

## Dynamic Memory Allocation.

\* There are two places that memory comes from in C++. One is called "the stack". The other is called "the heap." So far you've been claiming memory from the stack, which is the safest and easiest place from which to get it. The memory is reclaimed by the program automatically when variables go out of scope.

\* This way of claiming memory, however, is not always adequate.

\* Often, the allocation needs to change when the size of something grows.... think of your stack.

\* Often we need finer control over when memory is reclaimed.

\* E.G. suppose I am writing a plotting package. While a user is running the package, he/she requests a new view. Control passes to a function

```
view *  
manufactureView(double height, double width);
```

that manufactures a view, which results in a window coming up on the screen.

The program then uses this new view, draws into it, handles the users mouse clicks whenever they land on something therein, & cetera.

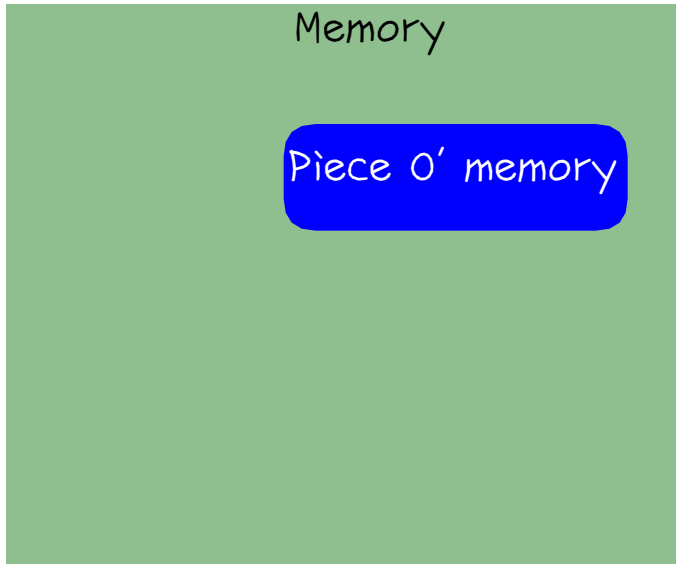
Later the user decides to get rid of the plotting window. The program needs to remove the "view" from the screen... A function called

```
void removeView (view * vw);
```

removes the view \* from the list of views being drawn by a view manager, and then releases the memory previously used by the view.



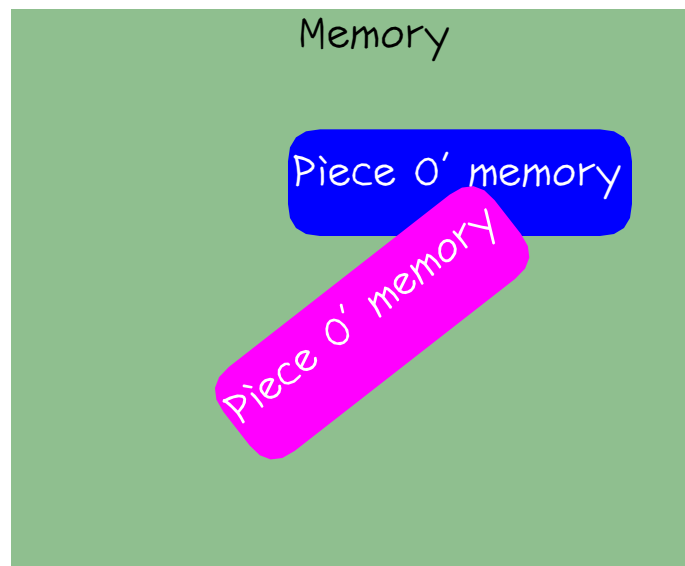
Memory overwrites:



You allocate a piece of memory. Before you are done using the object you inadvertently release it.

The memory is then no longer reserved for you.

Another memory allocation takes place, and uses your memory for its own data.



Both problems are unfortunately too common. Care is required to prevent them from happening, especially in C++. When the inevitable does occur, specialized debugging tools (e.g. purify) can be used to track down and eradicate memory leaks.

\* Here is how you allocate memory from the heap in C++.

```
int * myNewInteger = new int;
```

\* Here's how to delete it:

```
delete myNewInteger;
```

\* New returns a pointer of a new piece of memory in the heap.

Note: this means that the expression `new int` has both a type in this case, `int *`, and a value (the address).

\*New will return 0 if the request for memory cannot be satisfied. The return value should be checked for this (hopefully rare) condition. This is overlooked by almost everyone. Check Meyers item 7 for the solution to this pitfall.

\*You are allowed to

```
delete 0;
```

It has no effect.

A typical scenario is the following:

```
view *myView=0; // object can exist, or not,
                // at beginning of execution
                // it has not yet been created.

myView= new view; // now it is created.

if (myView) { // if it exists, you can send it
    (*myView).popup(); // messages. This one tells the
} // view to pop up.

if (myView) { // Delete the pointer. Unless the
    delete myView; // pointer itself is going out of
    myView=0; // scope, its best to set its value
} // to zero
```

C conscenti beware: don't mix new & delete with malloc and free.  
In fact, don't use malloc and free anymore.

\* You can also use new and delete to allocate arrays of things.

\* To do this, follow the following syntax:

```
int * myIntArray = new int [24];
```

\* You do not need to use a constant as the dimension for this form of new. You can use a variable.

\* when you free the memory, you now need to free an array.

\* you have to "tell this" to the delete operator.

\* use the following form of delete:

```
delete [] myIntArray;
```

\* You can use new and delete to allocate single user-defined objects and arrays thereof.

\* New calls the class constructor.

\* Delete calls the destructor.

\* You can provide arguments to the constructor like this:

```
complex *myComplexNumber=new complex(1,-1);
```

\* You cannot provide constructor arguments for the objects constructed in an array (logical, if you think about it).

\* Corollary: an object must have a default constructor in order to allocate it in an array.

\* In fact, this is also necessary to create an array of objects on the stack.

## How do I reallocate an array?

```
int *data=0;           // no memory usage;
int size =0;          // size of initial allocation
int increment=10;     // allocation increment

if (count==size) {   // end of array
    size+=increment; // increase allocation
    int *newLocation = new int[size]; // create new space
    for (int i=0;i<count;i++) { // copy old data to
        newLocation[i]=data[i]; // new location
    }
    delete data;      // delete the old data.
    data = newLocation; // set the pointer-to
}                    // data to the new
                    // location
```

\* Exercise: modify your stack class so that it allocates its internal data array dynamically. Take the following steps.

\* clean up your istack class, removing any extraneous print statements from previous exercises.

\* replace the `_data` data member by a pointer to an integer.

\* introduce an additional data member to store the number of integers that have been allocated.

\* introduce a constant data member to store the size of each increment, and initialize it to 10.

\* When the count exceeds the allocation, reallocate the memory as demonstrated above.

You will need to introduce changes in the header file, and also in the file `istack.cc`. In `istack.cc`, you will need to revisit the following routines:

- \* constructor
- \* copy constructor
- \* assignment operator
- \*push function.

\* Congratulations, you have now created a class that performs dynamic memory allocation. The size of the object now adjusts itself automatically to a the size of the stack, and there is no predefined limit to the stack size. This is no doubt superior to your first implementation of the stack.

\* But, now you have to confront some difficulties.

\* Your class probably memory in the push function. Each resize (if you succeeded last time) cleans up after the previous memory allocation. But nobody cleans up the last allocation.

\* To plug these leaks, we'll modifying the destructor, the copy constructor, and the assignment operator.

\* This is the main reason you are allowed to write your own versions of these important routines.

\* Here is a rule to live by: call delete on pointer members in destructors (Meyers, item. 6).

\* Here is another rule: define a copy constructor and an assignment operator for classes with dynamically allocated memory (Meyers, item 11).

Let's look carefully at some of the problems & their solutions.

- \* Objects are autonomous little buggers. They hide their data, including any pointer data.
- \* If you can't trust the object to clean up after itself, who can you trust??
- \* So there is this notion of "who owns the memory?" that you often hear when discussing this kind of thing.
- \* The only reasonable response is, "if you allocate it, you clean it up".
- \* If an object is in the process of being destroyed, it better do this with its dying gasp...

Hence:

"call delete on pointer members in destructors"

Reminder:

use the same form in corresponding calls to new and delete, (forgetting this is another recipe for a memory leak).

And one more warning:

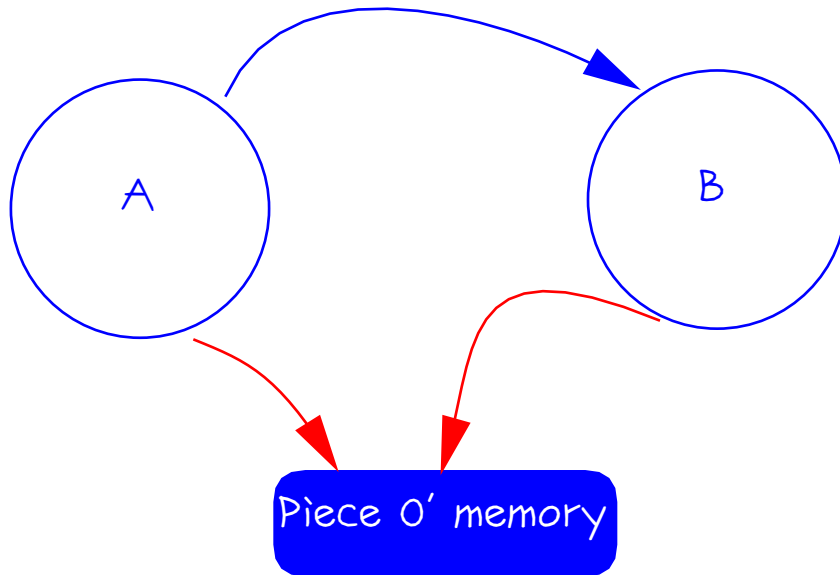
If no memory is to be allocated to a particular pointer in a constructor, initialize its value to zero.

**Exercise:** now, do this in your stack class.....

Note: you are only halfway through the changes, and depending on how your test program uses your improved class, it may crash with a core dump...

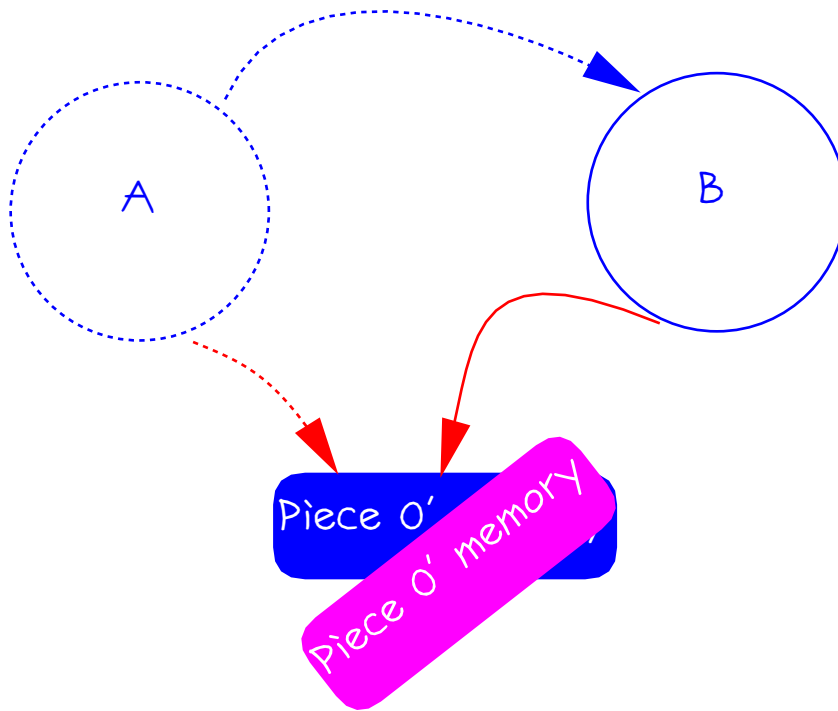
\* Now that your objects clean up after themselves, you have another problem.

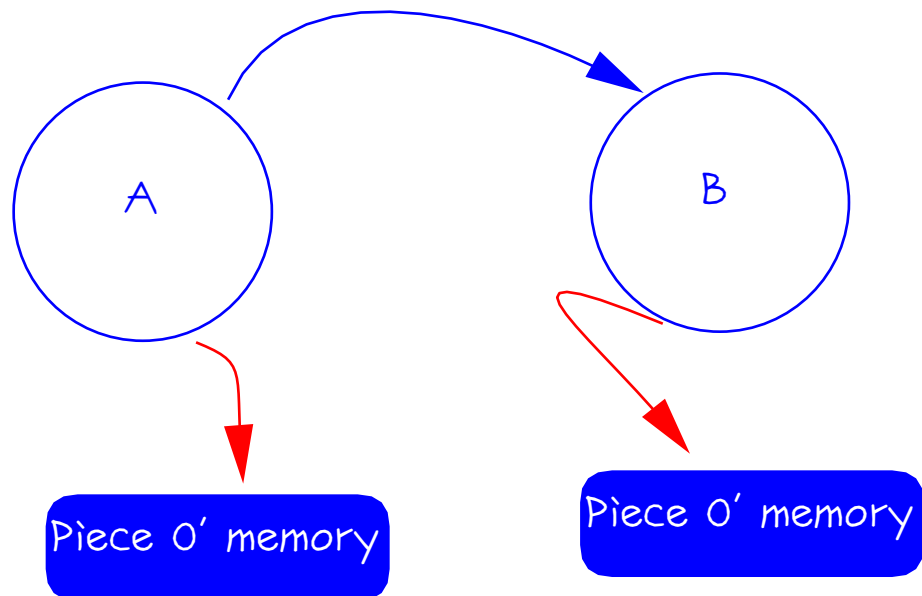
\* That problem arises when you copy your object. When you copy A to B, by default you do a memberwise copy. This means that the pointer to A's dynamically allocated memory is identical to the pointer in B's memory...i.e, the memory is the same.



\* This is bad news, because operations on object A result in changes to object B. This is clearly wrong! (Push data onto one stack and another stack gets larger???)

\* The worst problem is: when A (or B) is destroyed, the memory is released, so some other clown can now clobber it:





The solution to this nefarious state of affairs is: don't copy the pointer, **clone the memory**.

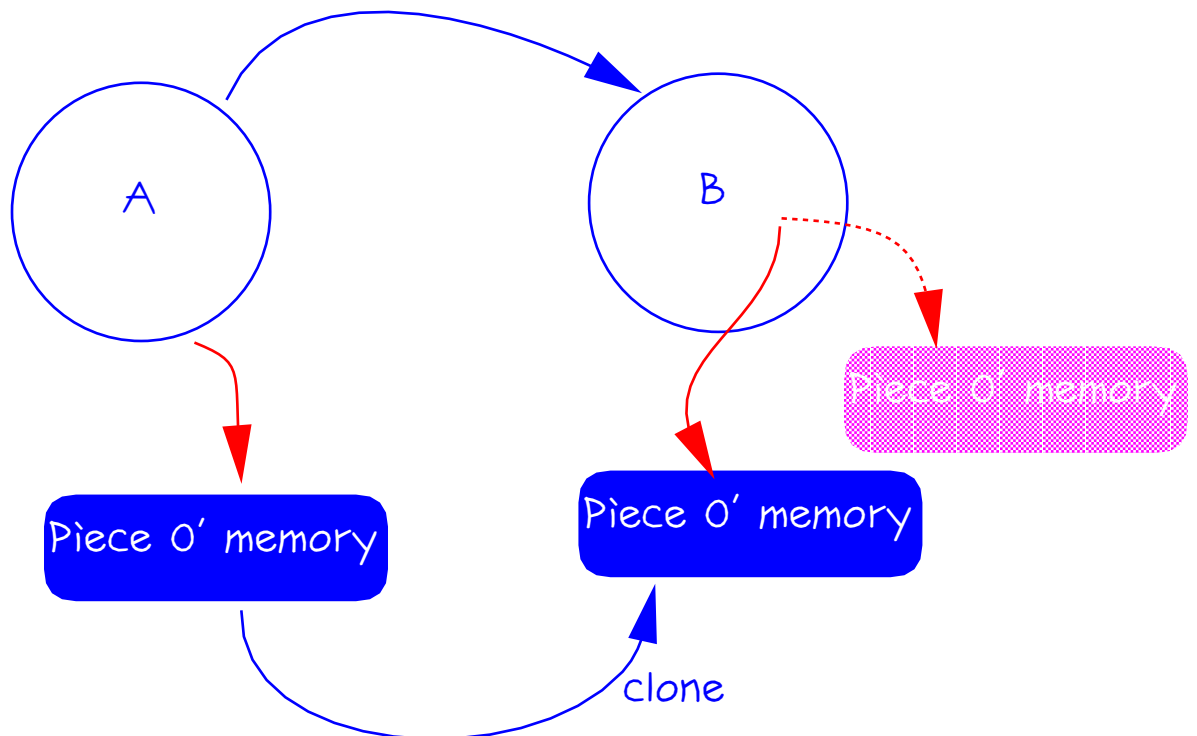
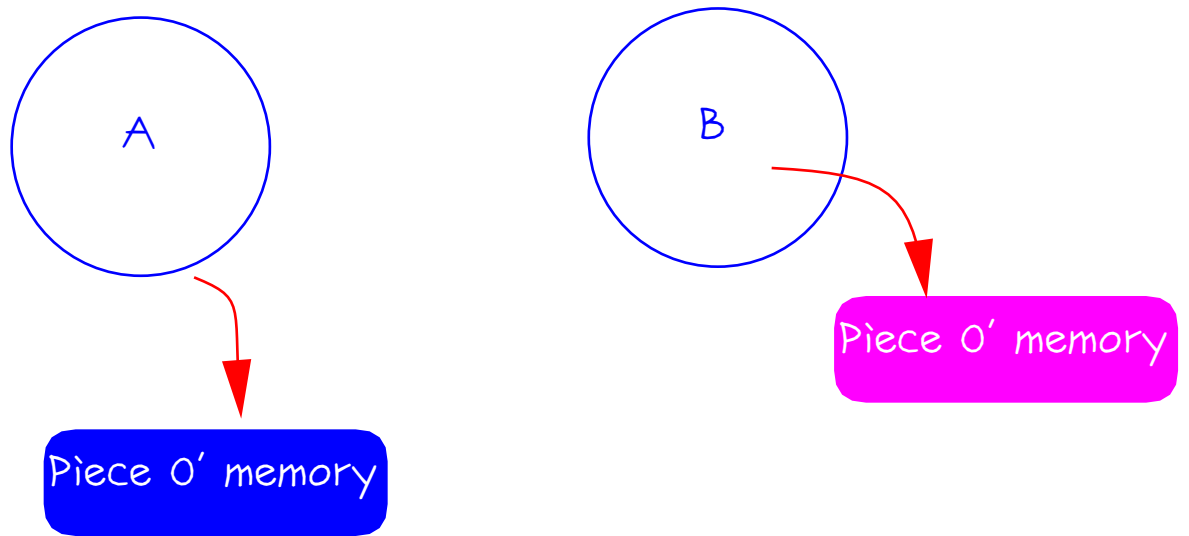
In the copy constructor:

```
_data = new int [right._size];
```

(For the stack class, if you used the variable `_size` to denote the size of the memory allocation. In general however, you have to do whatever it takes to clone the memory...)

Note that the default copy constructor, by performing a memberwise copy does the wrong thing. That's why you must write your own copy constructor.

Finally the assignment operator has the same problem as the copy constructor, but it has another problem as well. Assignment copies an objects to an existing object, which most probably has some of its own dynamically allocated memory lying around.....



=> make sure you delete the existing memory before you overwrite the pointer!

Exercise: fix the assignment operator for the istack class.