1 Pointers and Dynamic Memory

Dynamic memory allows you to build programs with data that is not of fixed size at compile time. Most nontrivial programs make use of dynamic memory in some form.

1.1 The Stack and the Heap

Memory in your C++ application is divided into two parts — the *stack* and the *heap*.

1.1.1 The Stack

The stack can be visualized as a deck of cards. The current top card is the current scope of the program, usually the function that is currently being executed. Variables declared in the current function will take up memory in its own stack frame. Any parameters passed from one function to the next are **copied** from the old stack frame to the new stack frame.

Stack frames are nice because they provide an isolated memory workspace for each function. When the function is done running, the stack frame goes away, and all the variables declade within the function no longer take up memory. Variables that are stack-allocated do not need to be deallocated by the programmer; it happens automatically.

1.1.2 The heap

The *heap* is an area of memory that is completely independent of the current function or stack frame. Variables on the heap exist even when the function in which they were created has completed. The heap is less structured than the stack, and its sort of like just a pile of bits. Your program can add new bits to the pile or modify them at any time. You have to make sure that you deallocate any memory that you allocated on the heap, this does not happen automatically, unless you use smart pointers.

1.2 Working with Pointers

You can put anything on the heap by explicity allocating memory for it. To put an integer on the heap, you need to allocate memory for it, but first you need to declare a *pointer*.

```
int* myIntegerPointer;
```

The * indicates that the variable you are declaring refers/points to some integer memory. Think of the pointer as an arrow that points at the dynamically allocated heap memory. The pointer we just declared does not yet point to anything specific because we haven't assigned it to anything; it is an uninitialized variable. Uninitialized variables and pointers should be avoided at all time, especially because pointers, when uninitialized, point to some random place in memory. Working with such pointers will most likely make your program crash. Initialize them to null pointer (nullptr) if you don't want to allocate memory right away.

```
int* myIntegerPointer = nullptr;
```

You can use the new operator to allocate the memory

```
myIntegerPointer = new int;
```

In this case, the pointer points to the address of just a single integer value. To access the value, you need to *dereference* the pointer. Dereferencing is just following the pointer's arrow to the actual value in the heap. To set the value of a newly allocated heap integer, you would use the following

```
*myIntegerPointer = 8;
```

This is not the same as setting myIntegerPointer to the value 8. You are not changing the memory that it points to. If you were to reassign the pointer value, it would point to the memory address 8. Which is probably random garbage that will make your program crash.

After you finished with your dynamically allocated memory, you need to deallocate the memory using the delete operator. To prevent the pointer from being used after having deallocated the memory it points to, its recommended to set your pointer to nullptr:

```
delet myIntegerPointer;
myIntegerPointer = nullptr;
```

Pointers don't necessarily point to heap memory. You can declare a pointer that points to a variable in the stack, even another pointer. To get a pointer to a variable, you use the & "address of" operator

```
int i = 8;
int* myIntegerPointer = &i;
```

C++ has a special syntax for dealing with pointers to structures. Technically if you have a pointer to a structure, you can access its fields by first dereferencing with *, then using the normal . syntax

```
Employee* anEmployee = getEmployee();
cout << (*anEmployee).salay << endl;
```

This syntax sucks, the operator lets you perform both the dereference and the field access in one step. The following code is equivalent to the preceding code

```
Employee* anEmployee = getEmployee();
cout << anEmployee->salary << endl;
```

Pointers to stack variables are often used in C to allow functions to modify variables in other stack frames, essentially *passing by reference*. By dereferencing the pointer, the function can change the memeory that represents the variable even though that variable isn't in the current stack frame. This is less common in C++, because C++ has a better mechanism, called *references*, which is covered later.

1.3 Dynamically Allocated Arrays

The heap can also be used to dynamically allocate arrays. You use the new[] operator to allocate memory for an array

```
int arraySize = 8;
int* myVariableSizedArray = new int[arraySize];
```

This allocates memory for enough integers to satisfy the arraySize variable. Now that the memory has been allocated, you can work with myVariableSizedArray as though it were a regular stack-based array.

```
myVariableSizedArray[3] = 2;
```

When your code is done with the array, it should remove it from the heap so that other variables can use the memory. In C++, you use the delete[] operator

```
delete[] myVariableSizedArray;
```

The brackets after delete indicate that you are deleting an array.

To prevent *memory leaks*, every call to new should be paired with a call to delete, and every call to new[] should be called with a call to delete[]. Pairing a new[] with a delete also causes a memory leak.

1.4 Null Pointer Constant

Before C++11, the constant 0 was used to define either the number 0 or a null pointer. This causes some problems. Take the following example

```
void func(char* str) {cout << "char* version" << endl;}
void func(int i) {cout << "int version" << endl;}
int main() {
      func(NULL);
      return 0;
}
```

Since NULL is not a pointer, but identical to the intger 0, the integer function of func() is called. This problem is solved with the introduction of a real null pointer constant, nullptr. The following code calls the char* function.

```
func(nullptr);
```

1.5 Smart Pointers

To avoid common memory problems, please use *smart pointers* instead of normal C-style pointers. Smart pointers automatically deallocate memory when the smart pointer object goes out of scope. There are three smart pointer types in C++

1. std::unique_ptr
2. std::shared_ptr
3. std::weak_ptr

1.5.1 Unique Pointer

The unique pointer is analogous to an ordinary pointer, except that it will automatically free the memory or resource when the unique_ptr goes out of scope or is deleted. It also has the sole ownership of the object pointed to. One advantage of the unique_ptr is that it simplifies coding where storage must be freed when an exceptional situation occurs.

```
auto anEmployee = std::make_unique<Employee>(); // C++14
std::unique_ptr<Employee> anEmployee(new Employee); // Before C++14
```

1.5.2 Shared Pointer

shared_ptr allows for distributed ”ownership” of the data. Each time a shared_ptr is assigned, a reference count is incremented indicating there is one morer ”owner” of the data. You cannot store an array in a shared_ptr. Use std::make¡¿() to create a shared_ptr.