Name: Soham Thaker SID: 011-748-159 Email: sdthaker1@myseneca.ca

In this workshop I learnt 3 important concepts among many in the C++ language, namely, allocating and deallocation dynamic memory & difference between static and dynamic memory, what overloaded functions are and its uses and what are reference variables and its applications.

Firstly, memory is allocated in 2 ways, statically(stack) and dynamically(heap). For variables that are allocated statically, its memory gets automatically released by the compiler after the program is successfully executed. For static variables memory is allocated at the compile time and they can simply be created by writing “type identifier” and initialization is optional, however recommended to 0 or null value for any type. On the other hand, dynamically allocated variables are the variables who are originally a static variable and who gets the memory allocated at the run time, dynamically in the memory block as requested by the user, if he/she is to input the number or by the program, where the programmer tells the compiler how much memory is to be allocated to the static variable. Mostly, dynamically allocated variables are statically allocated pointer variables so that they can easily point to the memory block where the dynamic initialization is to take place. A pointer variable at the time of creation should(recommended practice) be initialized to “nullptr” value so that it does not contain any garbage values. A pointer variable is then assigned a new memory block with the help of keyword “new” where “new” returns the address of the memory assigned to that pointer or simply put, memory that the pointer will be pointing from now onwards. Its initialization for dynamic memory would be “ptr = new char;”, where ptr’s type is char and is now assigned a memory block dynamically from the heap. Dynamically allocated variables can also be arrays where the pointer simply points to the address of the first element and since the memory is allocated contiguously to the arrays, the dynamically allocated pointer has access to the whole array. To create a dynamic array we simply write, “ptr = new char[]” where [] tells the compiler that ptr is a dynamically allocated array. One important thing to note is, we as programmers must keep a track of how much memory has been allocated since there is no way in C++ to check the size of the dynamically allocated variable. Also, the “new” keyword comes with a pair of “delete” keyword. Since the dynamic memory is created at the run time, compiler will not be able to free that memory which is why we as programmers must deallocate it since if not deallocated, will cause memory leak error. We free/deallocate the memory with the help of “delete ptr” or “delete[] ptr” in the case of an array. [] are very important when deallocating the array since writing only “delete ptr” will result in deleting the first element or free the first element’s memory leaving the rest of the elements’ value inaccessible and their memory not being deallocated properly.

Let us say a dynamically allocated pointer is pointing to a block in memory and now the programmer wants it to point to another memory block, before he/she points to another memory block, he/she must delete the previous pointer memory and then point the pointer to the new memory block. Furthermore, before deallocating the memory programmers always need to make sure to take care of the unfinished business if there is any, for example,

int n;

cout << "Enter the value: “<< endl;

cin >> n;

int\* ptr = nullptr;

ptr = new int[n];

for(int i = 0; i < n; i++) {

ptr[i] = i;

}

if(ptr != nullptr) {

//Take care of unfinished business here if there is any, means copy the data or store the data properly where the programmer wants to and then deallocate the memory.

}

delete [] ptr;

The above model is just an example and puts emphasis only on the data to be stored properly, that the dynamically allocated variable is pointing to, if there is any. If the data is dealt with properly or there is no need to take care of the data, then memory can be deallocated without any caution.

Also, when dealing with dynamically allocated arrays, programmers need make sure, to stay within the range of the allocated memory. Going over will cause a serious error and can cause the program to crash. For example, going over 5 or above in for loop, in the above example, will cause the program to fatally crash. Furthermore “delete [] ptr”, checks if the ptr is pointing to a nullptr (assuming that the pointer at the time of creation was initialized to a nullptr, which is always recommended), if it is, it will not delete the data because there is nothing to delete. However, if ptr is pointing to an actual address then it will do its work and go ahead and delete the data plus the memory allocated to it.

Secondly, function overloading is very important concept in C++ which basically means overloading a function with same identifier but with different components in function’s signature, where signature includes the functions’ identifier, parameter types & the order of the parameters. The compiler automatically decides which function is to be called at the time of function call, by analysing the types of arguments passed into the function. Compilers generate multiple definitions according to the existing functions(a.k.a mangling) whose identifiers are same and their parameters so the linker can bind the function calls according to the types of the arguments passed into the function.

Thirdly, reference variables are a new concept which was not present in C language which acts and behaves almost like the pointers but is an alias to the variable it is assigned to. Its only limitation is that it can only point to an unmodifiable address meaning that once it becomes an alias to a variable, it cannot change its memory address it is pointing to making it an unmodifiable address meaning it cannot become an alias to another variable in its entire lifetime. Also, a reference variable can only become an alias of an already existing variable which should already be initialized.

Reference variables are a lot safer due to its strict property, like unmodifiable address, since if a function takes an object as its primary parameters and wants to change the value of its alias but does not want it to point to any other address, other than its alias’, it can do so easily and the changes in its value will be directly reflected into the original variable. Whereas a pointer in its lifetime can point to infinite addresses, where a programmer must be very careful since he/she might lose very important data if the pointer starts points to another address and the data at previous address has not been stored properly.

Reference variables are represented with the help of “&” or “ampersand” right next to its type and don’t need the address of the variable, just the identifier of the variable is fine, unlike, using the “address of” in “pass by address” where ampersand or the address of the variable was necessary for the pointer to point to the address of a variable, for example,

int no = 3;

int& num = no;

where num is now an alias of no and can change the value of no to anything, within its type. Reference variables are also easy to read since the next time if it were to assign its value to any other variable it will use the identifier itself, for example,

int n;

n = num;