An Overview of Dynamic Memory

Generally, space to hold variables is allocated at compile time when the program is built or at runtime when space to hold local variables is allocated in the stack at runtime. Global memory and that allocated to variables in main aside, local variables have a transient lifetime. They appear when we enter their context and disappear when we leave; for example, several of the variables in the last homework assignment. Their allocation and deallocation is managed by the system. Note that when local variables are deallocated, they are not morphed into some third dimension where they disappear forever; nor are they set to any particular value. They still exist, with their latest values, on the stack memory (use your debugger to see).

Dynamic memory is a scheme that gives the user control (and responsibility) for such allocation and deallocation at runtime. Now the user says when the space for the variables is allocated and when it is deallocated. Like the local variables, when dynamically allocated memory is deallocated, it also is not morphed into some third dimension where it disappears forever; nor is it set to any particular value. It still exist, with its latest values, at the same address from which it was allocated.

Dynamic memory (which is also called the heap) is no different from any other RAM (random access memory). It is RAM…same color, same size, same shape, same smell, same taste. Like the stack, there is a system level software driver that manages that piece of memory, just like that which manages the stack, to give it its behaviour.

The C language provides the two operators malloc and free as the user interface to the heap. C++ replaces those with the operators new and delete. The new and delete operators come in several flavours.

```
new type;
new type [amount];

delete pointer
delete[] pointer
```

If the allocation succeeds, i.e. if you have not run out of heap, the allocation operators will return a pointer to the piece of memory that has been allocated to you…don't loose it.

When the allocation fails…now another change (or addition to) the language. Prior to exceptions being added to the language, on failure, the new operator would return NULL; now it throws a failure to allocate exception…which you must catch and handle.

If you are not managing exceptions (which we are not right now) then you can still utilize the earlier behaviour (which we will) by writing the new operators as:

```
new (nothrow) type;
new (nothrow) type [amount];
```

Since it is possible to run out of heap, you must always check an allocation to determine if it succeeded. Thus we write,

```
#include cassert;
```

assert (new (nothrow) type);

or

assert (new (nothrow) type [amount]);

If the allocation fails, the program will terminate…for now, this is fine. When you learn about exceptions, then you will use these.

When you dynamically allocate a piece of memory, the system underware, sets aside that amount of memory for you…basically puts your name on it…and returns the address of that piece to you. When you are allocating a block using the [], the same process occurs, however, the underware also makes a note of the size of that block and stores that with your allocation. So, when you return the memory through the delete[] operator, it knows that it has to delete the appropriate amount. It then takes your name off the list as owning that piece of memory and it is returned to the heap.

As we observed earlier, the underware does nothing to that memory than taking your name off of it. The values all remain. If you dereference it through the original pointer, you can still read those values. However, like writing beyond the end of a container, you do not own that memory.

This is dynamic memory in a nutshell.