```c
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
// Lecture 35 - Memory Allocation in C
void Malloc() {
    //int *p = (int*)calloc(5, sizeof(int));
    int *p = (int*)malloc(5 * sizeof(int));
    // if you don't do the type cast,
    // you will see the error message 'a value
      of type "void *" cannot be used to
      initialise an entity of type 'int*'
    // Use a cstyle cast here - it should work.
    if (p == NULL) {
        printf("Failed to allocate memory\n");
        return;
    }
    *p = 5;
    printf("%d", *p);
    //free(p);
    p = NULL;
    //free(p);
}

// Lecture 36 - Dynamic Memory Allocation in C++
void New() {
    int *p = new int; // we have not specified
      any size, we only specified the type.
    // the compiler knows what size this type is.
    // so new will automatically use that size
      and allocate the memory appropriately.
    // The memory at line 24 is uninitialised
      (same as malloc!)
    *p = 6;
    delete p;

    // An advantage of new is that you can
      initialise a memory as soon as it is
      allocated
```

```
    allocated.
    // So we can initialise the memory here:
    int *p = new int(5); // memory initialised
      to 5
    // This is something that malloc and calloc
      cannot do.
    // When we use new with classes for
      allocating objects, new will also call the
      constructor for those objects, and malloc
      cannot do that.

    *p = 6; // memory reassigned to 6.
    std::cout << *p << std::endl;
    delete p; // just like with free, delete p
      will leave p as a dangling pointer, which
      will be safe after setting to null.
    p = nullptr;

    // What does new do when it fails to
      allocate memory?
    // It throws an exception - so we'll discuss
      this behaviour of 'new' in the exception
      handling lecture.
}
// Lecture 37 - how to use 'new' for Arrays and
  Strings
void NewArrays() {
    // An integer array of size 5
    int *p = new int[5];
    for (int i = 0; i < 5; ++i) {
        p[i] = i;// TAKE NOTE of the syntax.
        // arrays are still pointers to the
          first element.
        // so this is allowed!
    }
    // We could also initialise this using
      uniform initialisation:
    int *p = new int[5]{1,2,3,4,5};
```

```
56      // and we can skip the for loop above.
57
58      // Freeing the memory
59      delete[]p; // if you don't use this, this
          may not delete the entire array - it may
          only delete the FIRST ELEMENT - so that's
          why it's important to specify the empty
          subscript.
60      // in this example, if this is the last line
          in the function then there is no need to
          assign null to it.
61      // Why? the p variable is declared ON THE
          STACK, which is for local variables, and
          will be destroyed once the function goes
          out of scope.
62  }
63  void Strings() {
64      char *p = new char[4];
65      // You have to remember that whenever you
          allocate
66      // memory for the string array, you always
          have to
67      // allocate one extra byte for the null
          terminating
68      // character.
69      // This is important if you want your string
          to be null terminated, and strcpy will
          automatically append a null after it
          performs the copy into the address at p.
70      strcpy(p, "C++");  // this is deprecated in
          C++11.
71      // Use the below instead - this is the safe
          version of strcpy.
72      // The first argument is the destination,
73      // second argument is the length of string
74      // last argument is the source string.
75      strcpy_s(p, 4, "C++");
```

```
76      // Display the string here:
77      std::cout << p << std::endl;
78      delete[]p;
79  }
80
81  // Lecture 38 - 2D Arrays on the Heap using new
82  void TwoD() {
83      // If let's say we wanted to create a 2x3
          array (2 row 3 col), this will be
          represented as a contiguous 1D array, as
          follows:
84      int data[2][3] = {
85          1,2,3, // first row
86          4,5,6 // second row
87      }
88      // In the memory, it is represented as a 1D
          array: 1, 2, 3, 4, 5, 6
89      // But because of the syntax that we have
          used to create the array, the compiler
          allows us to access the individual
          elements using the row-column syntax.
90
91      // To access the element in the 1st row, 2nd
          column, then the first index represents
          the row, and the second represents the
          column.
92      int firstrowsecondcol = data[0][1];
93
94      /// Creating a 2D array on the heap
95
96      // This is slightly different.
97      // We have to represent each row as a 1D
          array.
98      int *p1 = new int[3]; // p1 represents the
          first row.
99      int *p2 = new int[3]; // p2 represents the
          second row.
100
```

```
    // to use this as a 2D array, we will have to store these pointers in another array and that'll be an array of integer pointers.

    // So pData is a pointer to a pointer to an integer, because it is an ARRAY of pointers.
    int **pData = new int *[2]; //2 here represents the number of rows.
    // Each element in this 1D array will hold a pointer to another 1D array, as follows:
    pData[0] = p1;
    pData[1] = p2;
    // Now, we will be able to access the elements of the 2D array using the row and column syntax. (Hooray)
    int row1col2 = pData[0][1];

    // Free the memory of the 2D array
    // Note that you have to free it IN THE SAME ORDER
    // that you allocated it.
    delete[]p1;// OR delete []pData[0]
    delete[]p2;// OR delete []pData[1]

    // Then free the memory of the array of pointers.
    delete [] pData;
    // the number of delete calls should match the number of new calls.
}

```