You can find code from the last live lecture on Canvas Modules (Kennslustund04.zip). Students that want to start at the very basics should start with that code and simply experiment. Try out the reference (**&**) and dereference (*) operators. See how they work on value and pointer variables, what can be assigned to what and what the results are of printing different things. Experimentation is the only way to get properly used to using pointers and references.

Make a simple ***struct*** (or ***class*** with only public variables, no operations), **Stats**
Have it include tree variables, **games** (int), **home_runs** (int) and **average_points** (double).
If you want to make a proper class, make a constructor with parameters and override the << operator for printing it to a stream (like cout) and stuff like that.

Make the class **StatList**.
This class should implement a *dynamic array* of **Stats**. Make the *private* variable **Stats *array**.
In the constructor either *initialize* the **array** or set it to **nullptr** (or **NULL**).
Have the class keep track of the capacity (how many instances the array can hold) and size (how many instances have been added).
Implement public operations like **append(Stats)** and **Stats get_at(int index)**.
If you need the extra practice, implement **insert(value, index)**, shifting items in the list to make space, **remove(index)** and such operations.
Implement the destructor **StatList::~StatList()** (use virtual correctly in the declaration). The destructor must make sure the memory allocated using new is cleaned up using delete.
Now make some operator overrides, like << to print the list and maybe override the + operator to append to the list, or to concatenate two lists.

Can you change the list so that it doesn't contain an array of **Stats** but a dynamic array of **Stats*** so that the array is only keeping room for the pointer, until a stat is actually added, then it uses new to make an actual instance of **Stats** on the heap and stores the value of the pointer in the array? You may need to use a double pointer (**Stats** **array_of_pointers).
Now you also have to make sure to ***delete*** whenever a Stats instance is removed and also ***delete*** all the instances of **Stats** in the destructor.

Special practice exercise on the next page...

Here's one that I will not give a solution for until after **PA3** *is handed in, but it's worth the time to implement it and hang on to. It may not be the fastest way to do this (some students may later wish to optimize) but getting it to work properly is key to building bit strings in character arrays where the data may need to be read on a tighter level than in whole bytes*.

Make the class BitString (or whatever you want to call it).
Here is a suggestion for operations, but this class will not be handed in, but you may want to use it as part of a later assignment, so make it in a way that you understand and feel works for the way you may use it.

BitString::BitString()
You may want to have parameters, initial number of bytes in the array, etc. You may also wish to make a copy constructor and maybe both a default constructor and one with parameters, or one with parameters with default values, whichever works for your implementation.

BitString::~BitString()
Destructor to clean up the byte/bit data.

void BitString::setNextBit(bool value)
Sets the next bit in the bitstring.
use &, |, << and >> to actually set the next bit. Don't use a whole byte to represent a single bit.
It can help to keep track of a current_bit or bitstring_position variable or something like that.

bool BitString::getNextBit()
Gets the next bit in the bitstring.
use &, |, << and >> to actually get the value of the next bit. Then return a boolean value representing 1 or 0.
It can help to keep track of a current_bit or bitstring_position variable or something like that.

void BitString::goToStart()
Set the get/set position to the start of the string. Can be helpful when you start to read from the bitstring after having added to it. Students can of course have separate set_pos and get_pos variables if that works for them.

char* getBytes() (or unsigned char if that's what you're using)
Returns a pointer to the actual array of bytes holding the data. This can be used to write it to a file.

int getLength()
Return the length of the bit string (in bits)

bool isAtEnd()
Returns True if the last bit has been read (current_position is at the end). This can be helpful when reading back from the bit string.