

# Pointer

---

---

## 1. Motivation, Applicability, and Indications for Use

The programming type Pointer enables users to implement dynamic components, i.e. structures whose size may change during runtime. However, as will soon become abundantly clear, the pitfalls associated with the use of pointers are abundant; therefore, you must exercise extreme caution when using this data type. Remember this rule of thumb: whenever you want to implement a dynamic component and cannot effectively layer it on top of existing components, consider using pointers.

## 2. Related Components

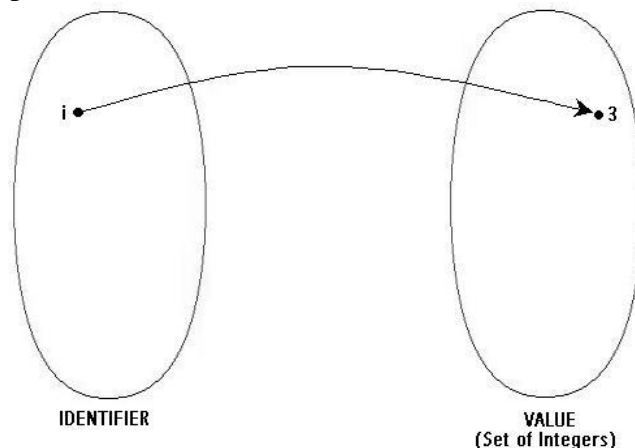
Pointers find their use in effectively implementing many components in the RESOLVE/C++ component catalog. Some examples are the List, Stack, and Queue data structures. Pointers are integral to the language, and are not layered on any other components, just as Integers within the catalog are not layered on any other components.

## 3. Modeling the Pointer component

Before we delve into the wonderful world of Pointers, we must revisit the old concept of an identifier (variable names or expressions like `array[ i ]`) and its value. Let us now consider the following example:

```
Object Integer i = 5;  
i = i + 3;
```

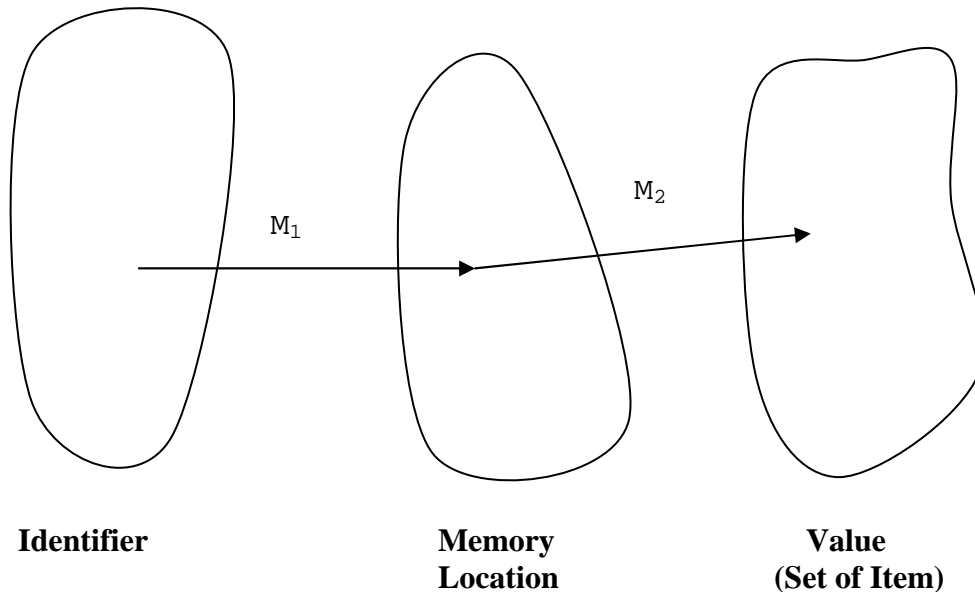
What does this code segment do? Identifier *i* is initialized with the integral value 5. In the second instruction, the value of the identifier *i* is fetched and then added to the value 3, the result of which is assigned back to the identifier *i*. We can model this idea using two sets, as shown in the following picture:



**Fig 1: Set Mapping of Identifier -> Value**

By now, you must be wondering why we went to so much trouble to explain such a simple concept? This is because pointers introduce a new dimension to this concept. With pointers, the identifier does not map to a value, but to another identifier, traditionally referred to as a “memory location.” This memory location is the real identifier that points to the value. In short, treat the *memory location*, not the pointer identifier itself, as you would any other variable.

The following picture is a slightly more accurate representation of the preceding idea of a pointer:



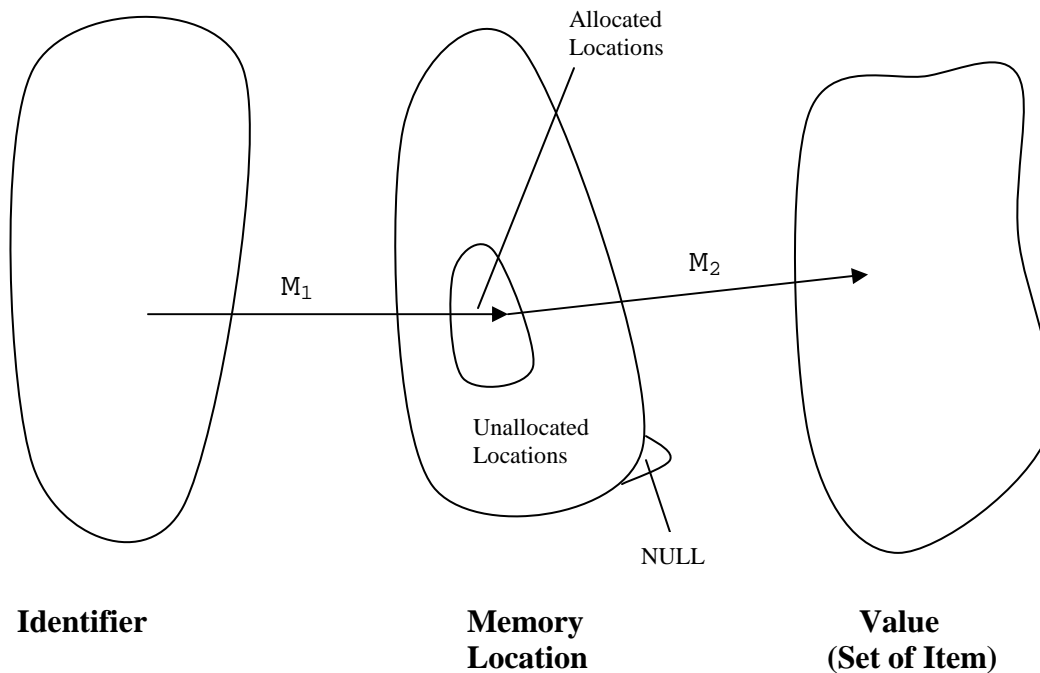
**Fig 2: More Accurate representation of a Pointer**

Notice that we have proceeded so far without telling you what the arrows actually mean. In Fig 1 you probably assumed that the arrow represented the ‘equals’ relationship. This was certainly the idea that Fig 1 was meant to convey. In Fig 2, the mapping labeled  $M_1$  simply tells us that an element of Identifier has a corresponding Memory Location associated with it, and the other mapping tells us that an element of Memory Location has a corresponding value.

Another question we have happily avoided thus far, relates to the cardinalities of the three sets that we use in our model, especially of the set we call Memory Location. Unfortunately, we do not live in an ideal world where resources are unlimited. Every object uses a certain amount of computer resources, depending on the complexity of the object. To deal with this problem, real world pointers are initialized without any well-defined maps, so that no resources are taken up initially. At this point, we say that the Pointer belongs to *Unallocated Memory*. Later, when the Pointer gets access to resources, we say that it belongs to *Allocated Memory*. To enable our model to represent this idea, we must think of our Memory Location set as being the disjoint union of two sets: Allocated Location set and Unallocated Location set.

There is one more matter we need to discuss before we get comfortable with our model of the Pointer component - the idea of NULL, a special purpose memory location with the restriction that it is never in the domain of  $M_2$ . Therefore, we can add NULL to the Memory Location set as well.

The following figure reflects these modifications to our model:



**Fig 3: Final model for the Pointer component**

Try to compare the ideas of Allocation and Deallocation from the mailbox and tent models with the way they are presented here.

Now, if you did not fully understand the development of this model thus far, do not despair! The next section will give us some intuition into the working of this model by discussing various operations we can perform on pointers.

## 4. OPERATIONS

Now that we have introduced Pointers, the next question we should ask ourselves is: How do I use them?

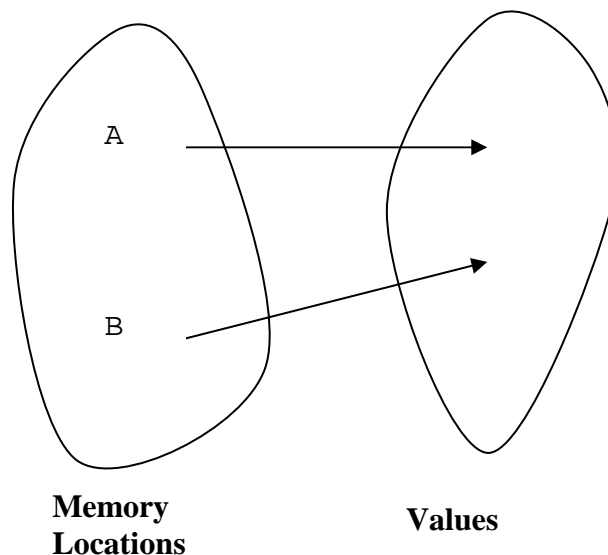
Pointers have several standard operations that can be performed to manipulate the relationships between the various mappings in our model. Most of these operations are explored in detail in the following paragraphs.

### Dereferencing

The pointer dereferencing operation allows us to modify the data stored at the memory location allocated to the pointer. It actually returns a memory location, which may be used just like a classical (non-pointer) identifier in expressions. The '\*' operator is used to indicate Dereferencing in RESOLVE/C++. For example, suppose we had a pointer to an integer called `MyIntegerPointer`, and wanted to increment the value stored at its memory location by 3, we would use the following command:

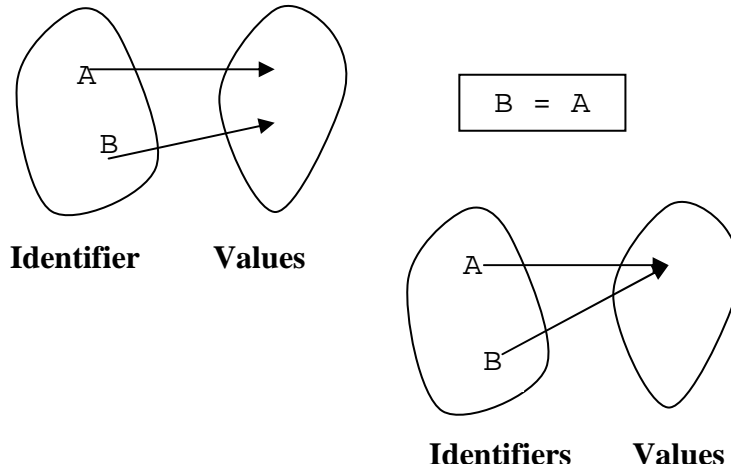
```
*MyIntegerPointer = *MyIntegerPointer + 3
```

In effect, dereferencing allows us to consider only the set of memory locations (used as traditional identifiers) and the set of values associated with each of those memory locations.

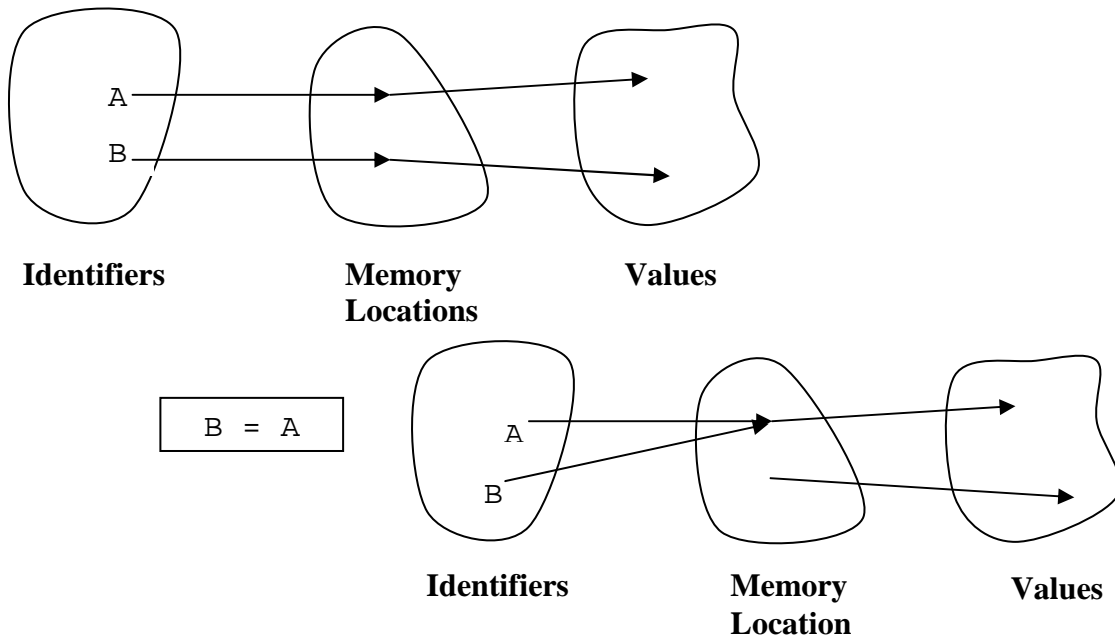


## Assignment

The pointer assignment operation does not work like assignment for other types. The non-pointer assignment operator maps an identifier to the value of another identifier.



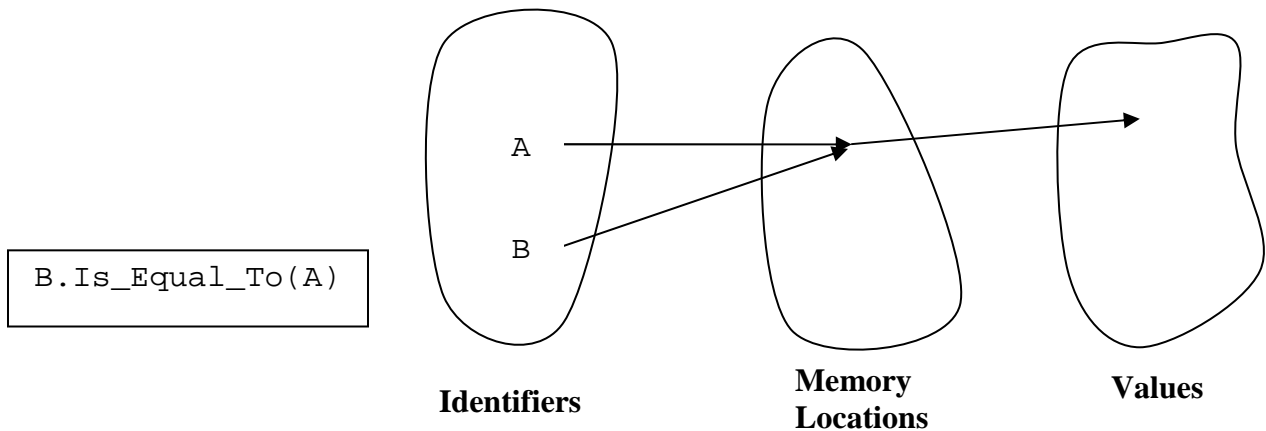
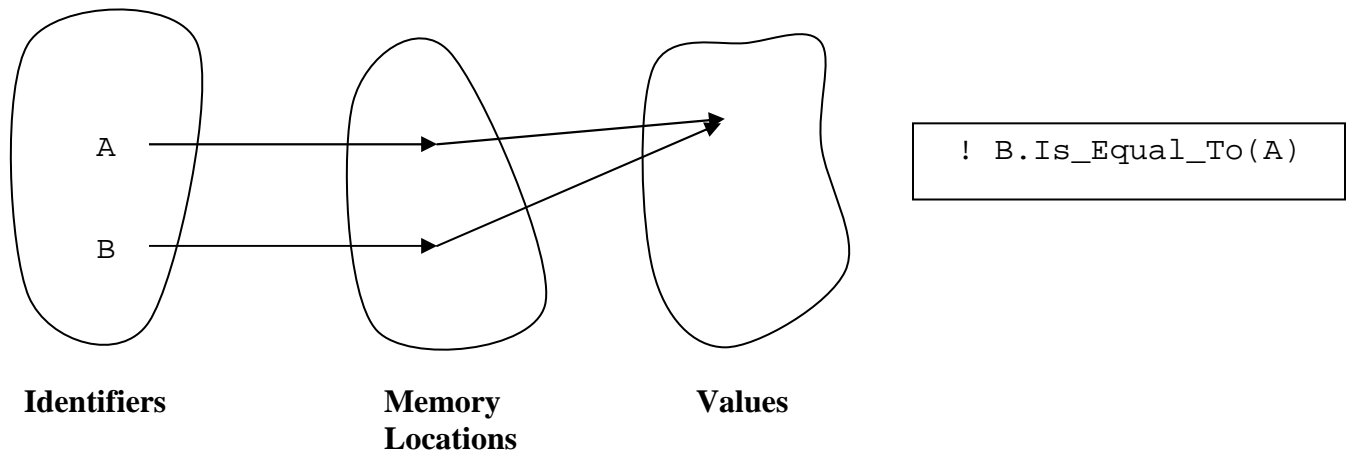
With pointer assignment, one identifier maps to the memory location mapped to by another identifier.



This behavior is consistent because non-pointer assignment follows one mapping from the identifier to values. Likewise, pointer assignment follows one mapping from identifier to memory location. The problem with pointer assignment is that now, both pointers dereference the same memory location: if one pointer's value is changed, so is the other's. This is known as aliasing, and is further discussed under Pointer Pitfalls.

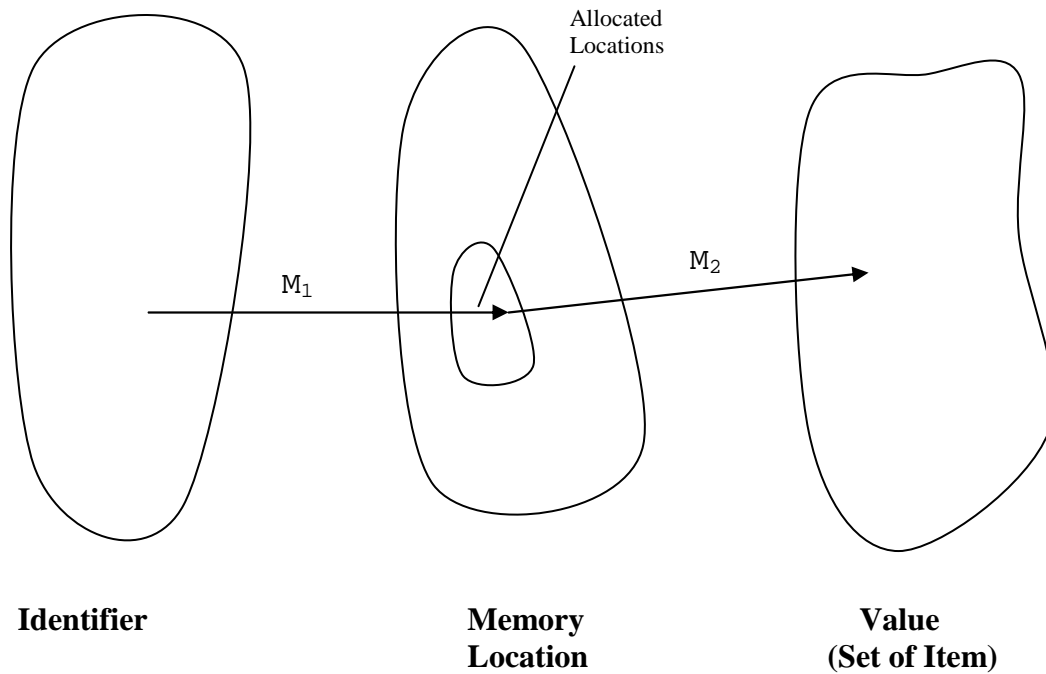
## Equality Testing

The pointer equality testing operations do not serve the same purpose as equality testing for other types. Non-pointer equality testing tests the equality of the values mapped to by the given identifiers. On the other hand, pointer equality testing determines if two memory locations are exactly identical. That is to say, the two identifiers map to the same memory locations. In effect, equality testing determines if two pointers are aliased (cf. Pointer Pitfalls).



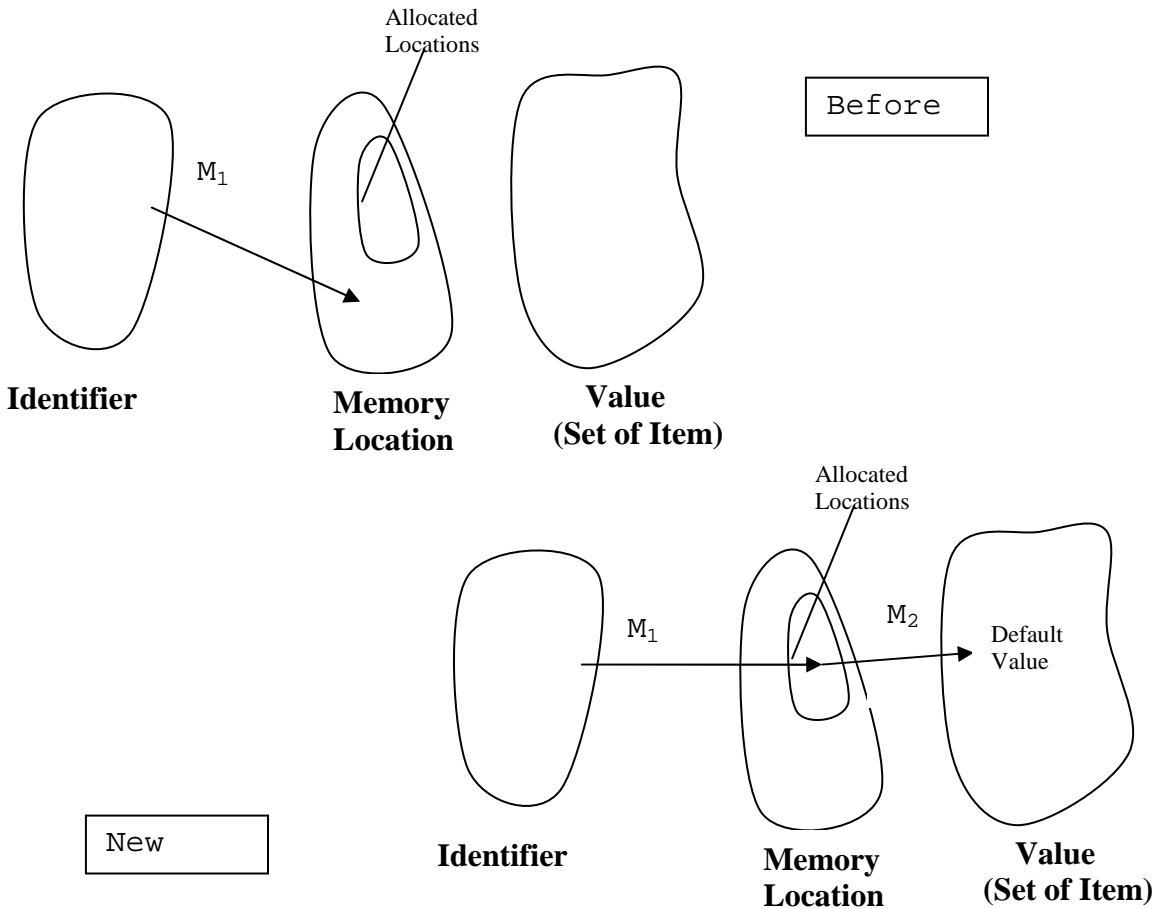
## Real-world computers

As we mentioned in Section 3, before a Pointer receives its own resources, the Allocation operation must be applied. This operation defines the missing map and allocates the necessary computer resources. The Deallocation operation is the inverse, removing the said map and freeing resources. A deallocated pointer cannot be used again until it is reallocated.



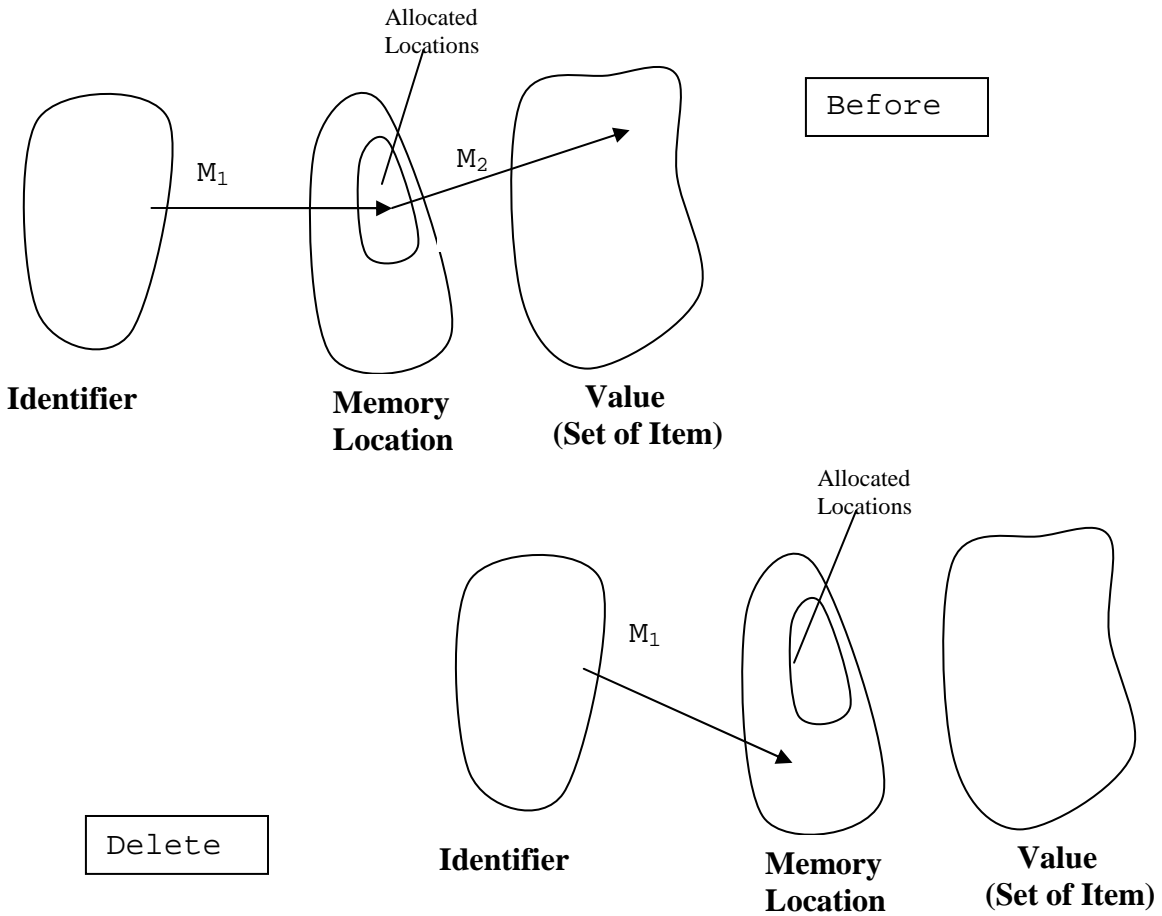
## Allocation - New

This operation actually provides the pointer with its own resources. In other words,  $M_1$  now maps to the Allocated subset of Memory Location.  $M_2$  then maps this new point in Allocated Location to the initial value for the object being considered. This initial value is predetermined for almost all objects in the RESOLVE catalog, with Pointers themselves being the only exception. In effect, we can consider the initial value to be set to the return value of the math operation 'random'.



## Deallocation - Delete

This operation frees the locations used by the pointer, i.e.,  $M_1$  will map to the unallocated subset of Memory Location after this operation.



## 6. Tracing Tables

### Declaration

```
concrete_instance class Integer_Pointer:  
    instantiates Pointer<  
        Integer>  
{};
```

```
object Integer_Pointer ptr1;
```

Alternately, we could just use `object Pointer<Integer> ptr1;` to declare `ptr1`.

### Associated Operations

---

The tracing tables below assume that the following concrete class instantiation has been performed:

```
concrete_instance class Integer_Pointer:  
    instantiates Pointer<  
        Integer>  
{};
```

### New (Pointer<Item> p)

The New operation is the RESOLVE version of the Allocation operation.

Statement	Object values
<code>object Integer_Pointer ptr1;</code>	<code>ptr1</code> maps to an element in the unallocated subset of memory location.
<code>New(ptr1);</code>	
	<code>ptr1</code> maps to an element in the allocated subset of memory location. Also, <code>*ptr1 = 0</code> .

### Delete (Pointer<Item> p)

The Delete operation is the RESOLVE version of the Deallocation operation.

Statement	Object values
<b>object</b> Integer_Pointer ptr1; New(ptr1);	ptr1 maps to an element in the allocated subset of memory location. Also, *ptr1 = 0.
Delete(ptr1);	
	ptr1 maps to an element in the unallocated subset of memory location.

### Dereference (\*)

Statement	Object values
<b>object</b> Integer_Pointer ptr1; New(ptr1);	ptr1 maps to an element in the allocated subset of memory location. Also, *ptr1 = 0.
*ptr1 = 5;	
	ptr1 maps the same as before. *ptr1 = 5

### Assignment (=)

Statement	Object values
<b>object</b> Integer_Pointer ptr1, ptr2; New(ptr1);	ptr1 maps to an element in the allocated subset of memory location. Ptr2 maps to an element in the unallocated subset of memory location. Also, *ptr1 = 0.
ptr2 = ptr1;	
	ptr1 and ptr2 are aliased: ptr1 and ptr2 map to the same element in the allocated subset of memory location. *ptr1 = *ptr2 = 0.

## Equality Testing (== and !=)

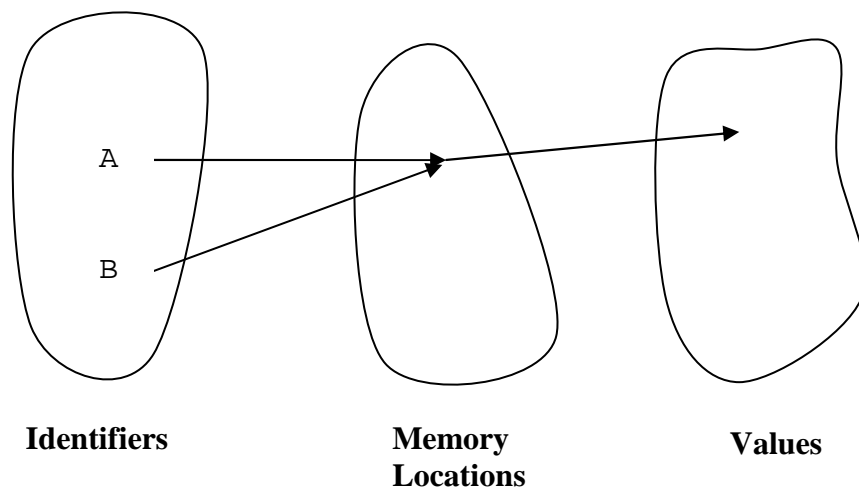
Testing two pointers for equality tests whether the two pointers map to the same element in the set of memory locations. Testing two dereferenced pointers for equality determines if their memory locations map to the same value.

Statement	Object values
<pre><b>object</b> Character_OStream out; out.Open_External("");  <b>object</b> Integer_Pointer ptr1, ptr2, ptr3;  New(ptr1); New(ptr3); *ptr3 = 4; ptr2 = ptr1;</pre>	
<pre><b>if</b> (ptr1 == ptr2)     out &lt;&lt; "ptr1 == ptr2";</pre>	<pre>ptr1 == ptr2</pre>
<pre><b>if</b> (ptr1 != ptr3)     out &lt;&lt; "ptr1 != ptr3";</pre>	<pre>ptr1 != ptr3</pre>
<pre><b>if</b> (*ptr1 != *ptr3)     out &lt;&lt; "*ptr1 != *ptr3";</pre>	<pre>*ptr1 != *ptr3</pre>
<pre>out.Close_External();</pre>	

## 7. Pointer Pitfalls

- Aliasing

This happens when you point two pointers to the same location in memory. You must remember that since both pointers are accessing the same location in memory, any change to the value that one pointer points to results in changing the value that the second pointer points to as well. Not only does this complicate the process of reasoning about our programs, but it can have dangerous consequences, as we shall see later on in this section. The following Figure illustrates the idea of aliasing.



- Dangling Reference

Consider the following RESOLVE/C++ code segment:

```
1: object Pointer_C<Integer> ptr1, ptr2;  
2: New(ptr1);  
3: ptr2 = ptr1;  
4: Delete(ptr1);
```

After the execution of line 3, ptr2 points to the same location as ptr1. Therefore, aliasing occurs, and the deletion of ptr1 in line 4 makes the previously used address available to other programs. At this point, dereferencing ptr2 can easily cause your program to terminate unexpectedly since you probably do not have access to the memory anymore.

- **Storage Leak**

A simple example of a storage leak is illustrated in the following code segment:

```
1: object Pointer_C<Integer> ptr1;  
2: New(ptr1);  
3: New(ptr1);
```

In line 2, memory for one integer variable has been allocated to *ptr1*. However, in line 3, memory is allocated a second time and this causes the leakage of memory since the memory address allocated in line 2 has been lost. Therefore, unlike other objects in C++ that the compiler automatically deletes when out of scope, you must remember to deallocate all memory allocated for the use of pointers. Failure to do this, especially in large programs can cause you to easily run out of heap space and subsequently crash the program (Note: Don't try this on the CIS servers!) Have you noticed how many programs that you use everyday work slower after being run for a long time? This is caused precisely because of storage leaks within these programs that consume more and more of the available memory over time.

## Pointer\_C

Fortunately, the RESOLVE/C++ catalog provides a solution to help you detect the problems mentioned above when using pointers – the checked pointer, *Pointer\_C*. This component tests your usage of pointers for common errors and notifies the user of the exact nature of the error. In the following section, we proceed to explore some of the functionality provided by *Pointer\_C*.

- **Report\_Storage\_Allocation**

This procedure does exactly what its name suggests; it allows you to find out how much memory you have allocated using Pointers. You must realize that if run at the end of your program, you usually want the *Report\_Storage\_Allocation* operation to produce no output, since that means you have no storage leaks in your program. In this section, we provide a storage allocation report for the program displayed below. This program is stored in a file named *Storage.cpp*.

```
#include "RESOLVE_Foundation.h"  
  
concrete_instance class Integer_Pointer:  
    instantiates Pointer_C<  
        Integer >  
{};  
  
procedure_body main() {  
  
    object Character_OStream out;  
    out.Open_External("");  
  
    object Integer_Pointer ptr1;  
    New(ptr1);  
    New(ptr1);  
}
```

```
    Report_Storage_Allocation();
    out.Close_External();
}
```

The report issued looks like the following:

```
=====
Pointer report: currently allocated memory locations:
-----
0x6b300 (allocated from line 15 of file Storage.cpp)
0x6b2f0 (allocated from line 14 of file Storage.cpp)
=====
```

## 8. References & Resources

- Pike, S.M., Weide, B.W., and Hollingsworth, J.E., "Checkmate: Cornering C++ Dynamic Memory Errors With Checked Pointers," *Proceedings 31st SIGCSE Technical Symposium on Computer Science Education*, ACM, March 2000,352-356.

Available at <http://www.cis.ohio-state.edu/rsrg>

- Informal introduction to Pointers video  
Nick Parlante, "Pointer Fun With Binky," *Stanford CS Education Library*

Available at <http://cslibrary.stanford.edu/104/>