# Lesson 5 - Memory
# Runtime Allocation on the heap

## Summary

Understand about allocating and freeing memory. Understand the difference between the stack and the heap.

### New Concepts

New, Delete, the stack and the heap.

## Memory

When we declare a variable such as `a` on line 6, we are telling the compiler to create this variable in an area of memory called the “stack”. Stack variables are sometimes referred to as “local variables” because they only exist within a certain scope (more on scope later).

Variables on the stack can hold any predefined type, including pointers, so we can also declare pointers on the stack like the declaration of the pointer variable `b` on line 7.

You have already seen how you can assign a value to a variable on the stack, with the assignment operator `=`. On line 8 we assign the stack variable `a` to hold the value 100.

```cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int main () {
6    int a;
7    int* b;
8    a = 100;
```

memory.cpp

Let’s introduce the `new` keyword (see line 9). The `new` keyword allows us to allocate memory on something called the “heap” instead of the stack. The keyword `new` is provided by C++ to handle the creation of variables on the heap for us. In order to use `new` you must specify the type of variable to create so that the compiler knows how much memory to allocate on the heap. Using `new` also returns the memory address where the newly allocated memory is located, so you must assign the result of `new` to a pointer type. The type of pointer will usually match the type of memory allocated.

Now we have a pointer to an integer called `b` which points to an address located in `heap` memory. The `cout` on line 11 displays this. We say the pointer `b` exists on the stack but the memory pointed to by `b` exists on the heap.

After calling `new` the memory is initialised to zero and we can verify this by dereferencing the pointer `b` (see the `cout` on line 13). If we want to assign the newly created heap memory to a value we must dereference pointer `b` and assign it to an integer value, as shown on line 15.

```cpp
 9    b = new int;
10
11    cout << "The pointer b points to the memory address " << b
12    << " located on the heap\n";
13    cout << "The value in memory location b is " << *b << "\n";
14
15    *b = a;
```

```
16
17    cout << "The value of stack variable a is " << a << "\n";
18    cout << "The 'heap' memory pointed to by b contains " << *b << "\n";
19
20    delete b;
21    b = NULL;
22
23    return 0;
24 }
```

memory.cpp

Now the area of heap memory pointed to by `b` contains a copy of the value assigned to the stack variable `a`. Two memory locations contain the value 100, one on the stack and the other on the heap. We display this fact using the `cout` statements on lines 17 and 18.

The memory containing stack variables is automatically released when the program execution leaves their scope. Variables allocated on the heap however are not automatically released like stack variables. it is very important that the programmer releases heap memory when finished with it or else the memory cannot be reused and a "memory leak" will be created. C++ provides the keyword `delete` for this task which expects a pointer to a memory address on the heap (see line 20).

When `delete` it called on a pointer variable it is good programming practise to set the pointer to `NULL` which we do on line 21. If you subsequently try to access a pointer that has been freed with `delete` you will probably cause your program to crash.

Having reached the end of the function, the stack variables `int a` and `int* b` will be automatically released. If you had not released the heap memory previously pointed to by `b` it would have be lost for the duration of the program execution always remember to free memory allocated with `new` once you are finished with it, by calling `delete` on the pointer which points to it!

## New Feature

*Shared* pointers are an addition to C++ 11, along with *Unique* and *Weak* pointers to replace the now obsolete 'auto_ptr' type. The code snippet below shows one of the benefits of shared pointers in particular. First a `std::list` is created, comprising of shared pointers to integer types (line 29). Then 100 shared pointers are added to the list (lines 30 to 32). With ordinary pointers, each element of the list would have to be manually deleted to prevent memory leaks from occurring. With shared pointers, however, a reference count is automatically consulted when a shared pointer goes out of scope (line 33). If there are no longer any references to the memory pointed to by the shared pointer, then that memory can be automatically reclaimed and a memory leak is avoided.

```
25 #include <list>
26 #include <memory>
27 ...
28 {
29 std::list<std::shared_ptr<int>> intList;
30    for(int i = 0; i < 100; i++) {
31       intList.push_back(std::make_shared<int>(i));
32    }
33 }// no memory leaks!
```

memory.cpp

Care must be taken with shared pointers, however, to ensure that reference cycles are not introduced when two shared pointers point to each other's memory. The code below demonstrates the danger. We begin with 'structs' `A` and `B` (lines 34 and 35). A `struct` is simply a structure, like a `class`, to group variables together (more on classes later). `A` and `B` each contain `shared_ptr` types (see lines 39,40 and 45) and we have added print statements to our 'structs' so we can see when their memory is reclaimed (lines 41 and 46).

```
34 struct A;
35 struct B;
36 struct C;
37
```

```cpp
38 struct A {
39    std::shared_ptr<B> bPtr;
40    std::shared_ptr<C> cPtr;
41    ~A() { std::cout << "A is deleted" << std::endl; }
42 };
43
44 struct B {
45    std::shared_ptr<A> aPtr;
46    ~B() { std::cout << "B is deleted" << std::endl; }
47 };
48
49 struct C {
50    std::weak_ptr<A> aptr;
51    ~C() { std::cout << "C is being deleted" << std::endl; }
52 };
53
54 int main(int argc, char* argv[]) {
55    {
56       std::shared_ptr<A> ap(new A);
57       std::shared_ptr<B> bp(new B);
58
59       ap->bPtr = bp;
60       bp->aPtr = ap;
61    }// objects should be destroyed but aren't
62
63    {
64       std::shared_ptr<A> ap(new A);
65       std::shared_ptr<C> cp(new C);
66
67       ap->cPtr = cp;
68       cp->aptr = ap;
69    }// weak pointer solve the problem
70    return 0;
71 }
```

memory.cpp

We shall now create shared pointers to our ‘structs’ (inadvertently creating a circular dependency in the process). On lines 56 and 57 we create our shared pointers and on lines 59-60 we set each ‘struct’s’ pointer to point to one another. On line 61 we close the scope of objects `ap` and `bp`, but their respective memory cannot be released because of the circular dependency we have introduced. When `ap` and `bp` go out of scope, each object’s reference count is decremented from 2 to 1 and so their memory will not be released.

We can avoid circular dependencies, however, by employing a `weak_ptr`. A `weak_ptr` doesn’t use the reference count mechanism of the shared pointer. When the only references remaining are weak pointers types, then their memory can be reclaimed. We define a new `struct` type called `C` (lines 49-52) which contains a `weak_ptr` type (line 50). Now on lines 63-69 we create `struct A` and `struct C` objects. We then set each object to point to one another (lines 67-68). This time, however, we avoid the circular dependency because we now have a weak pointer pointing to a shared pointer. The `ap` object will go out of scope as normal on line 69, and the reference count used by the `shared_ptr` is subsequently decremented and reaches zero. This only leaves a `weak_ptr` whose memory can be reclaimed.

## Exercises

1. What is “Stack Overflow” and why might a Recursive method be susceptible to this kind of problem?

2. Memory Leaks are a problem associated with which type of memory? What must you do to try and avoid causing Memory Leaks in your programs?

3. Amend the code to create a `char` variable on the stack and a `string` variable on the heap.

4. Examine some of the other options provided when calling `new` and `delete`. Amend the code to create an array of 10 integers on the heap. Ensure that you release the memory you created using the correct form of `delete`.

5. Tree structures are a widely used data construct in programming (including games programming). One such tree structure is a Binary Search Tree which stores data in a manner that allows fast searching. Familiarise yourself with this technique and write your own Binary Search Tree using the code below to help you:

```cpp
struct node {
    int value;
    struct node* left;
    struct node* right;
};

struct node* root = NULL;

// implement the functions described by these headers
void insert_integer(struct node** tree, int value);
void print_tree(struct node* tree);
void terminate_tree(struct node* tree);

/**
* Main function
*/
int main() {
   // call your implemented functions here to test
   // the binary search tree
   return 0;
}
```

memory.cpp

A Tree can be considered as a collection of nodes, linked to one another via pointers. You have been given the definition of a node in the Binary Search Tree. This is in the form of a `struct`; a record for holding together related variables.

You "build" the tree recursively using the `insert_integer` function. You will need to find the correct place to insert a node to hold the new value, then create a node on the heap.

The `print_tree` function should print out the values in the tree in ascending order, and the `terminate_tree` function should reclaim all the memory used in the tree.