C++ Foundation with Data Structures

Notes: Dynamic Memory Allocation

Types of memory allocation

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

1. *Compile Time (or static) Allocation*

In this, 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. Statically allocated space used in a program is called *Stack*

2. *Dynamic Memory Allocation*

In this, memory is allocated during run time. Dynamically allocated space usually placed in a program segment known as the *heap.* 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.

Scope and Lifetime

We have studied scope and lifetime of variables earlier, now lets relate them with the way they are allocated. Scope and lifetime for a variable in stack are same as the way have studied, so memory for variable created this way is de-allocated or freed as soon as its lifetime expires. Whereas for a variable created dynamically concept for remains the same but lifetime of a heap variable is the whole program that means even if its scope has been finished, lifetime still remains intact till the end of program. That is why for dynamically created variables, we need to manually de-allocate memory after its usage.

The Process of Dynamic Memory Allocation

Allocation:

We can dynamically allocate storage space while the program is running, but we cannot create new variable names on the fly. 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 scope. It is the programmer's job to deallocate dynamically created space. To de-allocate dynamic memory, we use the **delete** operator

Now let's see how to write codes for dynamically allocating and deallocating variables one by one.

Allocating space with new

Following is the syntax to dynamically allocate an integer variable -

```
int *a = new int;
```

Above statement declare a pointer a and allocate an int from heap at run time and a points to the address of heap memory assigned.
When this statement is executed a total of 8 + 4 bytes memory is allocated. 8 bytes for pointer (a) in stack and 4 bytes for integer in heap at run time.

Similarly consider the statement below,

```
double *d = new double;
```

In the above examples pointers a and d are statically stored at compile time and int and double are dynamically created. We cannot directly refer to memory in heap as there is no symbol table for heap. So we have to use pointers.
Now we can redefine the above two examples in two steps for more understanding.

```
int * a;        // declare a pointer p
 a = 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
```

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc and new operator returns a pointer. Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new int;
if (!p)
{
  cout << "Memory allocation failed\n";
}
```

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
```

In above example dynamically allocated integer p is pointed to by p and then is assigned 10 via dereference operator and then we just printed value of address of p

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]
```

In above example we declared a dynamic array with values from heap, then we used normal bracket notation we use for static arrays to operate with it

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 i.e. it should be of pointer type:

```
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 determined at compile time. It can be reused:

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 = NULL;                  // reset list to null pointer
```

In above example, we deallocated memory with delete operator.But as list is not finished with deallocation of heap memory, it is reassigned to NULL.
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? It will still remain in heap memory till its lifetime but it would be like a dangling pointer, i.e. you might not have its reference to use it.

What about multi-dimensional arrays?

Following is the syntax of new operator for a multi-dimensional array as follows:

```
int ROW = 2;
int COL = 3;
double **pvalue  = new double* [ROW]; // Allocate memory for rows

// Now allocate memory for columns
for(int i = 0; i < COL; i++) {
   pvalue[i] = new double[COL];
}

for(int i = 0; i < ROW; i++) {
  delete[] pvalue[i];
}
delete [] pvalue;
```

In above example, we created a 2-D dynamic array, then deallocated it as well.

Program Example

Below is an example of allocating and deallocating of dynamic memory

```
#include <iostream>
using namespace std;
int main ()
{
   int i,n;
   int * p;
   cout << "Enter size of input";
   cin >> i;
   p= new int[i];

   for (n=0; n<i; n++)
   {
      cout << "Enter number: ";
      cin >> p[n];
   }
   cout << "You have entered: ";
   for (n=0; n<i; n++)
      cout << p[n] << ", ";
   delete[] p;
   return 0;
}
```

OUTPUT:

```
Enter size of input
5
```

```
Enter number: 1
Enter number: 2
Enter number: 3
Enter number: 4
Enter number: 5
You have entered: 1, 2, 3, 4, 5,
```

In above example we created a dynamic array used it to input valuesand then deallocated it from heap.

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).
Below is the stepwise algorithm for the same.

1. Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).
2. int * temp = new int[size + 5];
3. Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.
4. for (int i = 0; i < size; i++)
       temp[i] = list[i];
5. Delete the old array -- you don't need it anymore!
6. delete [] list; // this deletes the array pointed to by "list"
7. Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.
8. 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.

Summary

The main advantage of using dynamic memory allocation is preventing the wastage of memory. In static memory allocation, if we allocate 1000 memory locations as int name[1000]. While running the program only half of this may be used. The rest is unused and idle. It is a wastage of memory.So we will allocate memory on the fly to prevent this.