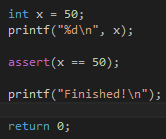


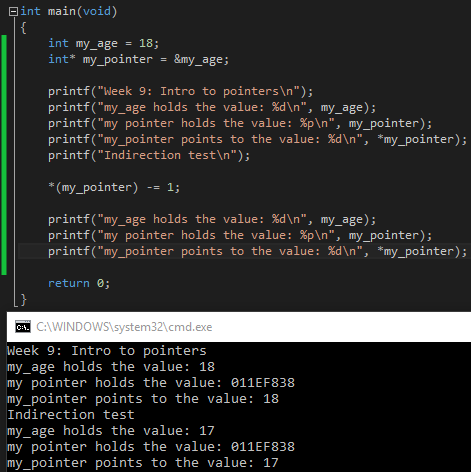
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Entry ID:** | **Date:** | **Day:** | **Start Time:** | **Duration:** | **Session Type:** | **Lab/Lecture Number:** | **Location:** |
| Week 9 | | | | | | | |
| 1 | 8/05/2017 | Monday | 9:00am | 1 hour | Lecture | 25 | WG403 |
| 2 | 8/05/2017 | Monday | 10:00am | 3 hours | Lab |  | WT201 |
| 3 | 11/05/2017 | Thursday | 8:00am | 1 hour | Lecture | 26 | WG403 |
| 4 | 12/05/2017 | Friday | 11:00am | 1 hour | Lecture | 27 | WG403 |
| Week 10 | | | | | | | |
| 5 | 15/05/2017 | Monday | 9:00am | 1 hour | Lecture | 28 | WG403 |
| 6 | 15/05/2017 | Monday | 10:00am | 3 hours | Lab |  | WT201 |
| 7 | 18/05/2017 | Thursday | 8:00am | 1 hour | Lecture | 29 | WG403 |
| 8 | 19/05/2017 | Friday | 11:00am | 1 hour | Lecture | 30 | WG403 |
| Week 11 | | | | | | | |
| 9 | 22/05/2017 | Monday | 9:00am | 1 hour | Lecture | 31 | WG403 |
| 10 | 22/05/2017 | Monday | 10:00am | 3 hours | Lab |  | WT201 |
| 11 | 25/05/2017 | Thursday | 8:00am | 1 hour | Lecture | 32 | WG403 |
| 12 | 26/05/2017 | Friday | 11:00am | 1 hour | Lecture | 33 | WG403 |
| Week 12 | | | | | | | |
| 13 | 29/05/2017 | Monday | 9:00am | 1 hour | Lecture | 34 | WG403 |
| 14 | 29/05/2017 | Monday | 10:00am | 3 hours | Lab |  | WT201 |
| 15 | 31/05/2017 | Wednesday | 2:00am | 9 hours | Self – Study |  | Home |
| 16 | 1/06/2017 | Thursday | 8:00am | 1 hour | Lecture | 35 | WG403 |
| 17 | 1/06/2017 | Thursday | 6:00am | 8 hours | Self – Study |  | Home |
| 18 | 2/06/2017 | Friday | 11:00am | 1 hour | Lecture | 36 | WG403 |

1. Lecture 25 – 8/05/2017
   1. **Overview:**
      1. Assert
         1. The assert function is a sanity check. It is used by the coder and remains only in debugger and never in a release build. Because of this, it is important to note that an assert should never change any data.
         2. An assert can be a pre-conditional post-conditional. The data needs to be a certain state or in a certain state to be passed in and out.
      2. Build Targets:
         1. Debugger mode vs release mode
      3. The pre-processor:
         1. Conditional Compilation
      4. Coding standards:
         1. Function comments.
   2. **Lessons learnt: Assert**:
      1. An assert is a sanity check. It is a line of code to check the state of a program or variable. They’re really nice debuggers as you can see where things are at certain or specific points in the program. It checks the states of anything you choose as a programmer.
      2. To use them, you must use the #include <assert.h> (assert header file). The function takes in one parameter which is the data or information piece that you want to check the state of.
      3. 0 is equal to false and a non-0 is equal to true.
      4. Asserts check very, VERY specific things. You must understand exactly what the customer wants before using sanity checks.
      5. High level English questions that an assert might ask are: Is the variable negative at this moment? Has a certain number of things been counted? Is a pointer NULL at this moment? Should it be should it have data?
      6. So, if a condition is true, it continues the code. If the condition is false, then it breaks the code and tells you exactly what went wrong with the state of the variable you’re asserting.



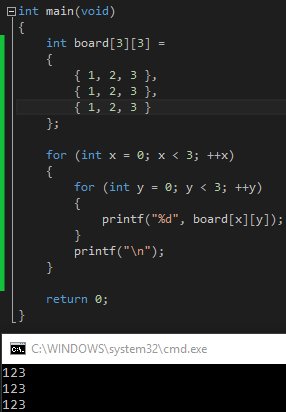
* + 1. The code above checks that x is still equal to 50. If for some reason, the RAM changed or the memory location changed, and the value was no longer 50, then the assert would trigger and the program would stop and break at that point. The assert when failed, prints x == 50, file[*name*], [*line number*].
    2. Assert is also actually just a macro being defined inside the assert header file using #define.
  1. **Lessons Learnt: Design by contract:**
     1. There are pre and post conditions using assert. They check if a function has correct input or output.
     2. An example of a pre-condition is like a check before the calling of a function that asserts user input values to make sure the user has inputted the right things. A post-condition example is like after a function is run, an assert can check to see if the correct type of data has been concluded before moving on with the program.
     3. Assert is an easy way to debug and only really fixes very obvious errors or at least lets people know about obvious, but maybe unseen errors.
     4. The proper syntax for asserts is to use them everywhere to check everything from memory leaks, to incorrect values to incorrect timing and counting
     5. It is important to note and stress that asserts check the state of code. They should never be used to change states of variables and they should never save things. Because whilst the code might work for debugger mode, the code will be stripped and will not work for release mode.
  2. **Lessons Learnt: Build targets:**
     1. The two different modes talked about above are called debugger target and release target. The release target strips out different things. It runs faster and it gets rid of asserts most importantly.
     2. You can easily switch from the drop-down box in visual studio.

1. Lab – 8/05/2017
   1. **Overview:**
      1. This lab we went over pointers, passing by reference, structures and passing them by reference, memory sharing with pointers. Passing arrays and printing elements from them and simple asserting and simple target building.
   2. **Exercises Done:**
      1. The first exercise was called introduction to pointers and just included the use of simplistic pointers. We had to create a variable and make multiple pointers to it and pointers to pointers and use printf with deferencing to deference the pointers.

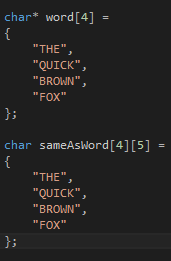


* + 1. The second exercise was called *pass by value vs pass by reference, with structures*. In this exercise, the goal was to declare a structure, and then save it into a pointer and run a function that has the structure being passed by reference. And another function that is being passed the structure by value. The value structure just creates a copy of the structure and you have to set the data inside the structure from what results from the function. Whilst the pass by reference function allows the function to directly edit the original structure due to being able to access its address.
    2. The third exercise was called pass by reference with pointers. The exercises goal was to create three variables for three different die and get the values being passed into one function by reference to get random dice rolls as well as print out the location of the dice in memory.
    3. The fourth exercise was called super hero structure by reference. The goal for this exercise was to declare a structure that represents a comic book super hero with members for his name, alter ego and the year they first appear. Inside the main function, the exercise asks that you define and set the members of the structure and then pass by reference into a function to print out the details.

1. Lecture 26 – 11/05/2017
   1. **Overview:**
      1. Multidimensional arrays – dimensional string arrays
         1. 2D arrays, 3D arrays, C-string arrays
         2. C-string arrays and enumerations.
      2. The parameters of main
      3. Type casting
         1. Implicit vs explicit conversion
         2. Cast operator.
      4. Void pointers
         1. Type casting with pointers.
   2. **Lessons Learnt:**
      1. A string by itself is an array, but a group of strings is a multidimensional string array. The more dimensions, the more depth you get when controlling what is inside an array. Integer 3D arrays are different than strings. Integer multidimensional arrays can be used to hold matrices or matrixes to hold 3D information.



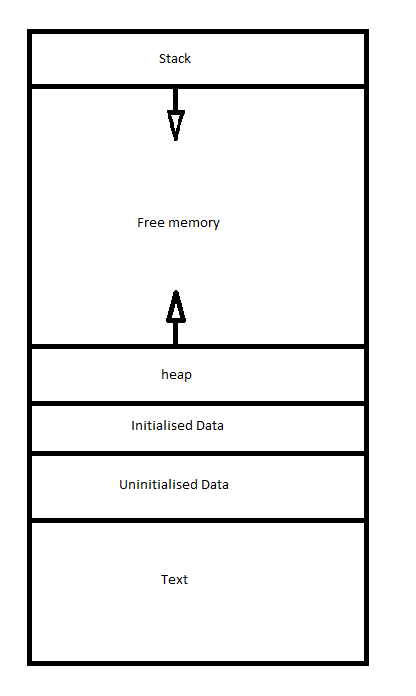
* + 1. Using the same logic with multidimensional arrays, we can create an array of c-strings. This means each element in the array will store the address of a char. And at that location, there may be a c-string which is a NULL character terminated string.



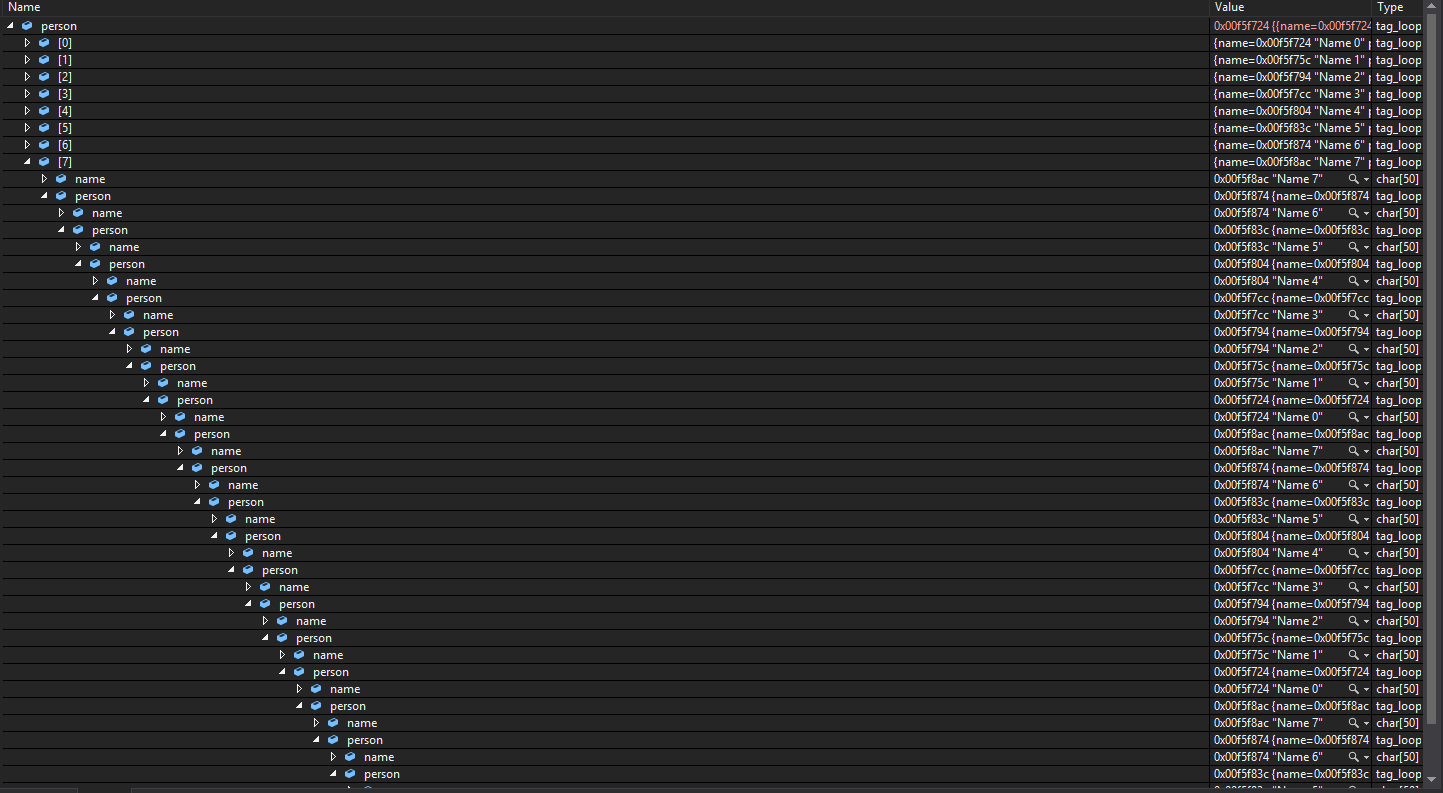
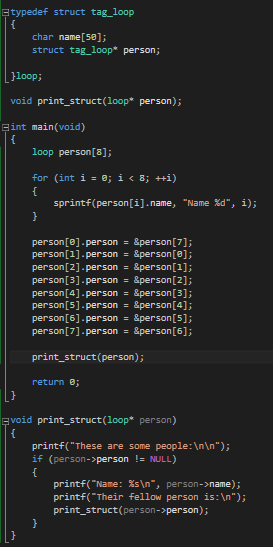
* + 1. The code above acts the exact same in both instances. The top one is only readable and not writable while the bottom version is writable. The array is stored in what’s called the: Stack. And the pointer points to somewhere in memory. The star pointer in the first version is a pointer to an array of pointers pointing at single char’s. You could also write the top one as char\*\* word = but that becomes quite deep with many pointers.
  1. **Lessons Learnt Enums with pointers and arrays:**
     1. We order enums the same as a char array and look at the string array at index of an enum. To note again, an enum is a type of variable that has infinite numbers that can be represented as a name. If you want the number 10 to represented by the word ten. Then an enum could do that. With this, we can use enums to find specific indexes. Say you wanted to set into an array, at index of player504004. You could use enums to instead save into player at index [*NAME*].
  2. **Lessons Learnt: More main inputs and outputs**
     1. To send and receive data from main, we need to add in more attributes to main instead of just a void. To get these, you must use int argc and char\* argv []

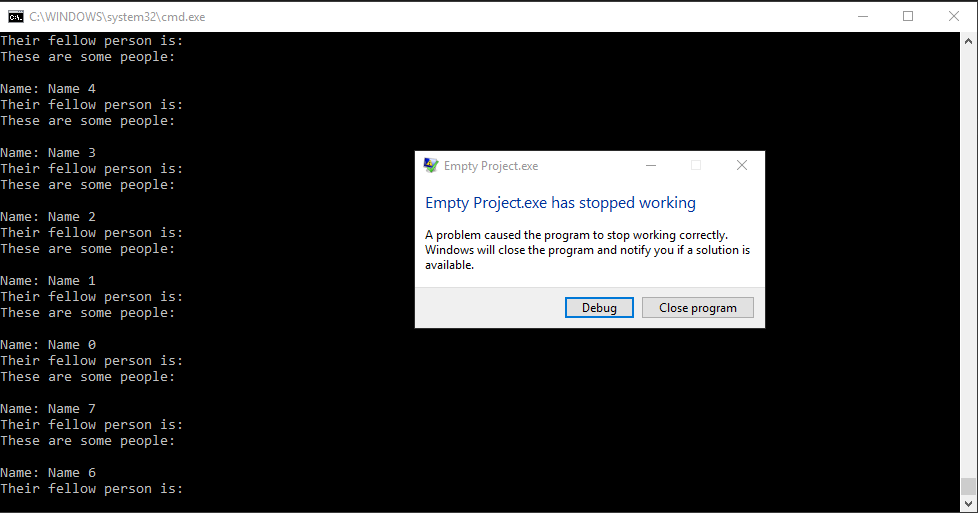


* + 1. Argc represents how many parameters are inside the parameter location. This value will always be 1 as there is the name of the file. Anything else added like ---help into the start-up of the program will be sent into the second argument. The argc part is the actual text array that holds all the arguments that can be taken in and used as commands and programs.
  1. **Lessons Learnt: Type conversion:**
     1. Implicit type conversion is performed by the compiler automatically. Like when you convert a float to an int. The compiler implicitly changes the float value into an integer and truncates the value of the original floating-point number.
     2. Explicit type conversion is performed by the programmer through using the cast operator.
     3. The syntax is = (*type to cast to like int*) *Expression*.
  2. **Lessons Learnt: Void Pointers:**
     1. A void pointer is a generic old address that can’t be dereferenced. They’re used for general purpose pointers. They have no data type associated with void\* and you can store the address of any type of variable within them.
     2. To Typecast pointers, you have to use the syntax: \*((char\*) pointer)); This is like saying have this pointer, probably a void pointer. First dereference it and then cast it to a char pointer.

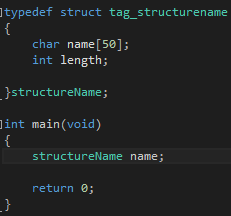
1. Lecture 27 – 12/05/2017
   1. **Overview:**
      1. Dynamic memory
         1. Malloc and free. Malloc is memory allocation and free acts as a memory deallocation.
      2. The heap.
      3. Memory leaks
      4. Structure on the heap
         1. Malloc and structures. If you forget to free the memory, it becomes a memory leaker. Too much gets on the stack or heap.
      5. Returning heap structures
      6. Recursive Composition
         1. Aggregation: Storing a member inside a structure of its own structure.
   2. **Lessons Learnt: Memory allocation**
      1. So far, we have learnt about and can predict memory space such as an int array with size 50. However, what if we wanted to make an array that has the size of a variable. Normal arrays do not allow this and result in error code C2057.
      2. To do this, we can use malloc. This is blocks of bytes allocated via a call to memory.
      3. The malloc must be explicitly de-allocated using free after the program no longer needs memory. This is essentially setting it back to NULL. Otherwise the program starts to have memory leaks and if you lose track of an allocation, it is unable to be deallocated (going out of scope without freeing memory).
      4. To use mallocs, you must #include <stdlib.h> header file:



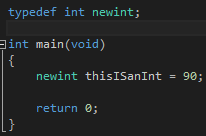
* + 1. The above code allocates 8 pieces of memory the size of an int, so a total of 32 bytes and saves the address of it into info as an int pointer. This means that info is now essentially an array 8 elements wide that can hold integers.
    2. After all is said and done, you MUST free the allocated pointer.
    3. The code to the left frees info from the memory and just as a precaution, it also sets info to NULL. Blocks of allocated memory can last forever if it’s not free because it’s in the heap so you need to deallocate.
    4. If you have an allocated memory block in a loop, you may forget the area where you stored the allocation but the memory keeps growing and it causes a memory leak. Each loop, you’re essentially breaking the previous loops scope and losing all the mallocs. Thus, you’re not able to free them and the memory grows.
    5. When the program finishes, the compiler cancels and frees all allocations.
  1. **Lessons Learnt: Dynamic memory allocation**
     1. You can create arrays with the memory allocation. You can set the size of a memory allocation to the size of an input variable to get dynamic size.
     2. Mallocs are extremely powerful. You can set a pointer with a name to a memory allocation to the size of whatever you want. Times the data to get the number of ints. 5 Integers for example.
  2. **Lessons Learnt: Memory leaks**
     1. In windows, you can check to see if your program is leaking. Using #include <crtdbg.h> header file allows you to access the code: \_CrtSetDbgFlag (\_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF); which allows you to know if you’ve leaked memory and how much or if you didn’t leak any memory.
  3. **Lessons Learnt: Recursion:**
     1. The use of structures inside structures is extremely useful. To do so, you need to use pointers but it essentially allows you to have links to other structures within one structure, like having parents of one structure and having parts of those and so on.
     2. The code bellow is an example of structure aggregation and recursion used together. Because the structure has a member called person which is also a structure, it can hold the structural data of another person. As the code bellow shows, because it’s a loop, where the person member is holding the data for the previous person around and around in a loop forever, the recursion printing from the function is going to crash the program because the stack will fill the available memory.

The code is trying to run the function until it hits a NULL person being stored, but because the member for person is storing around in a loop in the structures, I now have an endless structure. The recursion causes the stack to overflow and result in this bug:

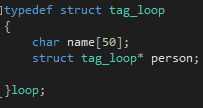
1. Lecture 28 – 15/05/2017
   1. **Overview:**
      1. The typedef keyword
         1. Creating a type aliases
         2. Typedef with structures and enumerations
      2. Testing/debugging
         1. Stubs
         2. Instrumentation: Tracing
         3. Build targets: Debug memory layout
      3. Creating new solutions and projects
      4. Compiler warnings: Warning levels.
   2. **Lessons Learnt: Typedef Keyword:**
      1. Alias: To avoid having to use the struct keyword when declaring a structure inside the main or any other function as a variable, the typedef keyword can be used when the structure is declared.
      2. Typedef struct *[tagname]* is the usual syntax for making a typedef of a struct. You put typedef in front of the struct and then the name of the variable at this section should be a tag name, or something to differentiate it from the actual name of the type that is going to be declared after the members of the structure are declared:



* + 1. Above, it is shown that you use the typedef keyword, then the struct keyword, then the *[tagname]* of the structure. This needs to be anything OTHER than the actual type name. After the members are declared, the actual type name is labelled after the scope braces and before the NULL terminator. In the case above, the type name is structureName. Then inside the main, or any other function, to declare the structure, you just have to type the type name to create the variable.
    2. It is important to note, typedef can be used for literally anything. If you want to type def absolutely anything, you can. Typedef is similar to using #define as you can define any type you want as a custom name.



* + 1. A lot of game developers like to typedef different types to shorten them and make them easier to use. It is common to typedef an unsigned char as a type: byte, or u8 or s8. It makes the code able to work the way you want it to work. You can use u8 and then type u8 instead of unsigned char as the type.
    2. These are like hotkeys for different variable types.
    3. Enums and structures have great use for typedef since instead of typing out the long version of struct [*structNameOfType]* [*structurename*] when declaring a structure you can just type [*typedefNameOfStructure*] to declare it. Typedefs are incredibly useful and a brilliant way of shortening types.
    4. It is important to note, if you make a structure, and you want a member to be aggregated, so it’s, its own structure within itself, because the *typename* has not been declared yet, being that they are declared AFTER the members, a structure member needs to be declared normally using the full type declaring version.

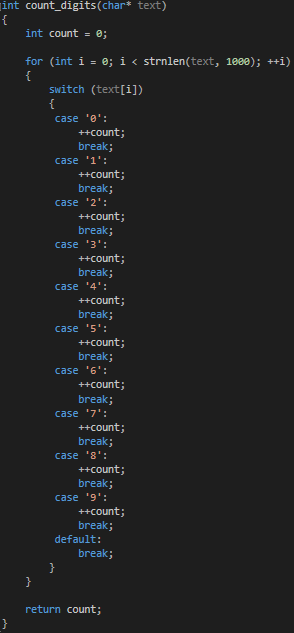


* + 1. In the above code, the type “loop” has not yet been declared until the scope ends. This means that the member inside called person, which is a structure type cannot use the type “loop”. It has to use the type “struct tag\_loop”
  1. **Debugging and testing:**
     1. Static testing: Verification.
        1. Code review is also known as peer review. Code review is when someone, either a peer or yourself, does an examination of the programs source code looking for mistakes.
        2. A walkthrough is when someone goes through the code and shows an example usage of the program. Interested parties can then critique the program during the walkthrough
     2. Dynamic Testing: Validation
        1. Testing of modules. Each part is tested. If it works, you test the next section, if it doesn’t work, then you stop testing and you work on fixing that module.
        2. You are checking the individual functionality of the program.
        3. There is a testing format or syntax called stubs. A stub is a piece of code that is a temporary stand in for future pieces of code. Usually you would comment //to-do: [stub]
        4. This allows you to build the code to test first. You build bits and pieces of code piece at a time so as to not have one large broken block of code. You can build the bits of code without worrying about some functions you want to run later. You test different bits at different times.
        5. The Instrumentation of this type of testing is that it throws errors purposefully and then logs errors as well. So if you have one section of code that calculates gravity in a game for example, then you can run that, test all the variables and such, and then when it gets to the end of its run time, it throws an error and jumps out of the code.
     3. Debugging: Memory patterns
        1. In the debug build, the compiler injects certain values so you can check certain things.
        2. There are certain hex codes that the compiler puts inside values and elements to allow the programmer to see what he’s doing.
        3. 0xCCCCCCCC = A clear variable being stored to the stack
        4. 0xCDCDCDCD = A clear variable/pointer being stored to the heap
        5. 0xFEEEFEEE = Is free memory. You can’t write here.
        6. 0xBAADFOOD = Is allocated memory
        7. 0xDEADBEEF = Deallocated memory.
        8. 0xFDFDFDFD = The end of an array. This is very important as It allows you to see where an array can no longer work from. Do not go past this point
  2. **Lessons Learnt: Creating a new visual studio project:**
     1. You go: file > new > project > (inside the wizard) > visual C++ > win32 > win32 console application > set location > set the name of the project > click next > tick empty project > turn SDL off.
     2. This project will have default of level 3 warnings, which is quite low in terms of warnings.
     3. To add a source file, you go: Right click on project > add > new item > set the source file as .c instead of .cpp
     4. It is important to restate, that this file will have relatively low warning levels. 3 is the default warning level. You can change it by going into the properties of the project and setting it higher. Setting it to 4 or 5 will help, as it will stop us from creating obvious errors. You can also set the warnings to show as errors so that the program cannot run until you have gotten everything perfect.

1. Lab – 15/05/2017
   1. **Code Done:**
      1. The first exercise was called passing an array. This exercise asked to explain in high level English what it does. So, the code first creates an array with a size of two, able to hold three elements inside the array. It then sets the first element to the number 10 (this will be the hours). Then it sets the second element to 5 (this will be the minutes). Then lastly, it will set the last element with 0, to leave it empty. (this will be the seconds). It then calls a function with this array and this data.

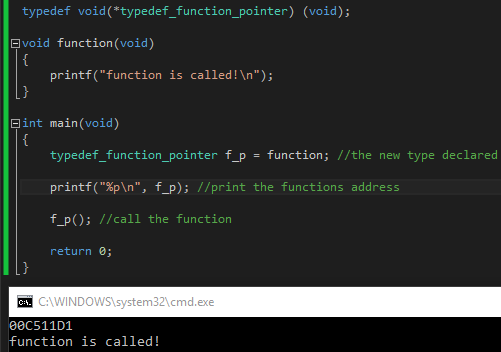
This function, once run, takes the array in with first element being hours, the second being minutes and the last being seconds, and it prints out the hours, mins and seconds in order of normal time syntax. It then jumps back to the main body of the code, and finishes. The different between an array and a structure is that an array has essentially, a defined number of unlabelled elements. Anything can be inside the elements but because they’re not labelled, a new programmer would not be able to immediately tell what the elements where. A structure can have well set out, plainly labelled members that can all be accessed separately and can easily be set into. A structure in this codes case would be much better suited as the three members could just be called hours, mins and seconds. Plus, making a structure would allow the programmer to save on memory because he could use different types of data per member. The hours would only need a short or a char, and the mins and seconds could also be in short or char mode. Having a structure would avoid the need to explanation comments as the members would be called what they are holding.

* + 1. The second program is called square array fill. This program uses multidimensional arrays and asks to declare a 5 by 5 2D array. The exercise then asks to fill each member with a different number such that it makes a sort of pattern. It then asks to print it out in a pattern using nested loops. This is quite simple as you make a 2 deep nested loop and print out the element at index of x and y (x and y being the two integers declared with each level of the loop). And at the end of the outer loop (x), you print a new line. This creates the pattern.
    2. The third exercise is called using 2D char array. This exercise asks to take in an array by passing by reference and then count how many digits are inside the text. I did not know an easier way to do this, so I made a switch case for the text array and then if the individual char being looked at in the loop was either 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9, it counted, otherwise, it would break from the switch and look at the next char value.

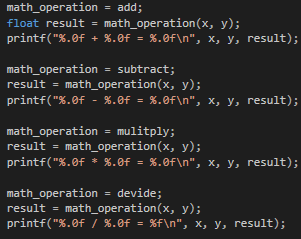
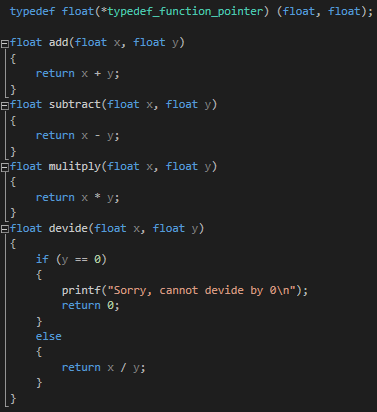


* + 1. The fourth program, and the last one I got up to was called detecting main’s arguments. The exercise asks to write a program that processes the arguments passed into main from the command line. Unfortunately, after trying later at home, the same code does not seem to work.

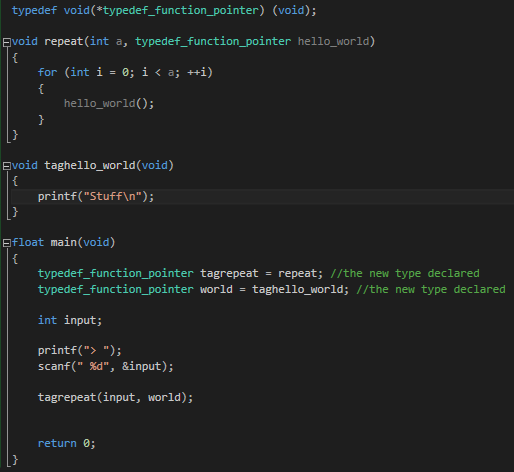
1. Lecture 29 – 18/05/2017
   1. **Overview:**
      1. Function Pointers
         1. Storing a function’s address.
         2. Calling a function via pointer
         3. Typedef and function pointers
      2. Call backs
         1. Comparators
         2. Stdlib.h and qsort function (quick sorting).
      3. Functional Abstraction.
   2. **Lessons Learnt: Pointers**
      1. Pointers are a variable that stores an address. You can use -> operator to access the members from a function, beyond the scope with an address. You can use dot operator to access members directly. An & symbol is to find the address of something, and the \* operator is to dereference.
      2. If you make a function and then use printf and print the &[function] or the function with the call braces, it will print the ADDRESS of the function, because without the call braces, it is not calling the function but treating the function as if it’s a type and it’s finding its address.
      3. Printing main or &main will resolve at the same address where main is held.
      4. Printf and %printf printing are the same address. They are functions themselves and are thusly held at certain places in the memory. These memory locations are quite far from main and any other functions.
      5. You can find where a function is stored in memory by right clicking on it, and then go to definition. Alternatively, you can right click on the project or source and click go to disassembly. This allows you to view the assembly code.
   3. **Lessons Learnt: Function pointers**
      1. Using the syntax void\* pointer\_function = (void\*) function; is essentially making a void pointer called pointer function to a function that is being type cast to a void pointer. You can’t call the pointer as a function because it lost all of the function context when it was typecast into a void pointer.
      2. Using the syntax void (\*pointer\_function)(void) = function; is essentially like saying: This is a function pointer that takes nothing and returns nothing. So, you can call this. This syntax allows a pointer to hold context of the function. To call it, you just must use syntax: pointer\_function() like a normal function.
      3. The difference between using void (\*pointer\_function)(void) and using void \*pointer\_function(void) is that the first one is a void function with no arguments, and context, and the second one is a variable declaration which returns a void pointer. This has no context.
      4. *(Return Type* (\*) *arguments)*
      5. The other syntax, and slightly easier syntax of making a function pointer is pointer2\_function = (void (\*) (void)) pointer\_function; which is like saying cast pointer\_function and treat it as if it’s a function pointer.
      6. Calling it uses the syntax (\*(void (\*) ())) pointer\_function or (void (\*) ()) pointer\_function. Obviously, the syntax for all of these is extremely messy and very inefficient.
      7. There’s a much easier way we can restore sanity using typedef.
      8. Typedef void (\* *typedef\_void\_function\_pointer*) (void); This is essentially saying type define a function pointer that takes in a void, and returns a void. This is done before the function is declared and above the main.
      9. There is now a new type called typedef\_void\_function\_pointer. And this type can be declared in the main, with a name. The name would be function\_pointer or f\_p or whatever, and then it would equal the function itself as shown in the code bellow:



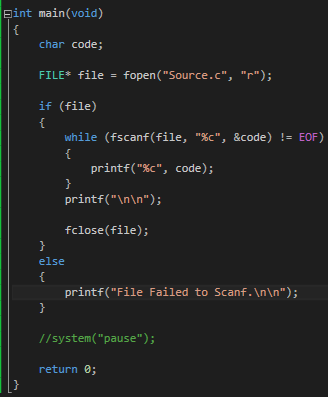
* + 1. Function pointers are extremely useful now because you can set the new pointer type to any function you want and then call the pointer with the variables you need like the code bellow which takes in two numbers and runs different functions on them based on what you set the pointer to:



* 1. **Lessons Learnt: Call backs**
     1. Call-backs are used when a program needs to keep working until it is interrupted by (and can respond to) some external event. This could be either a user or an environmental interruption. Basically: “Do not call us, we’ll call you”
     2. This is like saying: When you discover something, call a function.
  2. **Lessons Learnt: From call backs, come Qsort**
     1. Quick sort needs four arguments.
        1. The address of an array. This is where the quicksort will start. It then needs the size of the array, then the size of each element. This can be done easily by going sizeof(*element type*) and then it needs the function pointer to compare units.
        2. To compare units, the function itself needs to be written. It Is quite simple and takes in two void pointers (the function does not know what it is taking in yet) and then it needs to return the different between these two variables. To do this, you can type cast the two void pointers as int pointers and subtract one from the other and return the result. This gives the difference between the two and the quick sort can use that to sort the values inside the array.
        3. If the values are equal, the function described above returns 0. If the difference is positive, it’s larger (returns a non-0) and if it’s negative, the return value is smaller (returns a non-0). This type of data sorting is called a call-back.
        4. To actually cast the two void pointers, you have to cast the variable as an int (or other type) pointer and then dereference it. (\*(int\*)*variable*).
     2. **Lessons Learnt: Function abstraction**
        1. They are templates for action. Functions are for hiding code and making things modular.
        2. Basically, if you had a function for example, that you wanted to use to print something in a loop. You could have two functions, one that makes a loop that calls a function. This function takes in a variable for how many loops it is to do, and the other thing it takes in is a function pointer. This function pointer is the function it’s calling. So basically, this function is calling another function abstractly. The other function, is just printing a word, and in main, it is being made into a function pointer. So, when the first function is called, it is being called with a variable and a function pointer to the second function. ***Abstraction***

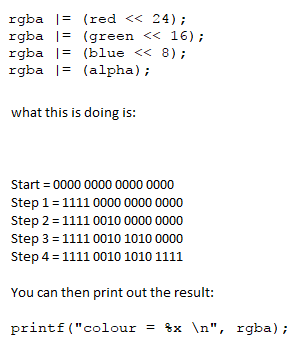


1. Lecture 30 – 19/05/2017
   1. **Overview**
      1. File input and output
         1. Text files, binary files
      2. The hex editor.
   2. **Lessons Learnt:**
      1. So far, all our programs output goes to the screen. However, now we can learn about programming so that information and programs output goes to a file.
      2. All data we store, and process, in our programs is saved in the computer’s RAM. This data disappears when the program exits. Input/output streams from files. The file I/O in C is abstracted into operators that occur on streams of bytes.
   3. **Lessons Learnt: With file input and output, there is a new file type…**
      1. Type: FILE.
      2. To use file, I/O, you have to use firstly, fopen(). This function takes in two parameters. A char string filename and a char string mode. It then returns a file. The file name is the file of type FILE. So at the start of a program, it is a good idea to declare a FILE type with the name file for example. The fopen will return a FILE pointer so you are able to take this value. If it is NULL it means the file failed to open the file. The mode can come in all different types described below.
      3. Just like malloc and having to free the allocated memory, you also must close the file you have opened. The function fclose() is how you do it. This takes in a file stream. The file stream is the same one you would have opened using fopen.
      4. The last function you can use is fprintf. This function is very similar to printf and sprintf. It will print a formatted data string to a stream. This function prints to a file you opened.
      5. The modes talked about in fopen are how the function opens a file. Below are all the listed modes. They must be passed into the fopen function in string format.
         1. r = Opens the file for reading only. The file must exist previously.
         2. w = empty file for writing. This overwrites same name files. This creates a new file
         3. a = Opens for writing at the end of a file (appends or adds to the file). This creates a file if it does not exist.
         4. rt = read/write. The file needs to exist to write and read into.
         5. wt = opens a new file to read/write.
         6. at = Reads and writes into an appending file. So, it adds into the file.
      6. You can also add to the modes by putting for example rb. These secondary modes are used for different formatting.
         1. B = Binary mode. This skips all white space and treats everything like it is an actual printable character. So, a new line will actually print as \n.
         2. T = text mode. This is automatically added on the end of all normal modes, but if you want to specify it, you can add it in. This mode makes sure to print everything in normal text mode.
      7. Important note, EOF is an integer value given off from fopen as one of the possible return values. This value indicates an error.
   4. **Lessons Learnt: More functions**
      1. fscanf(FILE\* stream, char\* format); This function scans like a normal scanf but scans from the file.
      2. fflush(FILE\* stream); This function flushes a stream. This works similar to fclose.
   5. **Lessons Learnt: Printing and Reading**
      1. An example of file printing is first setting up a file stream by writing FILE\* p\_file; then writing p\_file = fopen(“example.text”, “w”); This is opening a file called example.text in mode writing and saving the return value to a FILE type variable called p\_file.
      2. An example of file reading is first setting a buffer array for text to be saved into. Then setting up the file type again, then writing fscanf(p\_file, “%79s”, first\_line); which will trim off white space and scan in text up until a new line.
      3. Another way to scan in so that it is scanning individual char symbols so you don’t have to worry about new lines or other annoying white space. Like the code bellow:

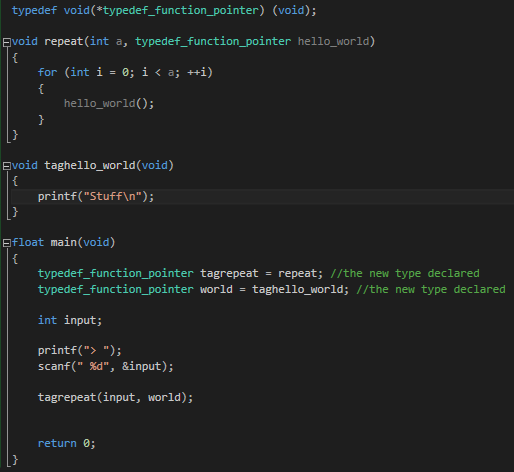


* + 1. The code above creates a FILE stream and then opens the file *source.c* which is its own source code in mode *read*. Then it checks if the file properly opened before continuing with the program. If it completed and opened, then the loop created inside the if will check if the fscanf has reached the end of the file, otherwise, it will print any and all symbols and character it encounters. This includes new lines, white space and other white space symbols.
    2. However, not all files are text files. Because some files are not spannable, you must use fwrite and fread.
    3. The syntax for fwrite is fwrite(void\* ptr, size\_t, size\_t, nmemb, FILE\* stream) So, the void pointer is a variable, the first size of is the size of the type of data. So, you can use sizeof(int) for example. The second size variable is the size of the variable itself. So, if it’s an array, then it could be 5 elements. And the file is obviously the stream.
    4. The syntax for fread is fread(void\* ptr, size\_t, size\_t, nmemb, FILE\* stream) So, the void pointer is a variable. The first size is the size of the type of data again, the second size is the size of the variable and the file is the file stream.
    5. So, if you had a structure being type defined as cat, then fwrite would be written like so: fwrite(&write, sizeof(int), 1, p\_file);
  1. **Other functions:**
     1. ftell = tells the current file position of a stream.
     2. fseek = set the file position to a given offset.
     3. fgetpos = returns current file position.
     4. fsetpos = sets the file position to a position.

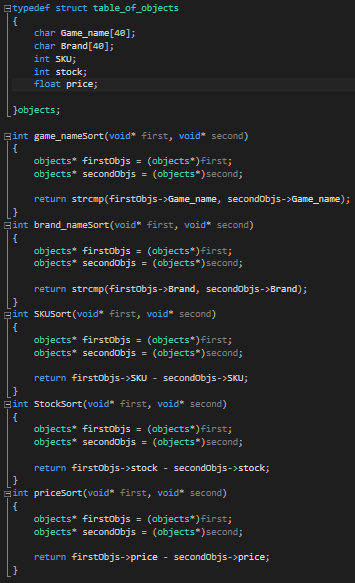
1. Lecture 31 – 22/05/2017
   1. **Overview:**
      1. Bitwise operator:
         1. Bitwise NOT, bitwise AND, bitwise OR, bitwise XOR
      2. Bit shifting
         1. Shifting left, shifting right
      3. Bitwise assignment operators
      4. Bit flags and bit masking
      5. Passing bit flags into functions.
   2. **Lessons Learnt: Bit operators**
      1. AND = & (for this, both bits must be on to equal to 1)
      2. OR = | (for this, either or both bits must be on to equal to 1)
      3. XOR = ^ (for this, either alone, not both together must be on, to equal 1)
      4. NOT = ~ (Flips o to 1 and 1 to 0).
      5. Examples of this is like setting an int x to 26. Then setting integer y to ~x (NOT x) so y is going to equal -27 due to bit noting.
      6. Setting x to 26, setting y to 15 and then making z = x & y is going to equal 10 because the bits of 26 are 00011010 and the bits for 15 are 00001111 so when you AND them together, it becomes 00001010 which is 10.
      7. Setting x to 26, y to 15 and z to x|y is going to make z = 31 because the bits of 26 are 00011010 and the bits for 15 are 00001111 so when you OR them together, it becomes 00011111 which is 31.
      8. And finally, setting x to 26 again, y to 15 and z to x ^ y is going to equal 17 because the bits of 26 are 00011010 and the bits for 15 are 00001111 so when you XOR them together, it becomes 00010001 which is 17.
   3. **Lessons Learnt: Bit shifting**
      1. >> is right shifting
      2. << is left shifting
      3. Bit shifting is simply moving the bits up and down the bit stream by certain amounts.
      4. So, setting x to 10, and y to x >> 1 is going to make y equal to 5 because 10 in binary is 00001010 and when you left shift by 1 (or divide by 2) it becomes 00000101 or 5.
   4. **Lessons Learnt: Compounds**
      1. It is important to note, that you can use compound version of all of these. So, like you can go += to easily set variables. You can also go &=, |=, ^=, <<=, and >>=
   5. **Lessons Learnt:**
      1. You can basically combine different states of data into one variable. Bit masking allows us to detect which bits are set in a variable. This is useful when multiple sets of data are in one variable.
      2. This is used massively everywhere when setting colours in any programming at the basic level.
      3. You can set char red = 0xf, set char green as 0x2, char blue as 0xA and char alpha as 0xF. What this is, is a darkish red, with a little green, a dark blue, and full alpha. You can then use bit masking and bit combinations to combine all of these values into one variable which can be an int rgba.



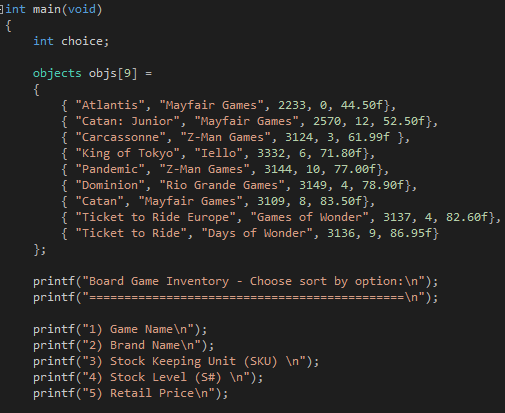
1. Lab – 22/05/2017
   1. **Code done:**
      1. The first exercise was called Simple function pointers. This exercise asked to work with function abstraction and have the main call a function with a variable and a function pointer and then the called function would use the passed in function pointer to call another function:



* + 1. The second exercise was called sort products with qsort and took such a long time, that this was only where I got up to. It asked to have an array of structures, and then to have a print out that used quick sort in 4 different ways chosen by the user, to sort and then print out a table of all the members inside all the elements of the structure array.

The code to the right first sets up a structure with members: Game\_name as a char pointer, Brand as a char pointer, SKU as an integer, stock as an integer and price as a floating number.

Below is then all the functions for the sorting forms. Each function first makes a structure pointer and sets it to a type casted version of the first and second void pointers being sent to the function. The game\_nameSort then does a string compare of the first structure array element game\_name, and then the second structure array element game\_name and returns it. This is to sort by the game name. The second function does a string compare the same way as the first function, but for the brand name. The third function simply subtracts the second structure array element SKU unit from the first. This is an integer, so all you need to do is subtract the two integers. The fourth function does the same as the third, but for the stock number, and the last function does the same as the last two, but for the floating variable called price.

The next piece of code is inside the main. This is the initialiser stage. This code simply sets the structure array elements. Each and every one of them in one large initiation.

The second section of code here simply prints out the title bar for the program, and then prints out a small list of the sorting types that the user can pick.

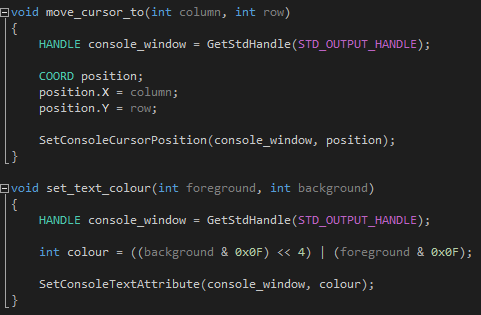
The last bit of code below, has a switch case for the choice that user decided on, and then runs a qsort function. The function takes in the entire array of structures, it takes in the size of the entire array (9), it takes in the size of the type (structure type, so objects here), and last it takes in the function pointer that it’s going to run.

Lastly, it prints out the final wholly sorted box of contents of the games in each element in the array of structures. It has a small loop to simplify the code and shrink it by printing out the list of actual numbers with the structure at element[*index of loop*].*member*.

The printing of the variables also uses specific formatting to get exactly the right (dynamic) size, and left aligned printout. Below is the final output.

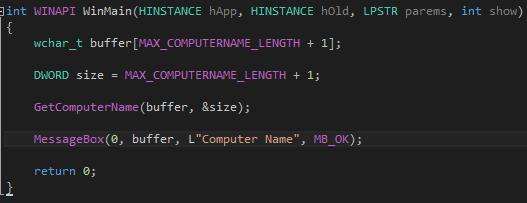


1. Lecture 32 – 25/05/2017
   1. **Overview:**
      1. Windows programming
         1. Windows handles
         2. Console Functionality
         3. Win32 project
         4. Win32 message boxes
      2. Microsoft dev network (MSDN)
         1. Programmer documentation.
   2. **Lessons Learnt:**
      1. P1colour.h and p1colour.lib are examples of things that only work in windows.
   3. **Lessons Learnt: Windows Handles**
      1. Windows uses “handles” to refer to components of windows.
      2. HANDLE console\_window = GetStdHandle(STD\_OUTPUT\_HANDLE); This code gives me the standard output handle and saves into a pointer (kind of) which is type HANDLE.
      3. HANDLE = This is a windows type (a void pointer)
      4. GetStdHandle = this is a windows.h function that returns the handle to a standard device, such as standard input, standard output or standard error.
      5. From p1colour, you got move\_crusor\_to and set\_text\_colour well that function was defined inside p1colour. Broken down, it is quite a complex, windows only function.

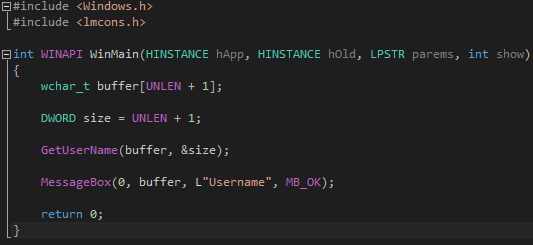


* 1. **Lessons Learnt: Proper w32 project**
     1. The entry point of a win32 project is not main, it is WinMain.
     2. To create a project, you click > new project > C ++ > win32 > win32 project (not console application) > turn off SDL checks and make it an empty project and create it > add source file.
     3. The windows API is massive though, there are a lot of new types, and many new structures and other types.
     4. The entry point for a win32 project is int WINAPI WinMain(); and it takes in a lot of new types of data.
     5. WinMain = The function implements the code required to run an exe. This is the windows entry point.
     6. WINAPI = The calling convention for system functions. Different platforms have different ways of calling functions.
     7. HINSTANCE = handle to an instance. This is the base address of the module in memory.]
     8. LPSTR = A pointer to a NULL terminated string of 8 – bit ASCII characters. This is for more info and command lines. Similar to argv
     9. Int nCmdShow = How the OS shows your program.
  2. **Lessons Learnt: Message Box**
     1. A message box syntax is like so: MessageBox(NULL, L”Hello people”, L”Hello!”, MB\_OK); The NULL is the handle to the owner of the window. NULL means the desktop is the owner. NULL means the root of everything. The middle string is Unicode string literal. The text to display inside the message box. The L is needed as this says that the text is going to be a wide text type or Unicode. The second string is the Unicode (remember the L) string literal for the title of the message box. The MB\_OK is the text box style. There is a lot of different types, MB\_OK just creates a message box with an OK button.
  3. **Lessons Learnt: Referencing Programmer documentation**
     1. Microsoft dev network or MSDN is a ***massive*** network and documentation of all of windows API. There is information on every single windows type and function and so on.

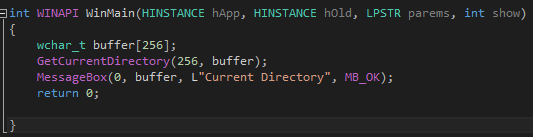
1. Lecture 33 – 26/05/2017
   1. **Overview:**
      1. Windows API
         1. System Functions
         2. The windows Class
         3. Registering a window class
         4. Creating a window
         5. The message loops
         6. The window procedure
         7. Message Handling
         8. Controls
   2. **Lessons Learnt: Functionality categories:**
      1. Base services
      2. Advanced services
      3. Graphics device interface
      4. User interface
      5. Common dialog box library
      6. Common control library
      7. Windows shell
      8. Network services.
   3. **Lessons Learnt: Windows API: Strings**
      1. The type char stores a single byte. This is useful for storing the value of ASCII characters. To make a char string array, you use char *name*[]; and char\* p\_str; to make character arrays.
      2. In windows main, there is a different type. The type wchar\_t stores wide characters. This is useful for storing the value of Unicode characters. wchar\_t *wname*[]; and wchar\_t\* p\_wstr;
   4. **Lessons Learnt: Windows API: System functions:**
   5. Get the screen size
      1. Get the screens size. To get the screen size, you must use the windows API function: GetSystemMetrics() which takes in one of two different types: SM\_CXSCREEN and SM\_CYSCREEN.
      2. To get the width value, you run the function and pass in the SM\_CXSCREEN and to get the height, you set a variable and run the function whilst passing in SM\_CYSCREEN. This function returns an int so it can be set into variables and then printed to show the user the size of his/her screen.
   6. Lock the work station
      1. To lock the workstation, you must use the windows API function: LockWorkStation(void); which locks the workstation’s display and protects it from unauthorized use. This function is run without any pass in or any output.
   7. Get last error
      1. To get the last error, and return it you must use the windows API function: GetLastError(void). This function takes in nothing but returns a DROWD data type. A DWORD type is an unsigned long.
      2. If many windows system functions fail, they set the last-error code. The get last error function returns a number. This number represents the type of error.
   8. Get the computer’s name
      1. To get the computers name, you must use the windows API function: GetComputerName(LPTSTR lpBuffer, LPDWORD lpnSize);
      2. This function passes back a BOOL type and takes in two variables of new types. The lpBuffer is a pointer to a buffer that receives the computer name or the cluster virtual server name. The buffer size should be large enough to contain MAX\_COMPUTERNAME\_LENGTH + 1 characters.
      3. The second parameter, lpnSize; On input, this specifies the size of the buffer, in TCHAR’s. On output, the number of TCHAR’s copied to the destination buffer, not including the terminating NULL character.



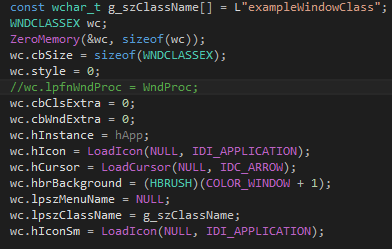
* + 1. The code above sets a wide char array to the MAX\_COMPUTERNAME\_LENGTH + 1 and then sets the DWORD called size to the same thing. Then it runs the get computer name function with the pointer to the buffer array and the pointer of the size variable.
  1. Get user name function
     1. To get the user name of the system running currently, you must use the function GetUserName(LPTSTR lpBuffer, LPDORD lpnSize);
     2. This function takes in two pointers and returns a BOOL type variable. The first parameter is the lpBuffer. This is a pointer to the buffer to receive the user’s logon name. If this buffer is not large enough to contain the user name, the function fails. A buffer size of (UNLEN + 1) characters will hold the maximum length user name including the terminating NULL character. To use UNLESS, you have to use #include lmcons.h
     3. The second parameter is lpnSize. On input, this variable specifies the size of the lpBuffer buffer, in TCHAR’s. On output, the variable receives the number of TCHAR’s copied to the buffer, including the terminating NULL character.



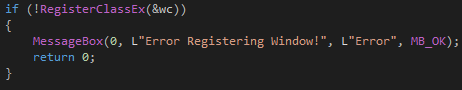
* 1. Getting current directory function
     1. To get the current directory, you have to use the windows API function: GetCurrentDirectory(DWORD nBufferLength, LPSTR lpBuffer);
     2. This function takes in two variables and passes back one variable. The first parameter called nBufferLength is the length of the buffer for the current directory string, in TCHAR’s. The buffer length must include room for terminating NULL characters.
     3. The second parameter is called lpBuffer. This is a pointer to the buffer that receives the current directory string. This null-terminated string specifies the absolute path to the current directory.
     4. And finally, the return value. If the function succeeds, the return value specifies the number of characters that are written to the buffer, not including NULL. If the function fails, the return value is 0. The return type is a DWORD.



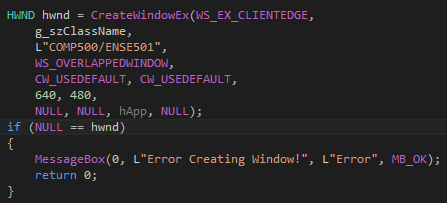
* 1. Setting current directory.
     1. To set the current directory you have to use the windows API function: SetCurrentDirectory(LPCTSTR lpPathName);
     2. This function takes in one variable and passes back one variable. The first parameter is the lpPathName. This is the path to the new current directory and this parameter may specify a relative path or a full path.
     3. The return value, if the function succeeds, the return value is non-zero. But if the function fails, the return value is 0.
  2. Zero memory creating
     1. The function called ZeroMemory() fills a block of memory with zeros. The function is written like so: ZeroMemory(PVOID Destination, SIZE\_T Length);
     2. The function takes in two variables and doesn’t return anything. The first parameter is the destination. This is a pointer to the starting address of the block of memory to fill with zeros.
     3. The second parameter is called length. This is the size of the block of memory to fill with zeros, in bytes. So, if you had one integer value, you could use sizeof(int) here, but if you have a wide char array with 256 elements. Then you would need to go sizeof(wchar\_t) \* 256;
  3. Get local time function
     1. To get the local time on the computer, and thus, in your local area so like New Zealand time, then you have to use the GetLocalTime() function. You have to make a structure with the type being SYSTEMTIME. This is a pre-type defined structure type that has members inside it. Then you run the get local time function passing in the new structure as an address (for example: &systemtime).
     2. You can then print out the members which are wDay, wMonth, wYear, wHour, wMinute, and wSecond.
     3. You can use a slightly different function called GetSystemTime() to get a universal time.
  4. **Lessons Learnt: Creating a Windows Application**
     1. Two important functions: The windows entry point for a GUI application is still: int WINAPI WinMain(HINSTANCE hApp, HINSTANCE hPreviousApp, LPSTR params, int nCmdShow);
     2. And the windows procedure. This is a function that receives messages for the window.
     3. LRESULT CALLBACK WindowProc(HWND hwnd, UNIT uMsg, WPARAM wParam, LPARAM lParam);
     4. The windows class. To create a window, the new type of window class must be registered with the windows OS. There a number of predefined window classes already registered. These can include buttons, static text, etc….
     5. To register a new type of window, we have to populate a WNDCLASSEX structure. Then call the RegisterClassEx() function. This can be the first thing done in the WinMain function.



* + 1. Once the WNDCLASSEX structure has been populated, we call RegisterClassEx();



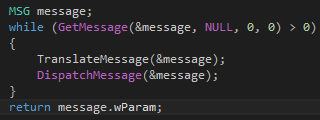
* + 1. Next, we must call CreateWindowEx() to create the window. The WINDCLASSEX member, wc.lpszClassName, uniquely identifies the window class to use.



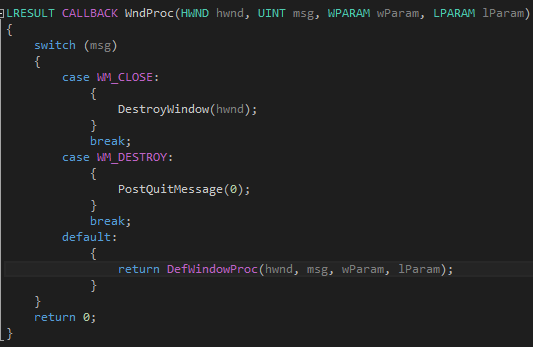
* + 1. The null handle means the window failed to create.
    2. After Both successful creations, we can call BOOL ShowWindow(HWND hWnd, int nCmdShow);
       1. The parameters are firstly, hWnd which is a handle to a window. Secondly, nCmdShow which controls how the window is to be shown.
    3. The second function we run is BOOL UpdateWindow(HWND hWnd);
       1. This function takes in one parameter called hWnd which is the handle to the window. This function causes the client area of the window specified to be updated by sending a WM\_PAINT.



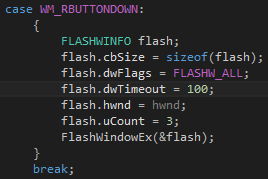
* + 1. Finally, we must process the event messages for the window.
       1. MSG is a structure containing message information from a thread’s message queue. GetMessage() retrieves a message from the calling thread’s message queue, then the function dispatches incoming sent messages until a posted message is available for retrieval.
       2. The message loop iterates until the program quits



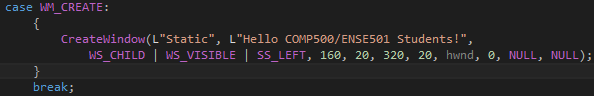
* + 1. Finally, we must process the event messages for the window.
       1. TranslateMessages(). This function translates virtual key messages into character messages.
       2. While DispatchMessage() dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.
    2. Finally, we need to define the message handling function. The window procedure. This window procedure call-backs function processes the messages it receives. Two new variables being passed into this function: wParam and 1Param contain additional information about messages. What they contain depends on the type of message.



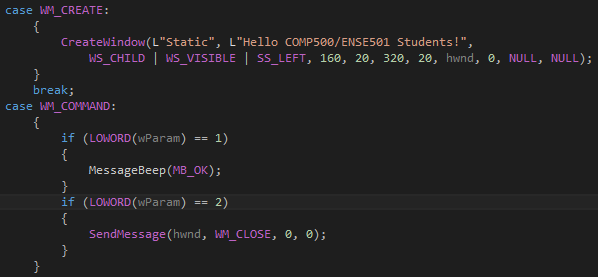
* + 1. The destroy function destroys the specified window. The function sends WM\_DESTROY and WM\_NCDESTROY messages to the window to deactivate it and remove the keyboard focus from it.
    2. The post quit message indicates to the system that a thread has made a request to terminate or quit. It is typically used to respond to a WM\_DESTROY message.
    3. In the default, the DefWindowProc function calls the default window procedure to provide default processing for any windows messages that an application does not process. This function ensures that every message is processed. DefWindowProc is called with the same parameters received by the window procedure.
    4. At this point, the window is working. To actually interact with it, you have to do a lot of message handling.
  1. **Lessons Learnt:** **Message handling**
     1. In the WndProc function, additional cases can be added to handle different events.
        1. The msg value of WM\_LBUTTONDOWN represents the click of the left mouse button inside the window.
        2. You13. can make a case for this and then make a message box in response.
     2. Many other cases can be used. WM\_KEYDOWN checks for any key at all that has been pressed down. Then there’s WM\_RBUTTONDOWN which is the right mouse button. You can also run a flash command:



* 1. **Lessons Learnt: Adding controls to a window**
     1. This is generally done in the WM\_CREATE message handler.
     2. The static control: The WM\_CREATE message is sent upon window creation. It can be used to setup the window’s contents. Controls are also known as widgets. Here, CreateWindow is used to make a static control. Static controls are used to display text and graphics. They cannot be interacted with. They have no selection, and no keyboard focus.

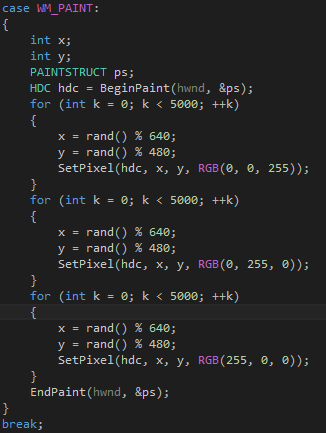


* + 1. Receiving button click messages uses the case WM\_COMMAND. It is a message handler and is sent when the user selects a control. The low word of the wparam contains the menu identifier.

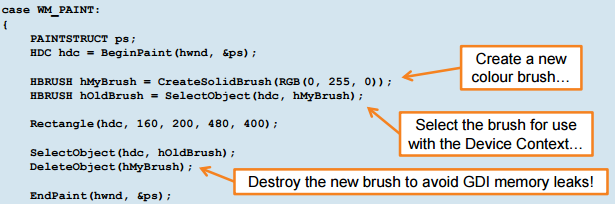


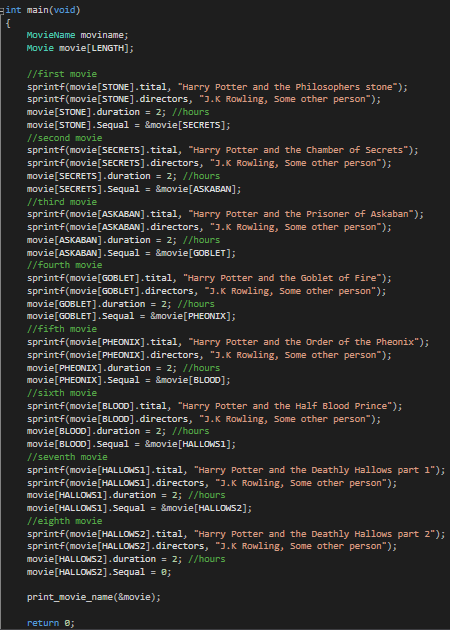
* + 1. There is an absolute metric, INSANE amount of windows API functionality. However, there is too much to cover within this journal.

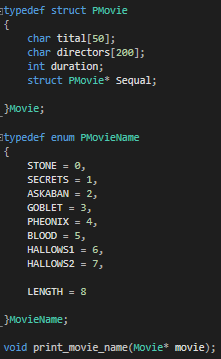
1. Lecture 34 – 29/05/2017
   1. **Overview:**
      1. Windows GDI drawing
         1. WM\_PAINT
         2. Pixels
         3. Lines
         4. Rectangles
         5. Text
         6. Pens
         7. Brushes
   2. **Lessons Learnt: Painting with Bob Ross**
      1. The function to start painting is BeginPain() and to end this, you have to go EndPaint(). This is similar to malloc and free again.
      2. This creates a PAINTSTRUCT.
      3. To set a single pixel of colour, you go SetPixel(hdc, x, y, RGB(0, 0, 255)); This will create a pixel at x and y with the colour of blue.
      4. Of course, to even start this, you need to make a message handler. The message for painting is WM\_PAINT. With this you can do things like this:

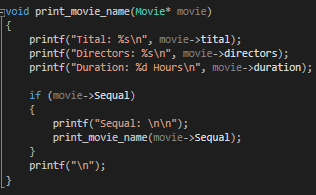


* + 1. This will draw 5000 randomly positioned blue dots, green dots and red dots.
    2. To draw a line, you use MoveToEx, and LineTo. You can also draw a rectangle using Rectangle(). Then there’s also TextOut which takes in an x-starting position, y starting position, the windows handler, the wide string and the length of the string. You can also create custom pens and brushes with certain colours and certain types.

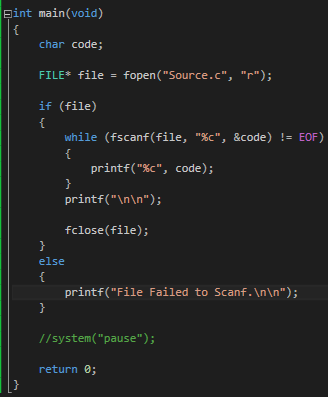


1. Lab – 29/05/2017
   1. **Overview:**
      1. The lab on this day was all about studying and practising with specific things that you might have felt weak with. In this lab, I went back over the structures within structures and recursion as well as simple message boxes as well as simple file IO
   2. **Code Done:**
      1. The first program I worked on was re-doing the practical test 3 last question as I did not get time to finish it in the actual exam. I wanted to go back over it and make sure I understood it.

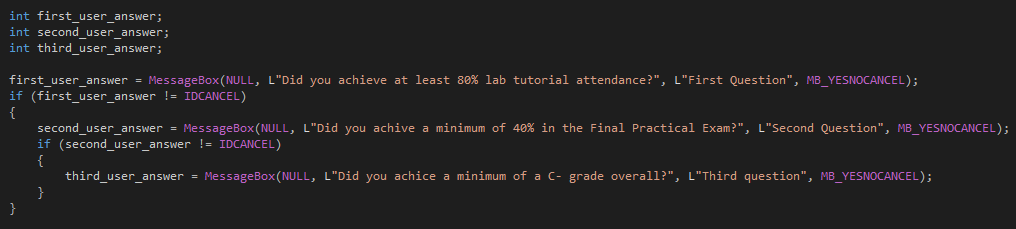
The first section of code here on the left just type defines two different types of data. One is a structure that is self-referential and creates a member called sequel which is a structure. The second type defined type is an enum that holds purely aesthetic values for the movie indexes for the soon to be declared structure array.

The second piece of code on the right first declares the two new types, making the structure a structure array. It then sets into each element and each member of the entire structure array and sets all the prequels to the next movie in the list. This means that all the elements in the structure are linked via self-referential structure. The last piece of code directly above is a simple print recursion function that takes in a structure pointer by reference and then prints out the structure array element then it’s sequel via recursion by passing in the sequel as a member/structure pointer into the print function again.

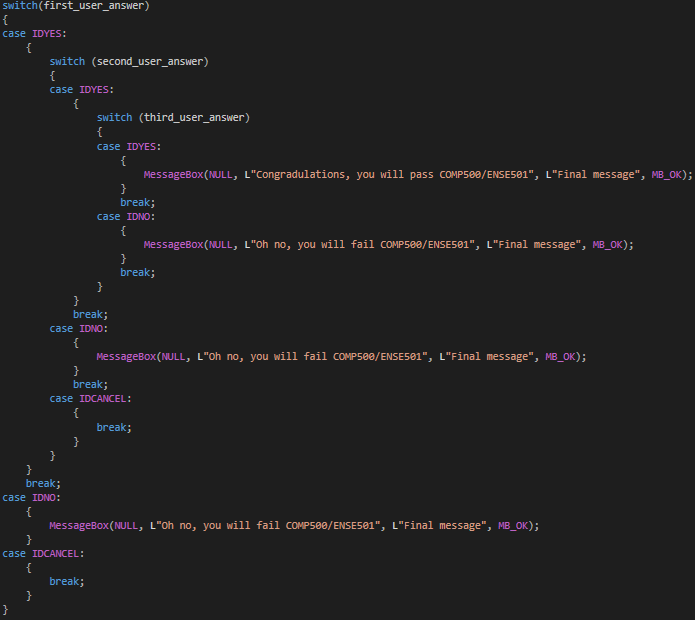
* + 1. The second code I worked one was a simple file IO I wanted to make to make sure I understood how it worked.



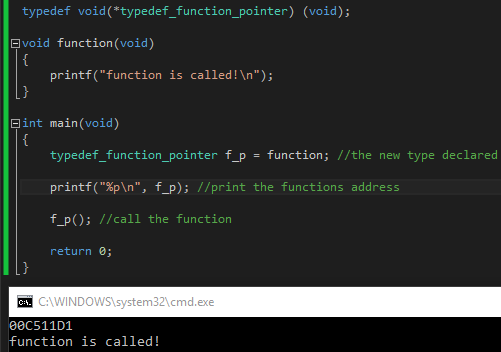
* + 1. The code above works quite well, and is modular to my source files inside all my projects. I created a release build and moved it in and out of the different source code files and got them all to print out the code to the console. This code unfortunately, does not work for some reason at home. This perfectly working, bug free code, doesn’t seem to work at my home visual studio.
    2. The third and last program I made in the lab was working on simple message box questions and message box message handling.



* + 1. The code above first sets three integers to the first, second and third user answer. It then creates a message box with the mode yesnocancel. It first tests before anything else, if you have clicked cancel on any of the boxes one by one. If you have, then it breaks the whole program and finishes. If it doesn’t, it goes through all the questions one by one and saves an answer to the first, second and third user answer
    2. The code bellow is the full check for what scores you got. It first checks if the case for the first user answer is IDYES (The message handler for a yes click), if it is a no, then it tells you, you have instantly failed. If it is yes however, then it checks the next user answer. If that is no, then you fail, if it is yes, then it checks the final user answer. If you clicked no, then you fail, and if you clicked yes, you passed. The good thing about this type of testing, is that it doesn’t waste on time since if you fail any of the message boxes, then it instantly fails you, it doesn’t waste time checking the other questions.

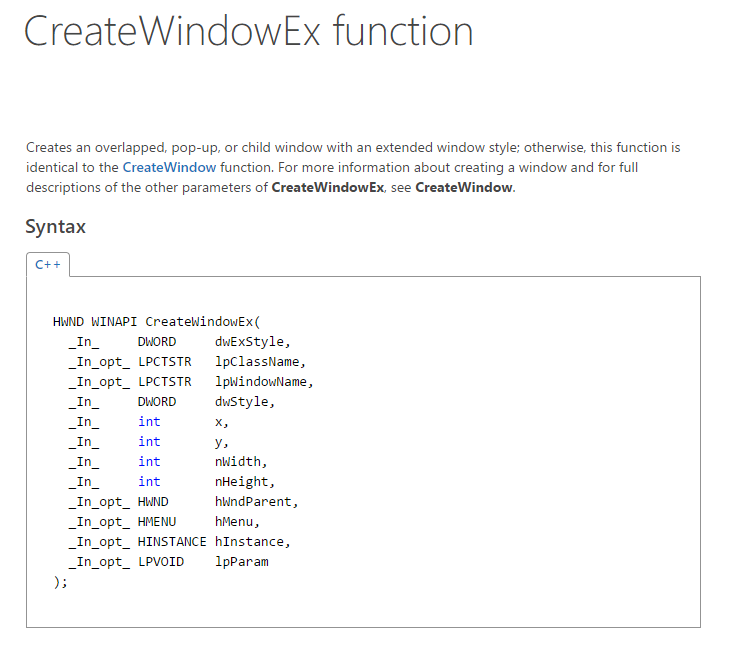


1. Self – Study - 31/05/2017
   1. **Overview:**
      1. This whole self-study period was me starting and almost completely finishing my stage 3 reporting journal. Because I had not started until now, it meant that I had a lot of time to nicely go over, and deeply study each and every crucial learning point from weeks 9 to 12. All the code shown in this stage 3 journal up to lecture 32 was all studied, and used as practise in this study period
   2. **Lessons Learnt and Bugs:**
      1. The only bugs I got where pointer indirection bugs and overlap bugs such as when I was making the recursion and self-referential structure. If you make it a full loop, you get an overflow/memory overflow bug. But these bugs where easily fixable. Because I was basically studying directly from my extremely extensive lecture notes and sometimes, straight from the slides, I did not come across to many bugs.
   3. **Hard sections to code:**
      1. The hardest sections to re-learn and study was definitely function pointers. The syntax was so confusing I had to pour over the code over and over to fully understand it. However, I now understand function pointers quite well with the typedef specifically. I am still extremely sketchy on the full syntax of function pointers.

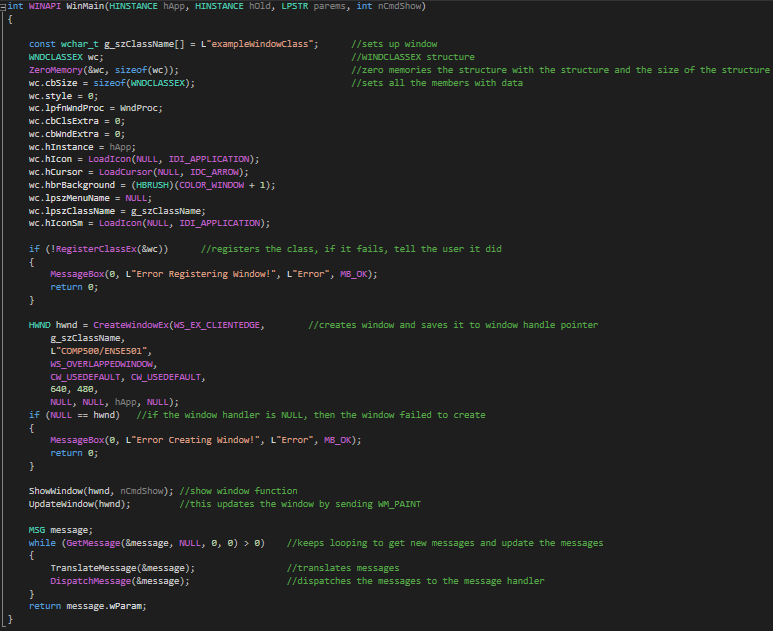
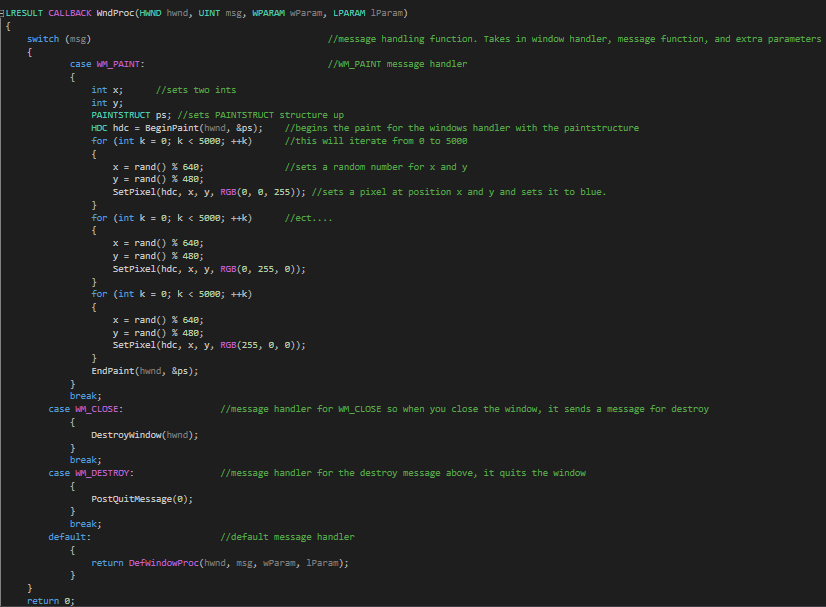


* + 1. Other hard sections to understand was the entire creation of a new window using windows API. During the actual lecture, I struggled to understand what was going on. But I got the basic gist of it. Whilst studying, I understood more, but still found windows API programming extremely tough to understand. Luckily, the windows message boxes where very easy to understand.

1. Lecture 35 – 1/06/2017
   1. **Overview:**
      1. This entire lecture was just going over what to expect in the exam. There were a few code examples shown, but nothing I hadn’t seen before.
   2. **Lessons Learnt: What to expect:**
      1. The exam consists of a lot of things, but these are all the things I gathered from this lecture:
      2. It will be 2 hours and 30 mins
      3. Don’t leave early, and try all questions
      4. The exam will be heavy on functions, structures, loops, logic and arrays.
      5. Pseudocode can be comments
      6. The exam will also look heavily at pointers, function pointers, passing by reference, p1colour and windows API a little bit. Mostly looking at simple message handling and message boxes.
      7. Can you debug? – Stepping through things.
      8. Read the questions carefully
      9. Remember off by 1 errors
      10. The exam will obviously have calling of functions, so make sure you know how storing of values works
      11. Follow the sample output if it is asked and do lots of tests to make sure the code works as intended.
2. Self – Study - 1/06/2017
   1. **Overview:**
      1. This whole self-study period was me finishing off and completely finishing my stage 3 reporting journal up till the Thursday lecture. Because I was going back over the windows API mostly it meant that I had a lot of time to nicely go over, and deeply study each and every crucial learning points from the last day in week 11 and all of current week 12. All the code shown in this stage 3 journal from week 11, the last day, till this overview right here was all studied, and used as practise in this study period.
   2. **Bugs:**
      1. There were no bugs, because all code I used was copied from the course notes. This was because I did not fully understand creating new windows and such. However, I did work on understanding message box handling more, and drawing with WM\_PAINT a bit. The one bug I did get was when I was copying in code for creating a new window, the WinProc function was not created yet, so when I saw that the WinProc was not defined yet, I didn’t know why since I wasn’t deviating from the code from the slides. At that time, I just commented the use of that function pointer out until later.
   3. **Hard sections and Lessons Learnt**
      1. I used F1 and read a bit more about each function to better understand what the functions are doing, what they take in, and what they actually do. Such as the CreateWindowEx function:



* + 1. I read over a lot of the function developer centre work to understand how most of them work, but I still don’t really know what is needed and how you know what most of the elements and functions are needed. I did go over the code in depth though to understand as much as possible and in the end my code was working so that was good. The commenting in my code bellow is what I understand of the code.



1. Lecture 36 – 2/06/2017
   1. **Overview:**
      1. Exam revision
      2. And class questions
   2. **Lessons Learnt: Learning outcomes**
      1. The lecture overall was not very useful because it just went over the exam and what to expect and answered some class questions. Because I think the lecturer (Hooper .2017) and the TA (Abbott. 2017) have been extremely good at teaching me programming, I am extremely confident when it comes to coding. I feel fully ready for the final exam, so while I did pay attention a little in class, the lecture wasn’t overall that useful.
      2. We did go over the learning outcomes that would be looked at in the exam:
      3. Learning outcome 1 – Read, comprehend, describe and example existing small scale computer program source code
      4. Learning outcome 2 – Design, implement and compile small scale computer programs that are syntactically correct, well documented and adhere to a programming standard
      5. Learning outcome 3 – Test, debug and modify small scale computer programs to resolve logic and runtime errors
      6. Learning outcome 4 – Use development tools to assist in the construction, maintenance and testing of computer programs.
      7. Learning outcome 5 – Write computer programs that feature program output and use input.
      8. Learning outcome 6 – Select appropriate data types for variables and use variables to store, retrieve and process simple data in computer memory.
      9. Learning outcome 7 – Write computer programs that contain statements which control the sequence, and iteration of computer instructions. Sequence is the order of a program
      10. Learning outcome 8 – Create modular and reusable functions featuring simple data input, output and processing.
      11. Learning outcome 9 – Write computer programs that use arrays and simple structures to store, retrieve and process data in computer memory
      12. Learning outcome 10 – Write computer programs that use computer file storage for data input, simple processing and data output.

Bibliography:

Abbott, J. (2017). *Weekly Labs* [Course Exercises and PowerPoint slides]. Learning Basic C. Retrieved from:

<https://blackboard.aut.ac.nz/webapps/blackboard/content/listContent.jsp?course_id=_85303_1&content_id=_3944184_1&mode=reset>

Burgener, R. (1988) *20Q, Twenty Questions*. 20q.net, Wikipedia. Retrieved from: <http://www.20q.net/>

<https://en.wikipedia.org/wiki/20Q>

Donati, L. (2013). *how to represent number bigger than long in c [duplicate].* Stack Overflow. Retrieved from:

<http://stackoverflow.com/questions/18675101/how-to-represent-number-bigger-than-long-in-c>

Hooper, S. (2017). *Weekly Lectures* [PowerPoint slides]. Learning Basic C. Retrieved from:

<https://blackboard.aut.ac.nz/webapps/blackboard/content/listContent.jsp?course_id=_85303_1&content_id=_3944184_1&mode=reset>

Hooper, S., Abbott, J. (2017). *Coding Examples* [Course Related coding examples]. Learning Basic C. Retrieved

from: [https://blackboard.aut.ac.nz/webapps/blackboard/content/listContent.jsp?course\_id=\_85303  
\_1&content\_id=\_3944184\_1&mode=reset](https://blackboard.aut.ac.nz/webapps/blackboard/content/listContent.jsp?course_id=_85303_1&content_id=_3944184_1&mode=reset)

Scott, T. (2014). *How the Self-Retweeting Tweet Worked: Cross-Site Scripting (XSS) and Twitter*. YouTube.

Retrieved from: <https://www.youtube.com/watch?v=zv0kZKC6GAM>

