**FoCA Fundamentals of Computer Architecture**

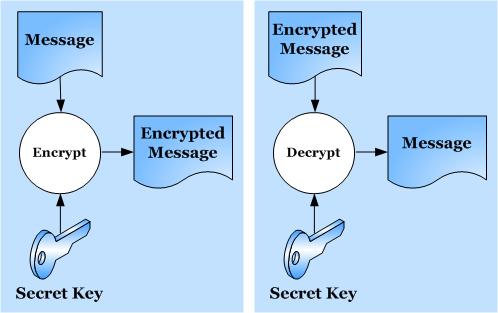
**Module code: 401756**

**Encryption Assignment Brief**

**x86 Assembly Programming**

**The Encryption/Decryption Program**

**Please read this brief thoroughly so you know what's involved, and follow the instructions therein. If you need clarification on any point just ask.**



**You write this! ☹**

**You're given this ☺**

**Not so !**

**Tutors: Adrian Oram, Sergio Davies, Mark Marshall**

**Cohort: 2019-2020**

**FoCA - Assignment Brief - The Encryption/Decryption Program**

**Overview**

This assignment is designed to test your understanding and application of x86 assembly and how it can be used in a C++ program. The tasks involved are based on an **encryption/decryption** program.

You are provided with a program written in C++ and x86 assembly that you will work with. Running this program and studying the code is recommended so that you become familiar with what it does - it allows the user to type in a number of alphanumeric (ASCII) characters and these are then ‘encrypted’, or converted, to another value. Both the original and encrypted data are echoed to the screen (see screen shot, overleaf), along with their hexadecimal values.

The example program initially provided currently uses an extremely simple form of encryption – it simply moves the character up one place in the ASCII table (by adding 1 to the character code), so decrypting it would be a simple matter of moving it back down one position - not a very strong encryption method, I’m sure you’ll agree!

However, you will each be allocated an encryption subroutine - written in x86 assembly - that you will paste into the original program to replace the existing ‘encrypt’ subroutine. There are currently 22 variations of these and each uses a more convoluted form of encryption than just adding 1. These encryption routines also employ a unique **encryption key** that will be allocated to each of you. Note that none of the encryption methods are 'real' - you won't find them being used by the security industry - they're invented purely for the purposes of this assignment.

Eventually you will write a **decryption** routine using **x86 assembly** to reverse the encryption process, but there are other tasks too, and these are phased as three milestones (described later). The due dates for each milestone is given on the back page of this handout.

**Please read this specification in its entirety before you start.**

**Preliminary Requirements**

This assignment document and other required materials are all available on Blackboard (Bb), under the ‘**Assessment**’ tab.

You may also need a copy of the **ASCII table** for reference. Access to your FoCA lecture notes and tutorials will also be useful.

Last, but not least, you need your brain!

**Please follow the next few steps carefully and fully.**

**a) Find your encryption key and allocated encryption routine number**

Your allocated key and encryption method are listed in a separate document on Bb:

**FoCA Encryption Assignment 2020 keys and code allocations.docx**

Open this file and find your name in the list. Make a note below of your **encryption key** (a single character, its case - upper or lower - is important) and which **encryption routine** (1-22) you should be using.

**b) Familiarise yourself with the supplied program & code**

Download the original C++/x86 assembly project file from the assignments area, and unzip it:

**FoCA Encryption Assignment.zip**

Open the **.sln** file in Visual Studio (VS) - this project is the starting point for the assignment. Take some time to inspect the C++ and x86 assembly code and how it's structured (it's not complicated).

**Immediately**, edit the first few lines as follows (see box, below):

* Overwrite "**Your name goes here**" with your name in the **StudentName** field. This ensures the code is identifiable as yours (and is used in the output file, see below).
* Overwrite the **Ekey = 'x'** with your allocated character replacing the **x**. This avoids having to enter it every time you run the program.

// Author: A.Oram (Feb 2018)

// Last revised Feb 2020 by A.Oram

char EKey = 'x'; // Replace x with your Encryption key.

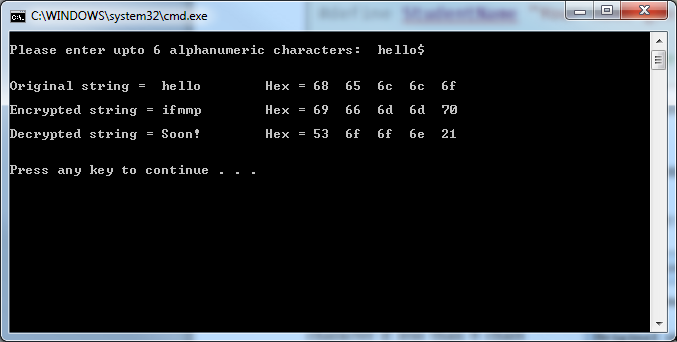
#define StudentName "Your name goes here"

#define MAXCHARS 6 // feel free to alter this, but 6 is the minimum

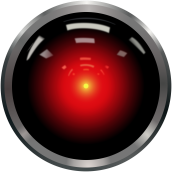
Note that the constant value governing the number of characters that can be entered by the user:

#define MAXCHARS 6

can be changed as you wish but 6 characters is the minimum number your final submitted version of the program should be run with.

Now build and run the program, entering a short text string when prompted.

In the screen shot opposite **‘hello’** is the original character string entered; '**$**' can be used as a terminating character if less than 6 chars are to be typed (any seventh or subsequent character is currently ignored by the program anyway). Press **Enter** after typing 6 or more characters.

The resulting **Encrypted string** version of ‘hello’ is ‘**ifmmp**’ - each character is simply moved one position up the alphabet in this simple version. ‘HAL’ would become ‘IBM’.[[1]](#footnote-1).

The **Decrypted string** shows “**Soon!**” indicating that you have some work to do - eventually the decrypted string will (should!) show the original string again.

It may eventually be fruitful to 'hardwire' a test string as the original data, to avoid entering it each time. To do this, simply add your required characters to the declarations as shown below (e.g. using "hello" as the test string):

**Before… After…**

char OChars[MAXCHARS], char OChars[MAXCHARS] = **"hello"**,

EChars[MAXCHARS], EChars[MAXCHARS],

DChars[MAXCHARS] = "Soon!"; DChars[MAXCHARS] = "Soon!";

**Note**: strictly speaking you should initialise this char array with individual characters, like this:

char OChars[MAXCHARS] = **{'h','e','l','l','o',' '}**

but using the double quotes, "string" version is OK. You will need to use one less character than MAXCHARS however, because of the way *strings* are stored, using a terminating null character (‘00’).

Then comment out the lines that request the original string input (in 'main') and initialise **char\_count** with the length of your test string (the number of characters, 5 here):

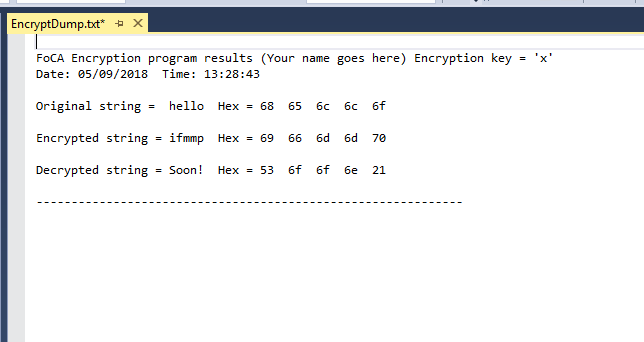
int char\_count (**5**);

**//**cout << "\nPlease enter upto " << MAXCHARS << " alphanumeric characters: ";

**//**get\_original\_chars (char\_count);

The hex ASCII codes of all of the original, encrypted, and decrypted characters are also displayed on the screen, as shown. Note that if the hex codes 0x**0A** or 0x**0D** happen to occur in the encrypted string then the display may appear strangely formatted. This isn't a bug – check these two values in your ASCII table to understand what is happening.

For convenience, the console output is also appended to a text file, called **EncryptDump.txt** which can be viewed in VS (see below) - find it under the **Resources** tab in the project. You can edit and delete this file easily in VS if need be, or use Notepad or similar on the file in the project directory. In VS use **Ctrl-A** then **delete** to clear it when necessary.



Each time you run the program you will be prompted about whether or not you want to update (actually, *append*, or add to) this file with the results - usually 'Yes' is the answer (press Enter) but you can avoid it too with 'No'.



**c) Install and run your allocated encryption code**

Open the document on Bb:

**FoCA Encryption Assignment 2020 - Routines.txt**

Locate your encryption code label, e.g. ‘encrypt\_5:’ and copy to the clipboard everything down to the next **ret** instruction (they are around 10 – 15 lines of x86 assembly), e.g.

;----------------------------------------

**encrypt\_5:**

push eax

mov al, byte ptr[eax]

push ecx

and eax, 0x7C

ror eax, 1

ror eax, 1

inc eax

mov edx, eax

pop ecx

pop eax

mov byte ptr[eax], dl

xor edx, ecx

rol dl, 1

ret

Replace the original code version of **encrypt\_nn**, shown below, in your program by pasting *your* encrypt routine over it:

**encrypt\_nn:**

mov edx, ecx

add edx, 1

ret

Ensure you also change the **call encrypt\_nn** instruction in the main program to match the name of yours, i.e. in this case:

**call encrypt\_5**

Re-build this version and ensure it works. The output will now show a very different set of encrypted characters. Note that these characters will frequently be non-ASCII, such as **¥æë☺** – the final encrypted character codes will often fall outside of the ‘normal’ range of printable ASCII characters which are between 2016 and 7E16. (see ASCII table), and thus may look strange - that's expected.

You should record the hex values generated for your chosen encrypt test string - you can then easily check that they remain constant as you develop your code (you may accidentally corrupt the encryption routine code whilst doing the assignment, for instance).

**Some points to note about the encryption routines**

* The **MOVSX** instruction is used to move byte-sized data (i.e. 8-bit ASCII characters) into dword (double-word) – sized registers, such as **EAX** (which are 32 bits wide). 'SX' means '**S**ign e**X**tended' and means that the sign bit (the left-most bit of the byte value) is duplicated (‘extended’) throughout the upper 24 bits of the 32 bit register whilst the byte data is copied into the least significant byte. In practice this often means that zeroes are used to fill up the unused parts of the register, e.g.:

**00000000 00000000 00000000 00110001** **EAX** register (32 bits)

**00110001** byte-sized ‘char’ data (8 bits)

Although x86 assembly language provides a lot of flexibility in moving different sized operands into one another, this is one example of an operand size mismatch, hence a special instruction has to be used. For most purposes a simple MOV instruction works.

The **MOVZX** (zero extend) instruction is similar – except **Z**eroes are used instead of the Sign bit. Note that byte data can be *signed* or *unsigned*…chars, however, are *unsigned*.

* An alternative to this would be to use the register names **AL** and **CL** as pseudonyms for the least significant byte of the **EAX** and **ECX** registers respectively for moves of byte-sized data (see register naming diagram, below) E.g. instead of:

movsx ecx,s\_char

movzx eax,EKey

you could use these:

mov cl,s\_char

mov al,EKey

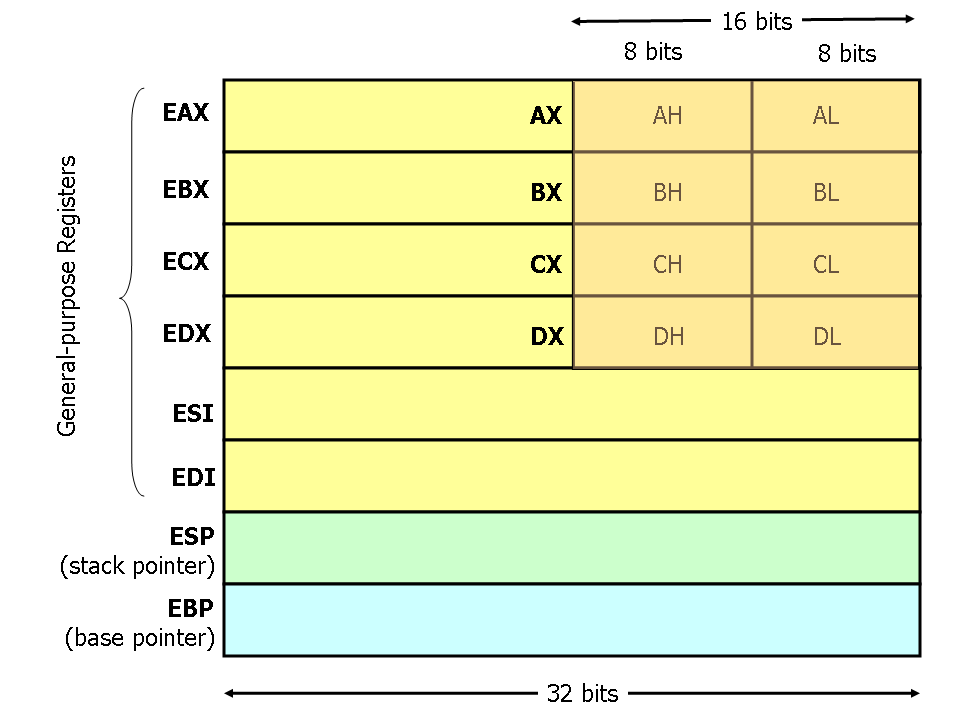
This does not however, put zeroes in the rest of the register, therefore if it so happens that the upper part of **EAX**, for example, has some old data in it, then you will probably get unexpected side-effects if some arithmetic is performed on that register! To avoid this zeroise the register before the move with an **XOR** operation, like this:

xor eax,eax // clear EAX register to zero

mov al,EKey // copy encryption key into the low byte of EAX

*(Programming in assembly is full of potential pitfalls like this – that’s why it’s a real programming skill!)*

See the guide on Bb to x86 assembly:

** FoCA x86 Assembly Guide.doc**

for the relationship between the 8, 16 and 32 bit register names, e.g.

**Note**: the encryption routines often employ the 8 bit register names for convenience, e.g. **AL**.

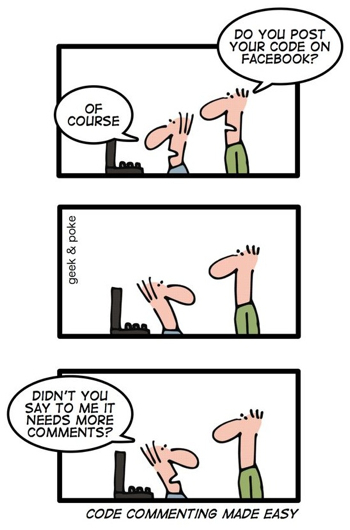
* The *parameters* to the assembly subroutine (inwards: the **encryption key pointer** and the **character to be encrypted**) are passed by moving them into registers **EAX** and **ECX**, respectively. The return of the assembly routine (outwards: the e**ncrypted version of the character**) is passed by moving it into **EDX**. One of your tasks will be to change this so that the standard C++ procedure calling mechanism is used instead, either **CDECL** or **StdCall**.
* Notice the use of **return** to return from the encrypt\_chars routine explicitly. Omitting this would allow execution to ‘fall through’ from the C++ code into the encrypt subroutine which would thus be erroneously executed one more time (with probably serious side effects!)

Remember that the full **Intel x86 reference manual** is on Bb. Refer to it when you need details of an instruction.

**Milestones to be met (and what you need to do.)**



1

**Milestone 1a: Understand the encryption code (up to 10 marks)**

Study the listing of your encryption routine and understand how it is called and encrypts a single character. Each subroutine does this differently; however, all are artificial in that they are not bona fide encryption methods, but that employ simple mathematical manipulations involving your encryption key in a fairly superficial way. You will not find them being used in a real application! They are also deliberately not written well and all could be improved (see Milestone 3)! Generally speaking, each routine alters your **Ekey** in some way ("mangles" it) and then applies it in a variety of ways to the character in order to encrypt it - for example, adding it, subtracting it, using it as a bit-shift count, and so on..

You should use the debugger to single step the program and/or run the program 'on paper' - following the way the registers are used and changed - to help with this task.

Once understood, you should then **add sufficient meaningful comments** (see note box on page 11 for what this means) to the source code to demonstrate your understanding of your encryption code. **Do not** attempt to improve the encryption code at this stage. You should refer to the **'Ekey'** and the **'char'** in your comments rather than **EAX** and **ECX**, as the former are more meaningful.

**DON'T forget to insert your name in the opening #define!**

Keep in mind how you might reverse the effect of the mathematics for the decryption task, later…

**Milestone 1b: Implement a C++ calling convention (up to 10 marks)**

The program you are provided with simply copies the parameters required by the encryption subroutine into the **ECX** and **EAX** registers, and passes the result back (the encrypted char) via the **EDX** register. You should **alter the program so that it adopts an accepted C++ standard calling procedure** (i.e. **cdecl** or **stdcall** - either will do) for passing parameters into the subroutine and returning values. This uses the stack and the **EBP** register rather than general purpose registers. Make sure you add comments explaining the purpose of each part of the cdecl/stdcall code that you use. **Note**: do not delete the existing two PUSH and two POP instructions in the main FOR loop for any reason. They can be altered for Milestone 3 purposes, later.

**The first parameter of your encryption method should be the character to be encrypted with the second parameter being the effective address of your Ekey.**

**Hint** - look at your lecture, tutorial, and other notes (e.g. FoCA x86 Assembly Guide.doc) to see how it could/should be done. Main points to note are:

**ESP**

**4 bytes wide**

**Stack Pointer**

**Base Pointer**

**EBP**

**saved ESI**

**saved EDI**

**local variable 3**

**local variable 2**

**local variable 1**

**saved EBP**

**return address**

**parameter 1**

**parameter 2**

**parameter 3**

**[ebp+8]**

**[ebp+12]**

**[ebp+16]**

**[ebp-4]**

**[ebp-8]**

**stack growth higher addresses**

* Subroutine parameters are pushed onto the stack before the call.
* Within the called subroutine the base pointer register, EBP, is used with a numeric offset to get the parameter values from the stack area, e.g.

// get first parameter

mov eax, [ebp+8]

The diagram opposite (taken from the **x86 Assembly Guide**) summarises the stack and register use where 3 parameters and 3 local variables are involved in a procedure call:

**Note**: your routine does not have any local variables.

**To meet milestones 1a and 1b you should:**

Submit your .cpp file to the Blackboard milestone 1 submission point by the deadline (see last page).

Your program code should be clearly laid out, with instructions and comments neatly aligned in columns. Use spaces as opposed to tabs to ensure correct display when viewed (you can set Visual Studio to automatically use spaces in place of tabs). **You will lose marks if you don't adhere to this advice! See the guidance on page 13 about listings.**

Be prepared to demo your code to show that it still works and produces encrypted characters during your tutorial session in the week of submission.

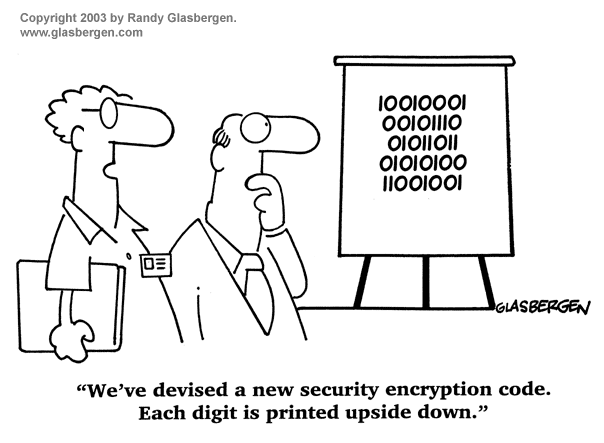
Your work will be marked and returned to you with feedback the following week – if you are struggling to meet this milestone seek help from your tutors. Any inadequate work (achieving less than 40% of the 20 marks) can be brought up to scratch after the return of feedback – improved work will then be re-assessed in due course but will be capped to 8 marks out of the available 20 (i.e. 40%) - this is **in-module retrieval**.

**Standard university rules apply to the submission of late work.**



2

In the meantime, move onto the work required of Milestone 2… (continued on the next page)



**Milestone 2a: Write the decryption routine (up to 40 marks)**



2 

Develop a **decrypt\_chars** subroutine, written in C++ and x86 assembly and similar in structure to the **encrypt\_chars** routine, that **reverses the encryption** so that the originally entered string is reconstituted.

You should thus be able to easily check that your decryption code works (or not!).

The assembly code you produce **should be fully and usefully commented** and use the same **stdcall/cdecl** mechanism employed in step 1b (if step 1b has not been completed, then use an appropriate method to pass the parameters to your decryption routine instead, although reduced marks will be awarded).

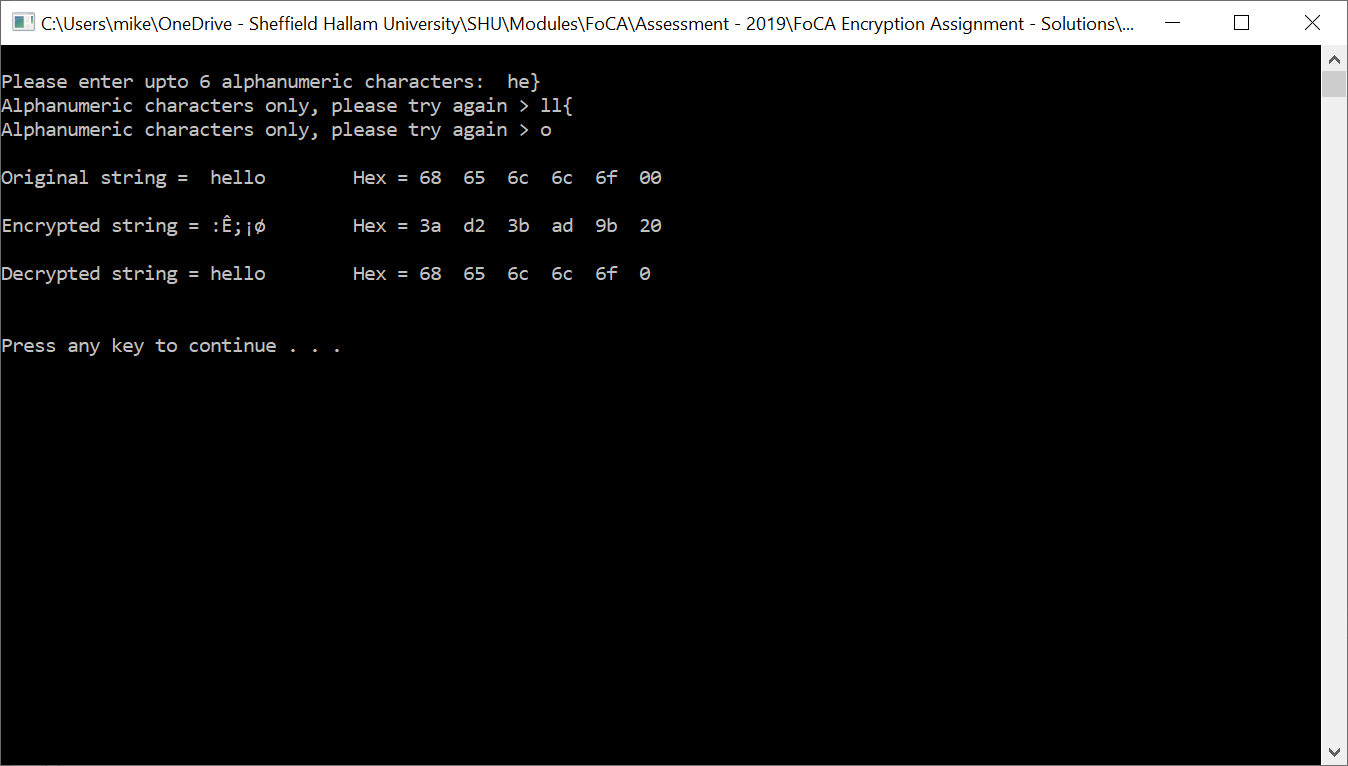
**Please note:** reversing the encryption algorithm is **not** done by simply reversing the order of the instructions and attempting to do the opposite of the original code, such as a 'pop' instead of a 'push'. **THIS WILL NOT WORK!**

Rather, you need to understand the algorithm and reverse its effect like a mathematical formula.

**Milestone 2b: Limit the character input range (up to 10 marks)**

The **get\_char** subroutine allows any character to be entered. Modify **get\_char**, using x86 assembly, to accept ASCII characters that are within the range of **'0'** (zero) to **'z'** (lower case) only (check your ASCII table). You will also need to accept **'$'** as a valid input (the terminating character) despite it being outside of this range. If an invalid character is entered, it should keep asking for a new input until one is given, additionally displaying the error message **"Alphanumeric characters only, please try again > "**.

Reading characters from the keyboard and displaying characters to the screen should be done using C++’s **cin/\_getwche** and **cout** methods but checking if a character is in range should be done in assembly (i.e. the branching IF – THEN code must be written in assembly). Keep the original **get\_char** subroutine as C++ and update the body with appropriate inline assembly code and, of course, plenty of comments.

An example of this behaviour is shown in the screenshot on the right.

If you are unable to fully implement this behaviour, comment out what you have and revert to the original code given. You may get partial marks for what you have attempted (even commented out), but more importantly, you will still have a working program to continue with the other steps.

**To meet milestone 2a and 2b you should:**

Submit your .cpp file to the Blackboard milestone 2 submission point by the deadline (see last page).

**Again, see the guidance on page 13 about listings.**

A demo of your code successfully (or otherwise!) decrypting a test string should be given during your tutorial session.

Your decryption code will be marked and returned to you with feedback before the Easter break. In-module retrieval is not available for this milestone.

**Standard university rules apply to the submission of late work**



3

In the meantime, move onto the work required of Milestone 3… (continued on the next page)

**Milestone 3a: Convert encrypt\_chars to x86 assembly and improve it.**



3

**(up to 20 marks)**

To the best of your abilityconvert the body of the **encrypt\_chars** function(shown highlighted on the original version given below) to **x86 assembly code**.

The **FOR** loop and the use of the **OChars** and **EChars** arrays should all be written in assembly language.

You should also attempt to improve the routine, for example by:

* optimising/minimising the number of registers required and instructions used to make the code smaller (reduce its footprint) - and thus potentially faster,
* identifying and removing any redundant (unnecessary) code that may be in there,
* removing the call to **encrypt\_nn** and inserting the encryption code directly in its place - this is called **'inlining'** - a technique commonly used by compilers to improve program performance. This might provide further opportunities to improve/simplify your code. If you do this you will need to remove the **CDECL/StdCall** code as it will thus be redundant (no calling now going on!).

Remember to always maintain/update your program's comments accordingly!



**Note: Keep separate copies of your original program (used for Milestones 1 and 2) as backups as you may find it useful to check your assembly code is working correctly (as per the original C/C++ code) – use a copy for Milestone 3.**

void encrypt\_chars (int length, char EKey)

{ char temp\_char; //

for (int i = 0; i < length; i++) //

{ temp\_char = OChars [i]; //

\_\_asm { //

push eax //

push ecx //

//

movzx ecx,temp\_char //

lea eax,EKey //

call encrypt\_nn // replace with the actual routine

mov temp\_char,al //

//

pop ecx //

pop eax //

}

EChars [i] = temp\_char; //

}

return;

**Milestone 3b: Convert upper case to lower case chars (up to 5 marks)**

As a final step in the encryption code you have written, add **efficient** assembly code to your encrypt\_chars routine that **converts any upper case alphabetic characters** to **lower case** characters before encrypting.

So, an original input string of e.g. “**SteWie**” should appear after decryption as “**stewie**”.

**Milestone 3c: Produce a well-presented program (up to 5 marks)**

A good program doesn’t stop at the quality of the code, be it C++ or assembly – documentation is very important and this manifests as copious and useful comments in program listings.

You should **ensure that your program is liberally commented** to the point that a third person could readily understand what you have done. You should remove stale comments (e.g. those that are no longer relevant) and update others (such as who wrote it!). Naturally, you should always ensure correct spelling is employed. These marks are a reward for producing understandable code, please try to gain them!

**Note: What are good comments?**

Commenting assembly (and other language) code is **very** important - understanding another person's program is much easier if comments are used properly. In assembly programs it's not unusual to have a comment on every line at least. Frequently an assembly program source file will contain more comment text than program text! The **source code formatting** used is also important; an assembly program listing generally contains 4 columns of information, in this order:

**label: instruction-mnemonic operand(s) // comments**

and these are normally aligned in columns (although loop sections of code may be indented by a couple of spaces to show the scope of the loop body). Spaces or tabs are used when the label column isn't used - see the encrypt\_5 example earlier. Blank lines can separate logically related stretches of code from other parts and aid the reader too.

**Comments** should aid the understanding of the reader as to **why** something is being done, not **what**. The 'what' is obvious from the instruction mnemonic and does not need repeating - you can assume any reader will be familiar with the language in use. Comments should augment instructions by explaining what purpose they’re fulfilling - often these comments can be based on the pseudo code from which the code was written - perfect!

Below is an example of good and bad commenting (and label use!) - can you deduce which is the good one?! The program is the same in each case:

//------------------------- **version 1** -----------------------------------

xor ecx,ecx // ?? wot's this do? **-UNPROFESSIONAL/WORTHLESS**

mov edx,0x64 // set edx **-WHY ARE YOU SETTING EDX?**

x1: call mySub //get data **-MEANINGLESS LABELS**

cmp eax,0x53 //compare with 53 **-DON'T REPEAT THE INSTRUCTION!**

jne x2 // jump **-ADDS NOTHING TO UNDERSTANDING**

add ecx,1 //add 1 to ECX **-DON'T-REPEAT-THE-INSTRUCTION!!!**

x2: sub edx,1 //decriment edx by 1 **-USELESS, AND BAD SPELLING TOO!**

jnz x1 // jump **-POOR LAYOUT/FORMATTING**

//-----------------------------------------------------------------------

//------------------------- **version 2** -----------------------------------

// Code to count how many times the letter 'S' is typed in 100 keypresses

// ECX is used as character counter. EDX is loop counter. EAX = char input.

xor ecx,ecx // initialise the 'S' counter to zero

mov edx,100 // set loop counter to 100

// FOR i=1 to 100 do

for: call getchar // input a char from keyboard (returned in EAX)

cmp eax,'S' // if character = 'S'

jne endif // then

add ecx,1 // increment the 'S' character counter

endif: // endif

sub edx,1 // decrement loop counter and...

jnz for // ...loop if counter <> zero

//-----------------------------------------------------------------------

Please avoid word-wrap on any listings you submit for marking (or reading!) to any of your programming tutors/employers for the rest of your life. Comments and/or code that word-wrap make code difficult to read and must be avoided - use landscape mode when printing if necessary or manually continue comments onto a second line (split them with a *return* char). The presentation marks for milestone 3 are there to reward well-presented and readable programs . (See page 13 too.)

Whether I or anybody else asks you to or not, get into the habit of commenting all code that you write - look how hard it's been for you to understand just a dozen or so assembly instructions because there are no comments! Use layout, indentation, blank lines and plenty of (useful!) comments to make your code easy to read and understand…

In a similar vein, remove all redundant and stale comments - I've seen many listings with the opening three comment lines still in there, yet they're irrelevant!

**To meet Milestone 3a, 3b, and 3c you should:**

Submit your .cpp file to the Blackboard milestone 3 submission point by the deadline (see last page).

Your optimised/improved version will be marked and returned, along with a generalised feedback sheet within 3 weeks of the final hand-in deadline.

**Standard university rules apply to the submission of late work**

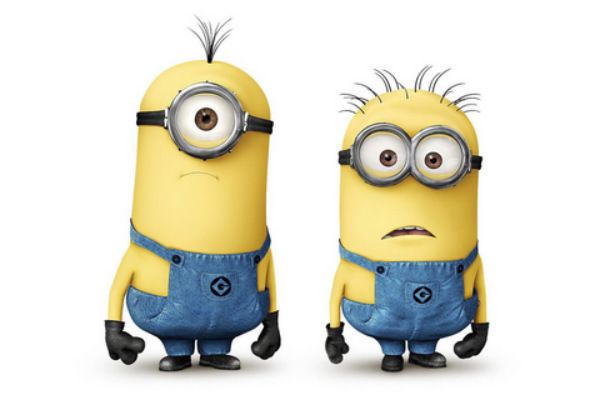
The Grade Centre on Bb will be updated with your marks. Anyone scoring less than 40% overall will be given the chance to do assignment referral work if they fail the module overall.

**Other information**

This is an **individual piece of work.** Incidents of cheating or copying will be heavily penalised and may lead to failure of the module. We mean this, so don't risk it. If you work closely with someone to the point that your work looks similar let us know, as we'll assume one of you copied off the other otherwise… we strongly encourage helping one another, and seeking help if stuck, but we deplore cheating.

An individual walkthrough of your code may be requested for any milestone before a mark is awarded. If you are asked to give a walkthrough of your code, you will be expected to demonstrate your knowledge and understanding of what you have submitted (i.e. make sure you are able to discuss all aspects of your submission). This assessment is designed for you to demonstrate your knowledge and application of techniques and not someone else's; if you are unable to successfully discuss all aspects of your work in a walkthrough, your grade will reflect only those aspects you can.

**Note**: Always keep backup copies of all your work, whether paper or electronic. Loss of material due to failure to do this is not grounds for ‘extenuating circumstances’ claims, or deadline extensions! University rules, I'm afraid.



This version has been edited down to the relevant parts only. **As tab stops can differ in visual studio, it is highly recommended that you adjust your tab settings to insert two spaces instead of a tab character – this will ensure your formatting will look the same regardless of how you view your code**. Add blank lines if it helps clarify code. Make sure your name, group and degree route appear at the top somewhere.

void encrypt\_chars (int length, char EKey)

{

char temp\_char; // Character temporary store

for (int i = 0; i < length; i++) // Encrypt characters one at a time

{

temp\_char = OChars[i]; // Get the next char from Original Chars array

// The following comments are WH Auden's "The Night Mail" poem...

\_\_asm

{ // (Well, you weren't expecting me to use actual comments, did you!?)

push eax // "This is the Night Mail crossing the border,

push ecx // Bringing the cheque and the postal order,

 // Letters for the rich, letters for the poor,

movzx ecx, temp\_char // The shop at the corner and the girl next door.

lea eax, EKey // Pulling up Beattock, a steady climb:

call encrypt\_X // The gradient's against her, but she's on time.

mov temp\_char, dl // Past cotton-grass and moorland boulder

// Shovelling white steam over her shoulder,

pop ecx // Snorting noisily as she passes

pop eax // Silent miles of wind-bent grasses."

}

EChars[i] = temp\_char; // Store encrypted char in the Encrypted Chars array

}

**Note**: These lines should not be deleted when doing milestones 1 and 2. It's not necessary for any reason.

return;

\_\_asm

{

encrypt\_X: push eax // "Letters of thanks, letters from banks, Letters of joy from the girl and the boy,

mov al, byte ptr[eax] // Receipted bills and invitations, To inspect new stock or visit relations,

push ecx // And applications for situations, And timid lovers' declarations

and eax, 0x7C // And gossip, gossip from all the nations, News circumstantial, news financial,

ror eax, 1 // Letters with holiday snaps to enlarge in,

ror eax, 1 // Letters with faces scrawled in the margin,

inc eax // Letters from uncles, cousins, and aunts,

mov edx, eax // Letters to Scotland from the South of France,

pop ecx // Letters of condolence to Highlands and Lowlands

pop eax // Notes from overseas to Hebrides, Written on paper of every hue,

mov byte ptr[eax], dl // The pink, the violet, the white and the blue, ...

xor edx, ecx // The chatty, the catty, the boring, adoring,

rol dl, 1 // Clever, stupid, short and long, The typed and the printed and the spelt all wrong."

Ret // <shortened> **WH Auden 1936**. See <https://www.youtube.com/watch?v=zmciuKsBOi0>

}} // OK, the closing brace should be on a new line, but I’ve run out of space

This version uses Word's default portrait orientation, tab stops, and margins (which are too wide to avoid word-wrap), and Times Roman font (which isn’t fixed width). **It looks awful.** Nothing's lined up neatly. If you are ever asked for a printout of your code avoid it looking like this, **it’s a** **sure-fire way to lose marks, so please don't do this! *We hate this!***

void encrypt\_chars (int length, char EKey)

{ char temp\_char; // Character temporary store

for (int i = 0; i < length; i++) // Encrypt characters one at a time

{

temp\_char = OChars [i]; // Get the next char from Original Chars array

// The following comments are WH Auden's "The Night Mail" poem...

The dreaded **word-wrap,** avoid at all costs!

\_\_asm { // (Well, you weren't expecting me to use actual comments, did you!?)

push eax // "This is the Night Mail crossing the border,

push ecx // Bringing the cheque and the postal order,

// Letters for the rich, letters for the poor,

movzx ecx,temp\_char // The shop at the corner and the girl next door.

lea eax,EKey // Pulling up Beattock, a steady climb:

call encryptX // The gradient's against her, but she's on time.

mov temp\_char,al // Past cotton-grass and moorland boulder

// Shovelling white steam over her shoulder,

pop ecx // Snorting noisily as she passes

Staggered indentation looks awful

 pop eax // Silent miles of wind-bent grasses."

}

EChars [i] = temp\_char; // Store encrypted char in the Encrypted Chars array

}

return;

// Inputs: register EAX = 32-bit address of Ekey,

// ECX = the character to be encrypted (in the low 8-bit field, CL).

// Output: register EAX = the encrypted value of the source character (in the low 8-bit field, AL).

\_\_asm {

encryptX: push eax // "Letters of thanks, letters from banks, Letters of joy from the girl and the boy,

mov al, byte ptr[eax] // Receipted bills and invitations, To inspect new stock or visit relations,

push ecx // And applications for situations, And timid lovers' declarations

and eax, 0x7C // And gossip, gossip from all the nations, News circumstantial, news financial,

ror eax, 1 // Letters with holiday snaps to enlarge in,

ror eax, 1 // Letters with faces scrawled in the margin,

inc eax // Letters from uncles, cousins, and aunts,

mov edx, eax // Letters to Scotland from the South of France,

pop ecx // Letters of condolence to Highlands and Lowlands

pop eax // Notes from overseas to Hebrides, Written on paper of every hue,

*etc, etc*

Doesn't fit on one page!

**Summary of marking structure and due dates.**



1

**Milestone 1** Due date and time: **3pm on Thursday 12th March 2020 via Blackboard**

a: Understand the encryption code (up to 10 marks)

b: Implement a C++ calling convention (up to 10 marks)



2

**Milestone 2** Due date and time: **3pm on Thursday 2nd April 2020 via Blackboard**

a: Write the decryption routine (up to 40 marks)

b: Limit the character input range (up to 10 marks)



3

**Milestone 3** Due date and time: **3pm on Thursday 30th April 2020 via Blackboard**

a: Convert encrypt\_chars and improve it (up to 20 marks)

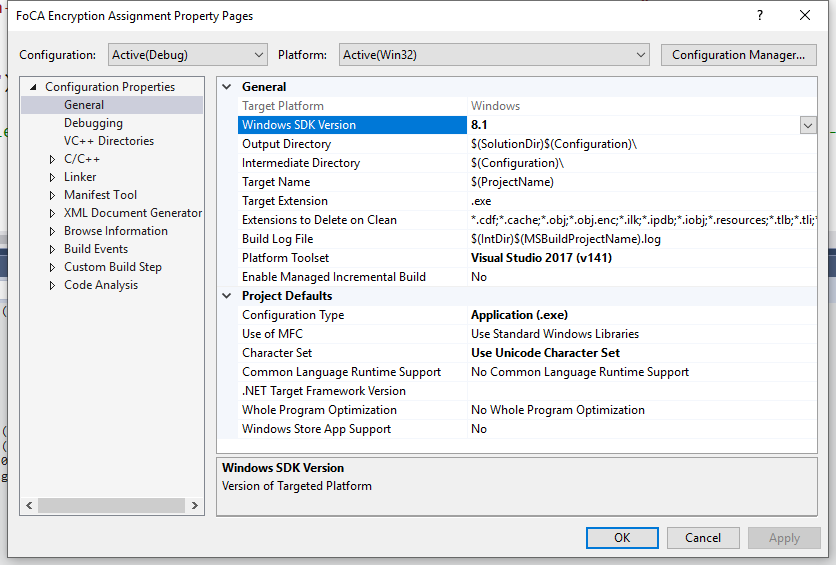
b: Convert upper case to lower case chars (up to 5 marks)

c: Produce a well-presented program (up to 5 marks)

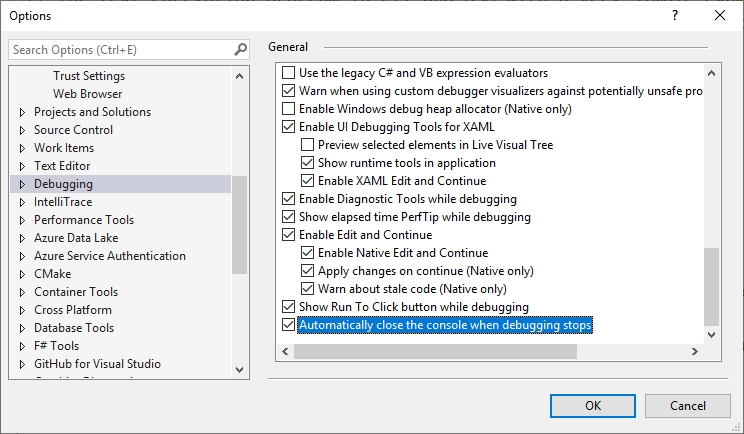
**Total marks available: up to 100 marks**

*This assignment is worth 65% of the module assessment. The other 35% comes from the final exam.*

Addendum 1– Project was originally built with SDK version 8.1, in VS2017 v141.



Addendum 2– Prevent terminating messages in VS via Tools/ Options, then Debugging, scroll to bottom of window and check the box…



1. Those of you who like conspiracy theories may like to know that the film “*2001, A Space Odyssey*” attracted some controversy about the origin of the name for the highly impressive main computer which features so centrally in the film: ‘HAL’. Was this an early example of subtle product placement by IBM? Given that the computer finally goes ‘nuts’ in the film, perhaps not! [↑](#footnote-ref-1)