Dynamic Memory Allocation

Allocating memory

There are two ways that memory gets allocated for data storage:

1. Compile Time (or static) Allocation
   - Memory for named variables is allocated by the compiler
   - Exact size and type of storage must be known at compile time
   - For standard array declarations, this is why the size has to be constant
2. Dynamic Memory Allocation
   - Memory allocated "on the fly" during run time
   - dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
   - Exact amount of space or number of items does not have to be known by the compiler in advance.
   - For dynamic memory allocation, pointers are crucial
   -

Dynamic Memory Allocation

- We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"
- For this reason, dynamic allocation requires two steps:
  1. Creating the dynamic space.
  2. Storing its **address** in a **pointer** (so that the space can be accesed)
- To dynamically allocate memory in C++, we use the **new** operator.
- De-allocation:
  - Deallocation is the "clean-up" of space being used for variables or other data storage
  - Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
  - It is the programmer's job to deallocate dynamically created space
  - To de-allocate dynamic memory, we use the **`delete`** operator

Allocating space with **`new`**

- To allocate space dynamically, use the unary operator **`new`**, followed by the *type* being allocated.

```
new int;        // dynamically allocates an int
new double;     // dynamically allocates a double
```

- If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
new int[40];      // dynamically allocates an array of 40 ints
new double[size]; // dynamically allocates an array of size doubles
```

```
                      //  note that the size can be a variable
```

- These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the `new` operator returns the starting address of the allocated space, and this address can be stored in a pointer:

```
int * p;        // declare a pointer p
p = new int;    // dynamically allocate an int and load address into p

double * d;     // declare a pointer d
d = new double; // dynamically allocate a double and load address into d

// we can also do these in single line statements
int x = 40;
int * list = new int[x];
float * numbers = new float[x+10];
```

  Notice that this is one more way of *initializing* a pointer to a valid target (and the most important one).

Accessing dynamically created space

- So once the space has been dynamically allocated, how do we use it?
- For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:

```
int * p = new int;    // dynamic integer, pointed to by p

*p = 10;              // assigns 10 to the dynamic integer
cout << *p;           // prints 10
```

- For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:

```
double * numList = new double[size];  // dynamic array

for (int i = 0; i < size; i++)
    numList[i] = 0;                   // initialize array elements to 0

numList[5] = 20;                      // bracket notation
*(numList + 7) = 15;                  // pointer-offset notation
                                      //   means same as numList[7]
```

Deallocation of dynamic memory

- To deallocate memory that was created with `new`, we use the unary operator **`delete`**. The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int;          // dynamically created int
// ...
delete ptr;                   // deletes the space that ptr points to
```

  Note that the pointer `ptr` *still exists* in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
  ptr = new int[10];            // point p to a brand new array
```

- To deallocate a dynamic array, use this form:

  ```
  delete [] name_of_pointer;
  ```

  Example:

  ```
  int * list = new int[40];     // dynamic array

  delete [] list;               // deallocates the array
  list = 0;                     // reset list to null pointer
  ```

  After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

- **To consider:** So what happens if you fail to deallocate dynamic memory when you are finished with it? (i.e. why is deallocation important?)

Application Example: Dynamically resizing an array

If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones. Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else. For that reason, the process takes a few more steps. Here is an example using an integer array. Let's say this is the original array:

```
 int * list = new int[size];
```

I want to resize this so that the array called **list** has space for 5 more numbers (presumably because the old one is full).
There are four main steps.

1. Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).

   ```
   int * temp = new int[size + 5];
   ```

2. Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.

   ```
   for (int i = 0; i < size; i++)
      temp[i] = list[i];
   ```

3. Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out the garbage!)

   ```
   delete [] list;  // this deletes the array pointed to by "list"
   ```

4. Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.

```
      list = temp;
```

That's it! The list array is now 5 larger than the previous one, and it has the same data in it that the original one had. But, now it has room for 5 more items.