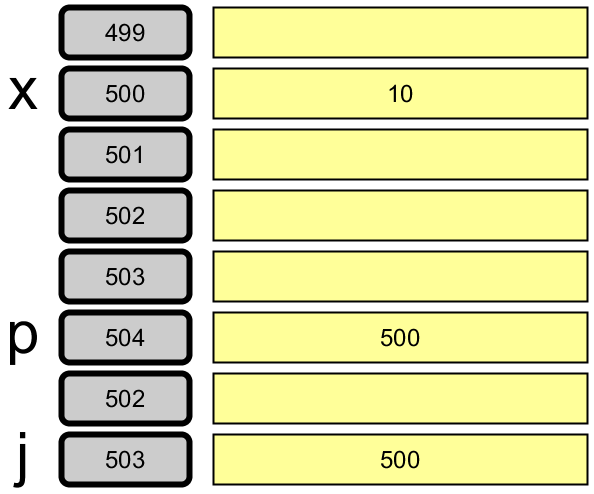

Fig.6 Computer memory upon invoking function process()

What happens when we invoke *process()* with a pointer turns out to be exactly the same as what happened with non-pointer variables — the contents of slot *p* are copied into a new slot *j* that is then passed to *process()*.

Intuitively, modifying the contents of slot *j* has no effect on slot *p* or slot *x*. Modifying the contents of a pointer slot means having it point to a different variable.

This explains why the code within *process()* below has no effect on *p* or *x* in *main()*:

```
void process(int* j)
{
    int y = 1000;
    j = &y;
}
```

In the snippet above, we introduced a new slot/variable *y* and had *j* point to it. This changes the contents of *j* to the address of *y* but it does not change *p* or *x*. What if we change the contents of the slot *j* is pointing to (#500) though?

```
void process(int* j)
{
    *j = 600;
}int main()
{
    int x = 10;
    int* p = &x;    process(p);
    printf("%d", x);    // prints 600    return 0;
}
```

This code changes the contents of the slot *j* is pointing to, i.e. the value at address 500, also known as our integer variable *x*. This explains why *printf()* prints 600 after *process()* is called even though *x* was originally initialized to 10.

Think of it this way:

- Pointer slots are similar to non-pointer slots in the sense that both are copied when they're passed to functions.
- Changing a copy of a slot has no effect on the original slot.
- You can use a pointer slot to modify the contents of another slot.

Changing pointers

So, in order for a function to be able to change the value of a variable/argument/slot, we need to pass that slot's address (pointer) to it. In fact, this is one of the most primary uses of pointers. But what if we're interested in having our pointer point to some other slot/variable instead?

```
void process(int* j)
{
   //
   // we are not modifying *j, we are modifying j itself
   //
   j = NULL;
}int main()
{
    int x = 10;
    int* p = &x;    process(p);
    if (p == NULL)
    {
        printf("p has been modified!\n");
    }
    else
    {
        printf("Tough luck!\n");       // this will be printed
    }    return 0;
}
```

In the code snippet above, we're attempting to change slot *j* by having it point to another address (NULL). But since *j* is a copy of *p*, slot *p* itself will not be affected. This is illustrated in the figure below.

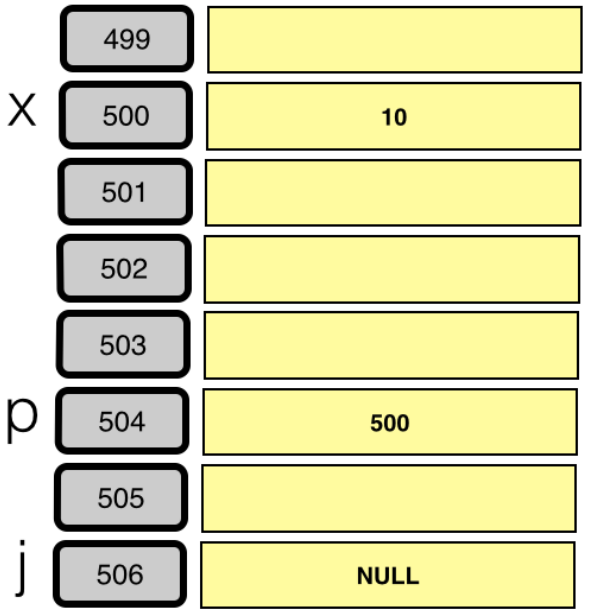

Fig.7 Computer memory after setting j to NULL. j initially had the same value as p, i.e. 500.

But what if *process()* wants to change *p* to have it point to something else? Well, we can employ the bit of wisdom we just learned: whenever you want a function to modify a slot, don't pass a copy of it, pass a

pointer to it! And pointers themselves are no exception to this rule.

To be able to modify *p* within *process()*, we need to pass the address of *p*, i.e. 504, to it. Only then can *process()* use this address to modify the contents of *p* to have it point elsewhere.

Think with me: what should the type of *j* be in this case? Well, *j* is definitely a pointer, and it's a pointer that is capable of pointing to stuff like *p*. What is *p*? *p* is a pointer to an integer. This means that *j* has to be a pointer to a pointer to an integer.

```
void process(int** j)
{
    *j = NULL;
}int main()
{
    int x = 10;
    int* p = &x;
    int** pp = &p;    process(pp);
    if (p == NULL)
    {
        printf("p has been modified!\n");   // this will be printed
    }
    else
    {
        printf("Tough luck!\n");
    }
}
```

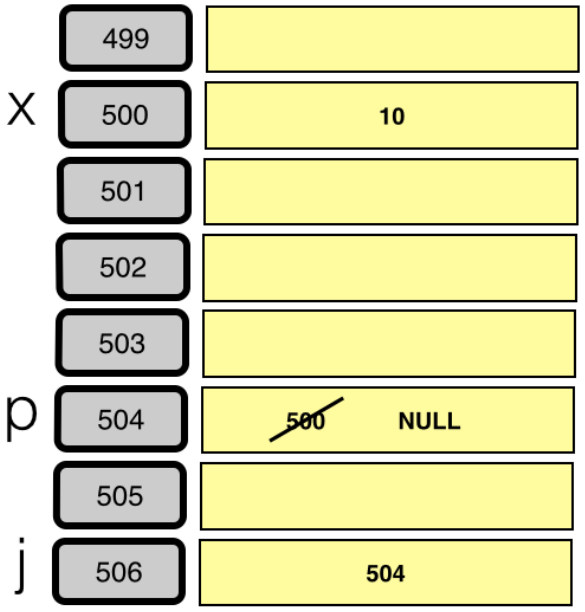

Fig.8 Computer memory after setting p to point to NULL via j.

Now, we can easily use the address of *p* stored in *j* to modify *p*'s contents and have it point to NULL instead.

Another easy way to achieve the same effect is to have *process()* return the new address and assign it directly to *p*.

```
int* process()
{
    return NULL;
}int main()
{
    int x = 10;
    int* p = &x;    p = process();
    if (p == NULL)
    {
        printf("p has been modified!\n");   // this will be printed
    }
    else
```

```
    {
        printf("Tough luck!\n");
    }
}
```

In fact, this is exactly how operator *new* works in C++; it is just a function that allocates a block of memory on the Operating System's heap/free-store and returns its address.

```
int main()
{
    int* p = NULL;
    p = new int;      // if memory allocation succeeds, p will
                      // point to newly-allocated int, not NULL    if (p != 
NULL)
    {
        printf("p changed successfully!");  // this is printed
    }
    else
    {
        printf("p is still pointing to NULL!");
    }
}
```

Tricky cases

Based on our newfound understanding, we should be able to start looking into some of the trickier cases of using pointers. As an example, let's check out the following code snippet:

Example 1

```
vector<int*> v;//
// some code that adds some pointers to vector v
//for(int i = 0; i < v.size(); ++i)
{
    int* p = v[i];    if (p == NULL)
    {
        p = new int;
       *p = 0;
    }
}
```

In this example, the code is trying to:

1. go through all the pointers-to-integers in *v*,
2. make sure all of them are initialized, i.e. not pointing to NULL,
3. only initialize those pointers pointing to NULL, and store the value zero in their newly allocated integers.

However, there's a subtle bug in this code: it won't actually initialize the pointers pointing to NULL in *v.* You can try it out yourself in a debugger. You might be wondering why. Aren't we calling *new* and assigning its return value to *p*? The answer is yes, but this initalizes *p*, a copy of *v[i],* as opposed to initializing *v[i]* itself. Let's try to visualize what happened.

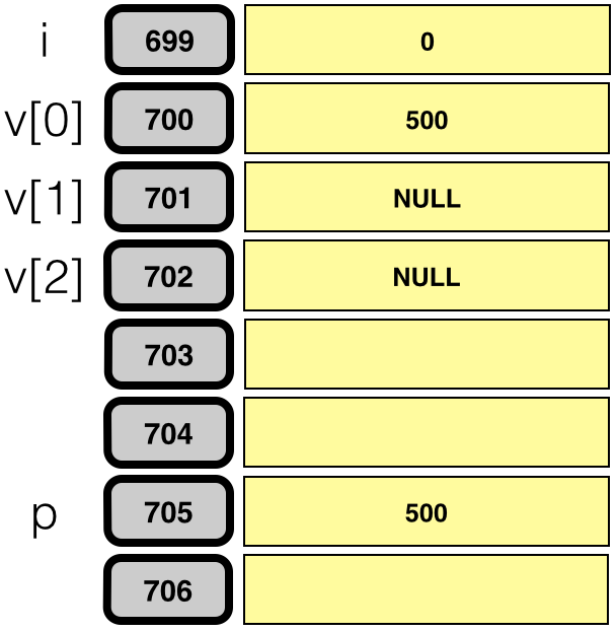

Fig.9 Computer memory after setting p to v[0], i.e. in the first iteration of the for loop where i = 0.

In Fig. 9, we assume that *v* contains three pointers: *v[0]* points to slot #500, while *v[1]* and *v[2]* both point to NULL.

We also assume we're about to execute the first iteration of the for loop (*i = 0*):

```
int* p = v[i];
```

The code checks if *p* is NULL, which is false, and moves on to check *v[1].*

```
if (p == NULL)  // p is pointing to slot #500,
                // i.e. condition is false
{
```

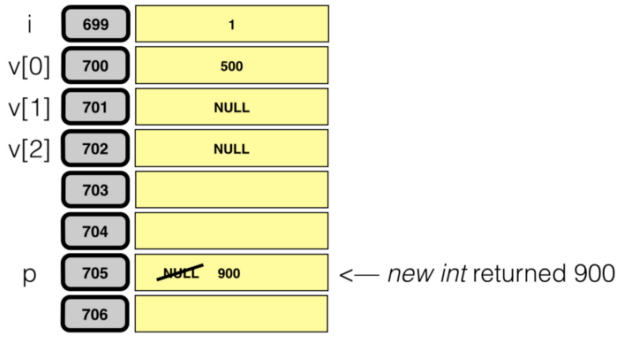

Fig.10 Computer memory after setting p to v[1], i.e. in the second iteration of the for loop where i = 1.

Now *p* is pointing to the same thing *v[1]* is pointing to, i.e. NULL. Checking if *p* equals NULL evaluates to true, and the code within the if block initializes *p* to the memory address allocated by operator *new*, 900 in this example.

Now, it becomes obvious that we have changed *p* but not *v[1],* and the same exact thing happens with *v[2]*.

Interlude

It is easy to see a discrepancy in the way we think about pointer (vs. non-pointer) variables if we change *v* in our running example to make it a vector-of-integers, and perhaps assume — to complete the analogy — that we need to assign the value zero to vector entries equal to -1. The modified code snippet then becomes:

```
vector<int> v;//
// some code that adds some integers to vector v
//for(int i = 0; i < v.size(); ++i)
{
    int x = v[i];    if (x == -1)
    {
        x = 0;      // it's clearer here we're not changing v[i]
    }
}
```

It is arguably much easier to observe that changing *x* has no effect on the values in *v* in this version of the example. Contrast this with the subtlety of the same issue when we were dealing with pointers. This observation seems to suggest we tend to reason a little differently about pointers. Not that it makes dealing with pointers less tricky, but this realization may make us slightly better at avoiding some of their common pitfalls.

Example 1 — fix v.1: Change the original pointers, not their copies

```
vector<int*> v;//
// some code that adds some pointers to vector v
//for(int i = 0; i < v.size(); ++i)
{
    if (v[i] == NULL)
    {
       v[i] = new int;
       *(v[i]) = 0;
    }
}
```

One possible fix for our code snippet could be to avoid creating copies of pointers altogether and use *v[i]* directly. This way we know for sure that the values we assign will make it to the original pointers directly, not copies thereof.

Example 1 — fix v.2: Use a pointer-to-pointer

```
vector<int*> v;//
// some code that adds some pointers to vector v
//for(int i = 0; i < v.size(); ++i)
{
    int** pp = &v[i];    if (*pp == NULL)
    {
       *pp = new int;
       **pp = 0;
    }
}
```

This is another fix that employs the common wisdom: if you need to change a variable/slot, don't attempt to do so via a copy of it but use a pointer to it instead. Again, it is better to visualize this in order to understand it.

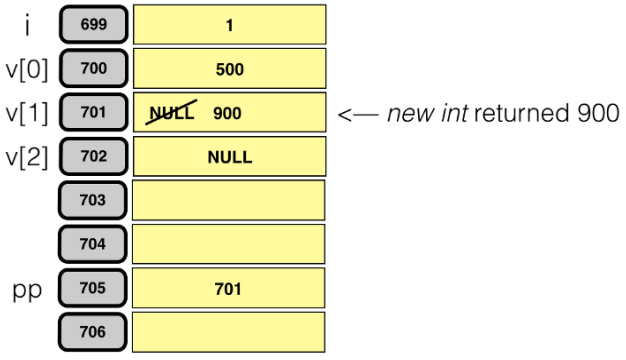

Fig.11 Computer memory after setting p to v[1], i.e. in the second iteration of the for loop when i = 1.

Fig. 11 shows the state of memory during the execution of the second iteration of the for loop (i = 1):

1. We create a pointer to int-pointers pp and have it point to the address of *v[1]*, i.e. 701.

2. The code checks if **pp* contains NULL, which evaluates to true (slot #701 contains NULL).
3. The code proceeds to call new int and store the result in **pp*, i.e. in slot #701.
4. The code finally sets ***pp*, i.e. contents-of(contents-of(*pp*)) = contents-of(contents-of(701)) = contents-of(900), to zero. This last step isn't illustrated in the figure.

Example 2

Trees are one of the most widely used data structures. They are utilized in a wide variety of applications, e.g. sorting, searching sets of elements, finding min/max among a set of elements and much more.

One of the most useful data structures that are based on trees is Binary Search Trees. You can think of a binary search tree as a sorted array. Sorted arrays are especially useful because we can perform binary search on them. However, inserting an element into a sorted array might be a little inefficient sometimes because we may need to shift some elements to make room for newly inserted ones. Binary search trees represented with nodes and pointers do not suffer from this particular inefficiency because they allow us to insert elements without having to do as much work.

A node in a binary search tree is often represented as follows:

```
struct tree_node
{
    int data;
    tree_node* right;
    tree_node* left;
};
```

A few variants are tree node structures that:

1. allow data of various types to be stored in a node, e.g. via void pointers or template-types (not just *int*),
2. feature a parent pointer to facilitate some tree operations,
3. feature fields to maintain the size of a sub-tree below any given node; such fields could be very useful in finding the order of any given node in the tree quickly

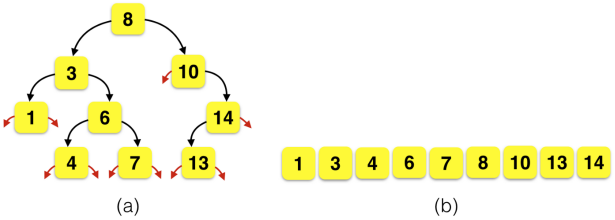

Fig.12 (a) An example binary search tree. Red arrows indicate left/right pointers not pointing to any nodes, i.e. pointing to NULL. (b) A sorted array containing the same set of elements. You can see that the order of the elements in the array is exactly the in-order traversal of the elements of the binary search tree.

Let's see how we can insert a new element into a binary search tree. Assuming we would like to insert the value 9 into the tree shown in Fig. 12, we first conduct a search for it. We either find it and return without actually inserting any new elements into the tree (since duplicates aren't generally welcome in binary search trees), or we wind up with a null-pointer indicating that the value we are inserting does not exist in the tree.

We can see an example of this operation as we attempt to insert 9 into the tree in Fig. 12 (a).

1. We start at the root node (data = 8)
2. We check if the current node contains the data we're looking for, i.e. 9
3. Failing to find 9, we compare 9 with 8 to decide which way to proceed with our search (left or right).
4. Since 9 is greater than 8, we move on to the right node (data = 10)
5. Failing to find 9, we compare 9 with 10 and decide we will move left.
6. We end up with a red-arrow, i.e. a pointer pointing to NULL which is the left arrow of the node whose data is equal to 10.

This procedure can be written recursively as follows:

or iteratively as follows:

One nice thing about this algorithm is that whenever we fail to find any given value, we end up landing at the pointer that should have been pointing to that value had it been in the tree. This means we only need to make a simple change to the *binary_search()* function above to have it insert any given value in

the tree if it fails to find it. Let's trace the code of the iterative version of the *binary_search()* function above assuming we would like to insert the value 9 into the binary search tree in Fig.12 (a).

You can see that the code will end up exiting the *while* loop, specifically when *x* is equal to the left pointer of the node whose data is equal to 10.

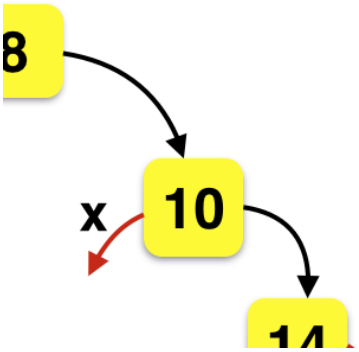

Fig.13 Inserting the value 9 — *x* is now equal to the left pointer of node whose data is 10.

OK. This looks easy. All we need to do now that we know 9 does not exist is:

1. create a new node,
2. give it the value 9,
3. add this new node to the tree by having *x* point to it.

So, we simply add the following line of code after the *while* loop:

```
x = new node();
x->data = value;    // this value is 9 in our example
x->left = x->right = NULL;
```

and we have our modified *binary_search()* function — let's call it *insert()* — return the value of the newly created node instead by modifying the return statement to be:

```
return x;
```

Everything looks fine; you call *insert()* with our example tree and the value 9, the code exits the *while* loop as expected, creates the new node, gives it the value 9, sets its left and right pointers to NULL, and returns the value of *x* , which happens to be the address of the newly allocated node . The only problem is that if you call *binary_search()* with the value 9 on your tree afterwards, it will still return NULL.

The root cause of this weird behavior is that the code snippet we've added does not attach the newly created node to the tree at all; we had *x* point to our newly allocated node thinking that would attach it to the left pointer of the node with value 10. But *x* is just a copy of that left pointer and having it point to the new node — or anywhere else for that matter — does not affect the original pointer.

What do we do to solve this problem? Just as usual, we can:

1. **use a pointer-to-pointer**
   This means we will have to change our variable *x* to be a pointer-to-pointer instead and change all of its references accordingly, i.e.

   Fig.14 Inserting an element into a binary search tree.

2. **avoid using copies of the pointer we want to change**
We use *x* throughout our *insert()* function to iterate over the nodes we're searching, i.e. we keep copying the address of every node we visit to it. We know the search is over if *x* becomes NULL. What if, instead of using *x*, we use a pointer to the node with value 10 to modify this node's left pointer?

   Fig. 15 Inserting an element into a binary search tree.

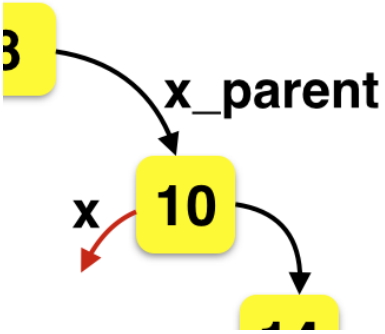

Fig.16 Inserting an element into a binary search tree. After the search terminates, x_parent will point to the last visited node. To learn whether we should insert the new value to its left or to its right pointer, we modify the code as in Fig.14

If you take a closer look at this solution, you will find out that it is more or less a variation of the first solution, i.e. we're using a pointer to pointer. Only this time we're using *x_parent* as the pointer-to-pointer.

Another variation of our second solution where we change the original pointers by direct assignment can be implemented as follows:

Fig.17 Changing pointers by direct assignment — a recursive version.

Short as it is, this solution may be a little tricky to understand. All the iterative solutions we have seen previously employed a pointer *x* to iterate through the nodes until it was time to add the new node.

This version does things a little differently:

1. Like the previous versions of *insert()*, it starts off by checking if the root node is NULL. However, unlike all the other versions, it does not return NULL in this case. Let's ignore this entire if block for now. We know for a fact that the root node is not null in our example anyways.
2. Starting at line #11, we proceed to compare the data of the current node (8) with the value we're inserting (9). Since they are not equal, we move on to line #16.
3. We check if the value we're inserting (9) is less than the current node (8), which evaluates to false and we proceed to execute the code in the else block.
4. The code in the else block calls the same function recursively assigning the value it returns to *n->right*.

But wait: why are we modifying *n->right* at all? We don't know if it's the right place to insert our value yet. To answer this question, let's see what will happen when we execute line #22, i.e.

```
n->right = insert(n->right, value);
```

What are we assigning to the root node's right pointer? We assign to *n->right* whatever value the call to *insert(n->right, value)* returns. And what value may this call return? To answer this question, let's look at the program execution illustrated on Fig.18.

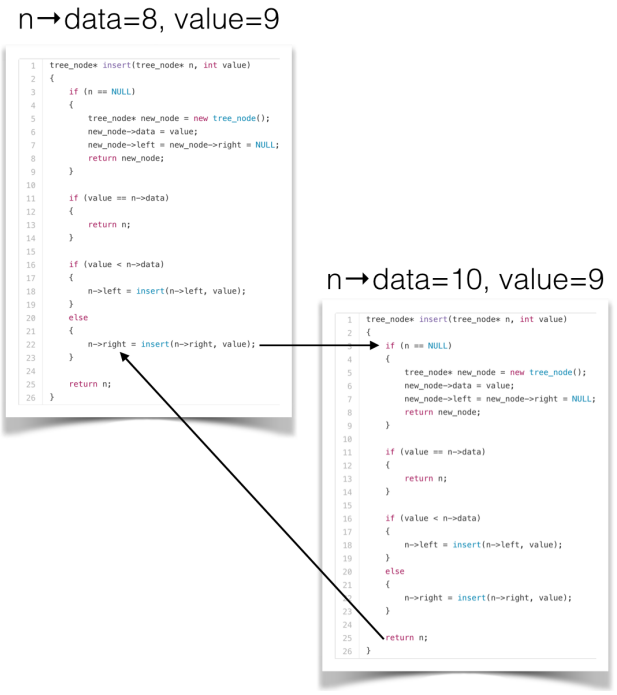

Fig.18 A closer look at the execution of the recursive version of insert().

- Calling *insert(n->right, value)* will start the same function *insert()* with *n* pointing to the node that carries the value 10.
- Without thinking about any further recursive calls, what does the second version of insert() do to *n?*
- Looking at line #3, *n* is definitely not NULL. Therefore, we proceed to line #11.
- On line #11, *value* is compared with *node->data*, 9 and 10 respectively. They're unequal and execution proceeds to line #16.
- Now regardless of what the code in the if-else block spanning lines 16 throughout 13 does, we can easily see that the code returns *n* — which is pointing to node with value 10 — in the end.
- This means that executing line #22 in the first copy of *insert()*, shown on the left side of Fig.18, has the effect of assigning the node with value 10 to the right pointer of the node with value 8. But this doesn't modify anything since they were already connected prior to this recursive call.
- This is definitely true. But what if the right pointer of the node (8) was pointing to NULL? What would have been the return value of the recursive call shown on the right side of Fig.18?

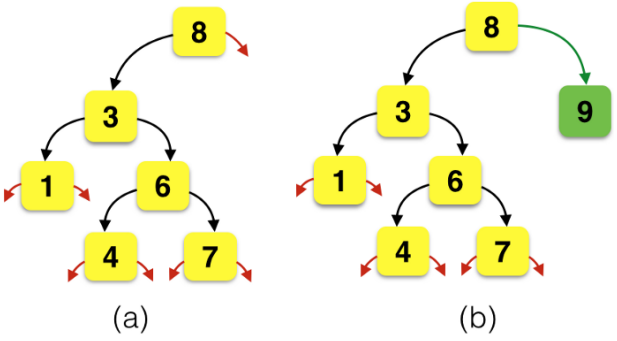

Fig.19 (a) A binary search tree where the right pointer of the root node points to NULL. (b) The same tree after executing the first recursive call of insert().

- The answer is simple: the pointer *n* in the recursive call would have been NULL, which would have caused the condition on line #3 to evaluate to true, hence returning a pointer to a newly allocated node that holds the value 9. This pointer to the new node would have in turn been assigned to the root's right pointer effectively adding the node (9) as the right node of the root.
- This is exactly how this version of the algorithm works; it keeps returning pointers to existing nodes until it encounters a NULL pointer, at which point it realizes this is the right moment to create the new node, assign the supplied value to it and return it to the parent node effectively assigning it to either the left or right pointers of the new node's parent node.

Summary

We have explored the pass-by-value semantics in C/C++ and discussed alternatives to use in case we would like to change variables or pointers passed to functions.

It is worth noting that C++ offers another option for changing pointers and variables in general — that is: references. You can think of references as an equivalent to pointers with some slight differences, e.g.

1. References use slightly different syntax than pointers, i.e.

A pointer to an integer is defined as follows:

```
int* p = &x;    // x is an integer that has been declared earlier
                // and p is a pointer to it*p = 10;        // this changes
the value of x
```

whereas a reference to an integer is defined as:

```
int& r = x;    // x is an integer that has been declared earlier
               // and r is a reference to itr = 10;        // this changes
the value of x just as well
```

2. An important difference between pointers and references that is evident from the above comparison is that pointers need the address-of operator *&* to be able to point to other variables. We cannot have *p* point to *x* by just assigning *x* to *p* directly. On the other hand, references point to other variables by assigning a variable directly to a reference, i.e. *x* to *r*, without any special operators.

3. Like pointers, you can use references to change the variables at which they’re pointing.

4. Unlike pointers, references cannot be NULL

5. References have to be given something to point to immediately upon declaration.

6. Unlike pointers, references cannot be pointed at other variables after they’re initialized.

7. Unlike pointers, we cannot have references-to-references. Otherwise, we would have been able to use a reference-to-reference to change what the original reference is pointing to just like what we did with pointers in the examples.

We have seen how we could change a variable or pointer by passing a pointer to it. We can achieve exactly the same effect by passing a reference to that variable or pointer we would like to change.

```
//
// process_ptr() can change what p is pointing to
//
void process_ptr(int* p)
{
   ...
}//
// process_ref() can also change what r is pointing to
//
void process_ref(int& r)
{
   ...
}//
// process() can change x but that won't affect the
// original value as it will be passed by value
//
void process(int x)
{
  ...
}int main()
{
    int i = 10;

        process_ptr(&i);    // process_ptr() can change i
    process_ref(i);     // process_ref() can change i as well
    process(i);         // process() can *not* change i
}
```

Some programmers prefer using pointers when passing variables to functions that could change their values because having to use a pointer or the address-of operator — as in the previous example — gives a strong reminder to the programmer that the function they're calling has the ability to change the value to which the pointer is pointing. On the other hand, calling a function that expects a reference looks exactly the same from a syntax perspective like calling a function that copies the parameters by value.

Although you cannot have a reference-to-reference, it is definitely valid to have a reference-to-pointer. A reference-to-pointer can be used to change what a pointer is pointing to in exactly the same way a pointer-to-pointer is used to achieve the same objective.

Long as it is, I hope that reading this article has brought you a step closer to understanding some of the basic yet commonly misunderstood aspects of using pointers. Please, do feel free to leave your comments, thoughts or questions.