**1**

سنحتاج الى بعض المتطلبات الاساسية لتعلم لغة التجميع (assembly language programming) باستخدام MASM 64-bit وايضا text editor لنتعامل مع MASM source files و linker و C++ Compiler .

**Setting Up MASM**

MASM هو احد منتجات Microsoft وهو جزء من ادوات visual studio developer tools . يعمل على كافة اصدارات ويندوز لكن الان انا اتعامل معها على اصدار windows 10 ويمكنك ايضا التعامل معها على اصدار windows 11 او windows 7 الخ. لذالك على الاقل يجب عليك ان تفهم اساسيات الـ visual studio community والتعامل معه ارجح لك هذا الشرح :

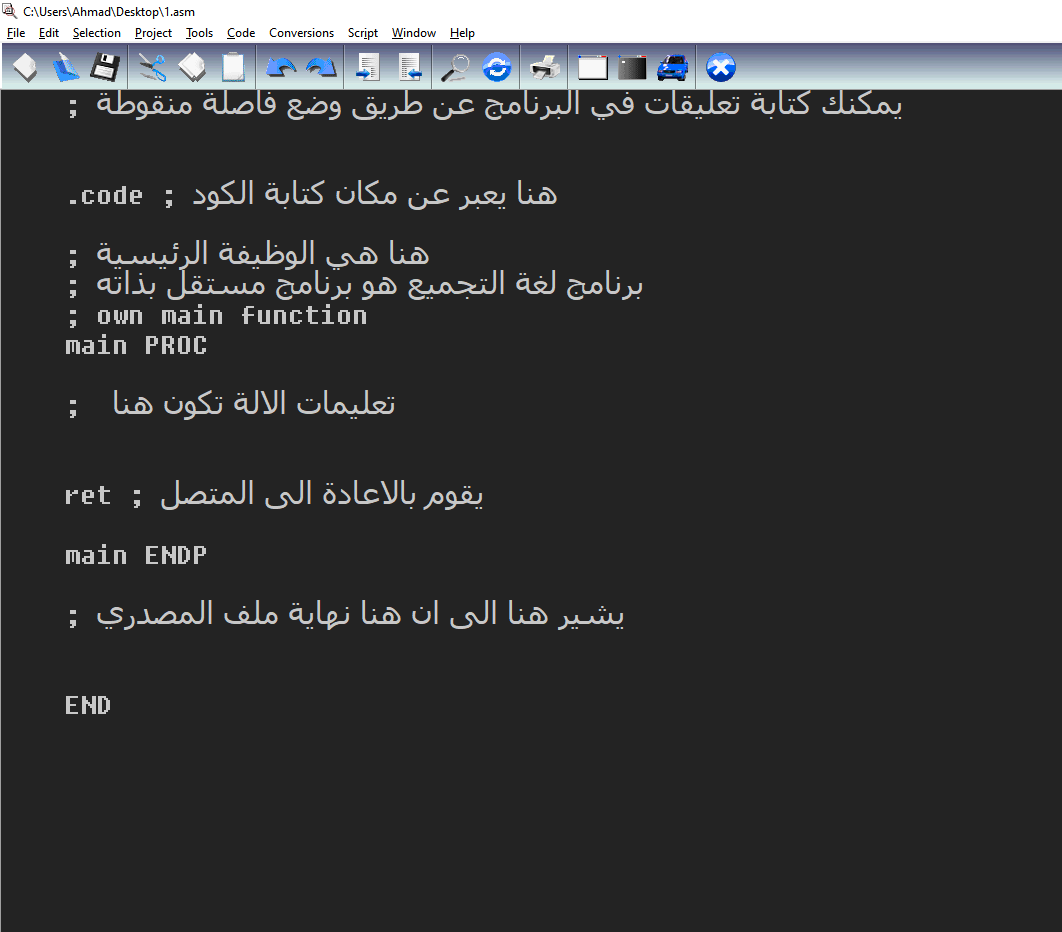
<https://www.youtube.com/watch?v=REG-p_eFNIw>

**Setting Up a Text Editor**

Visual Studio يتضمن محرر نصوص (Text Editor) ويمكنك استخدامه لانشاء برامج C++ & MASM . اذا قمت بتنزيل Visual Studio للحصول على MASM فانك تحصل تلقائيا على محرر نصوص عالي الجودة ويمكنك استخدامه لتعامل مع الملفات المصدريه للغة التجميع. وعلى كل حال يمكنك استخدام اي محرر يمكنه التعامل مع ASCII Files لأنشاء ملفات مصدرية لـ MASM & C++ مثل Notepad++ او sublime او المتاح في <https://www.masm32.com/> او حتى على الـ notepad العادي. برامج معالجة النصوص مثل Microsoft Word ليست مناسبة للملفات المصدرية للبرامج.

**The Anatomy of a MASM Program**

**2-1**



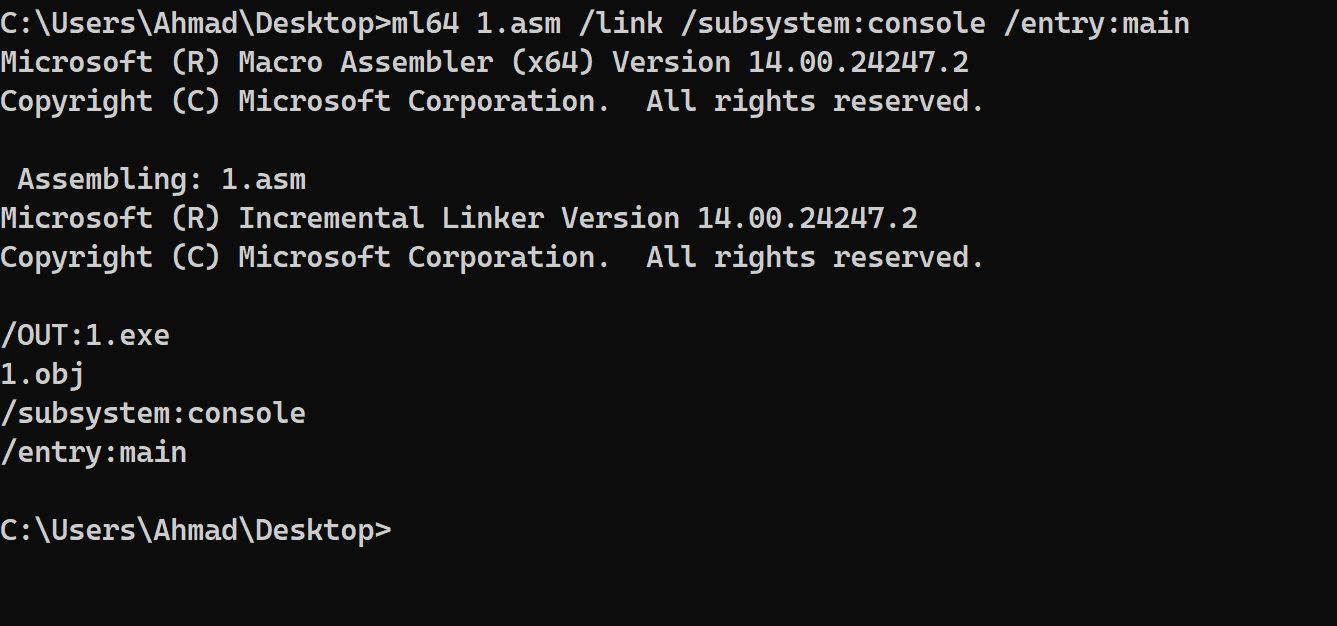
يحتوي برنامج MASM على قسم واحد او اكثر يمثل نوع البيانات التي تظهر في الذاكرة. تبدأ هاذي الاقسام (sections) مثل .code or .data المتغيرات وقيم في الذاكرة تكون في .data الـ Machine instructions تظهر ضمن قسم التعليمات البرمجية (.code). الخ تعتبر الاقسام الفردية التي تظهر في ملف مصدر لغة التجميع اختيارية (optional) .

الـ .code هي مثال لتوجيه الـ Assembler عبارة تخبر MASM بشيء عن البرنامج ولكنها ليست تعليمة من الـ X86 OR 64 الفعلية. بمعنى اخر يخبر بشكل خاص MASM ان يجمع العبارات التالية له في قسم خاص من الذاكرة خاص في machine instructions .

**Running Your First MASM Program**

طباعة شيء بسيط في لغة التجميع مثل “Hello World” يعتبر اساسي. لطباعة هذا النص يجب عليك اولا تعلم العديد من تعليمات الاله وتوجيهات المجمع وايضا استدعاءات النظام في ويندوز لطباعة سلسلة “Hello World” . النقطة من هذا الموضوع يعتبر طباعة Hello World امرا مختلف واعقد من لغات مثل C/++ او Python بتعليمة واحدة يمكنك الطباعة لكن هنا الامر مختلف. صورة 2-1 يعتبر برنامج كامل يمكنك عمل عليه assemble (compile) وتشغيله. في الواقع هو لا ينتج اي outputs انه ببساطة ينتج شيء واحد فقط هو return . مع ذالك هو يعمل ويمكن توضيح اليه تجميع(assemble) ملف مصدر للغة التجميع وربطه(link) وتشغيله. MASM يعتبر (traditional command line assembler) بمعنى يتم استخدامه في Command line ومتوفر في cmd.exe . للقيام بذالك يمكنك تجربة :

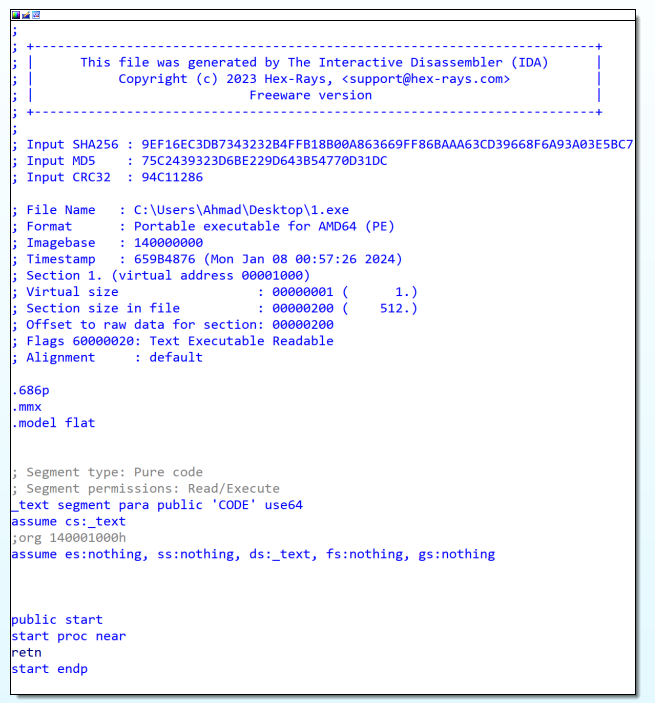
**ml64 1.asm /link /subsystem:console /entry:main**



هذا الامر يخبر بتجميع ملف المصدري 1.asm حيث قمت بحفظ الملف 2-1 الى ملف قابل للتنفيذ وربط النتيجة لانتاج تطبيق console application (تطبيق يمكن تشغيله من خلال سطر الاوامر) بعدها يمكنك تنفيذ البرنامج . بافتراض عدم حدوث اي خلل في البرنامج يمكنك تشغيله هكذا :



صورة من داخل IDA :



بعد تشغيل البرنامج يجب Windows ان يستجيب فورا من خلال تنفيذ سطر في الـ cmd حيث 1.exe يقوم بارجاع التحكم الى windows بعد بدء تشغيله.

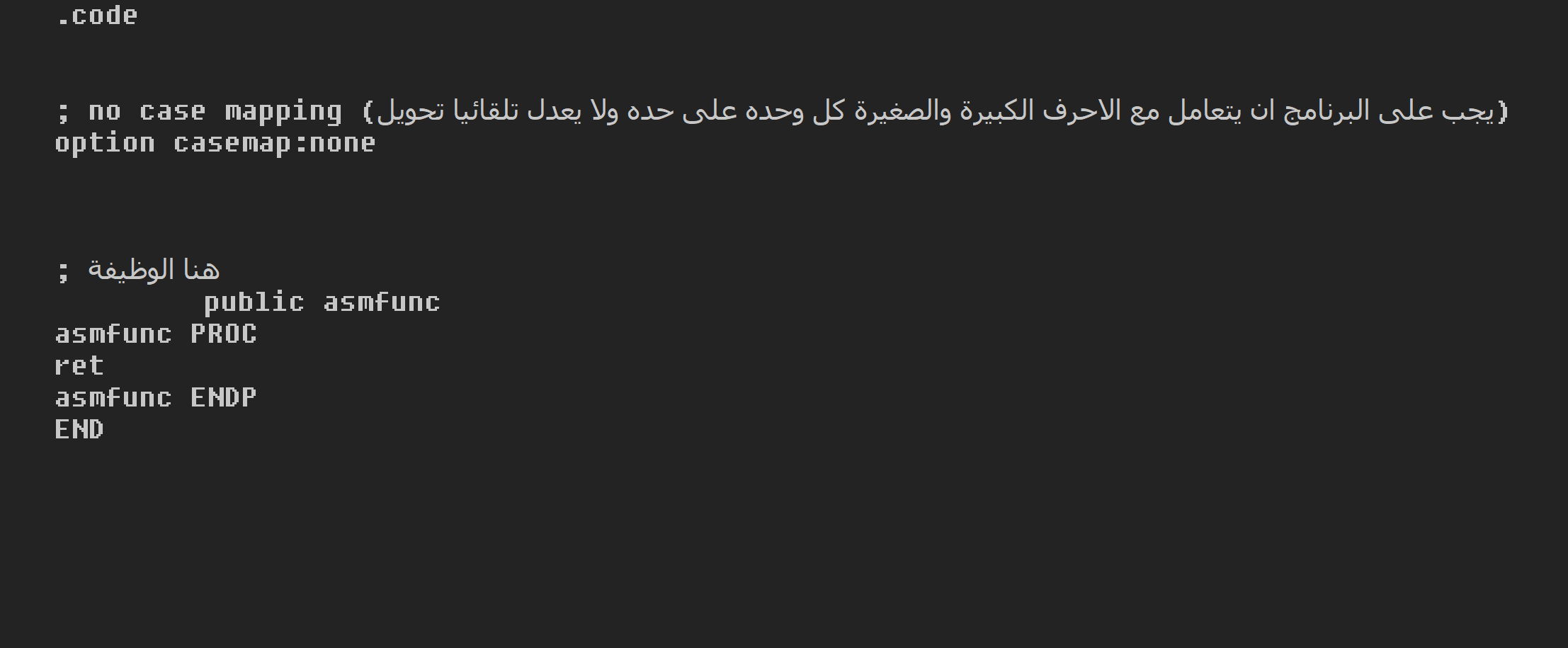
**Running Your First MASM/C++**

**عملية الترجمة مع Assembly & C تختلف قليلا عن برنامج MASM مستقل سنوضح هنا كيفية انشاء وتجميع وتشغيل البرنامج التجميع المختلط مع C/++ هذا هو البرنامج :**

**1.cpp**

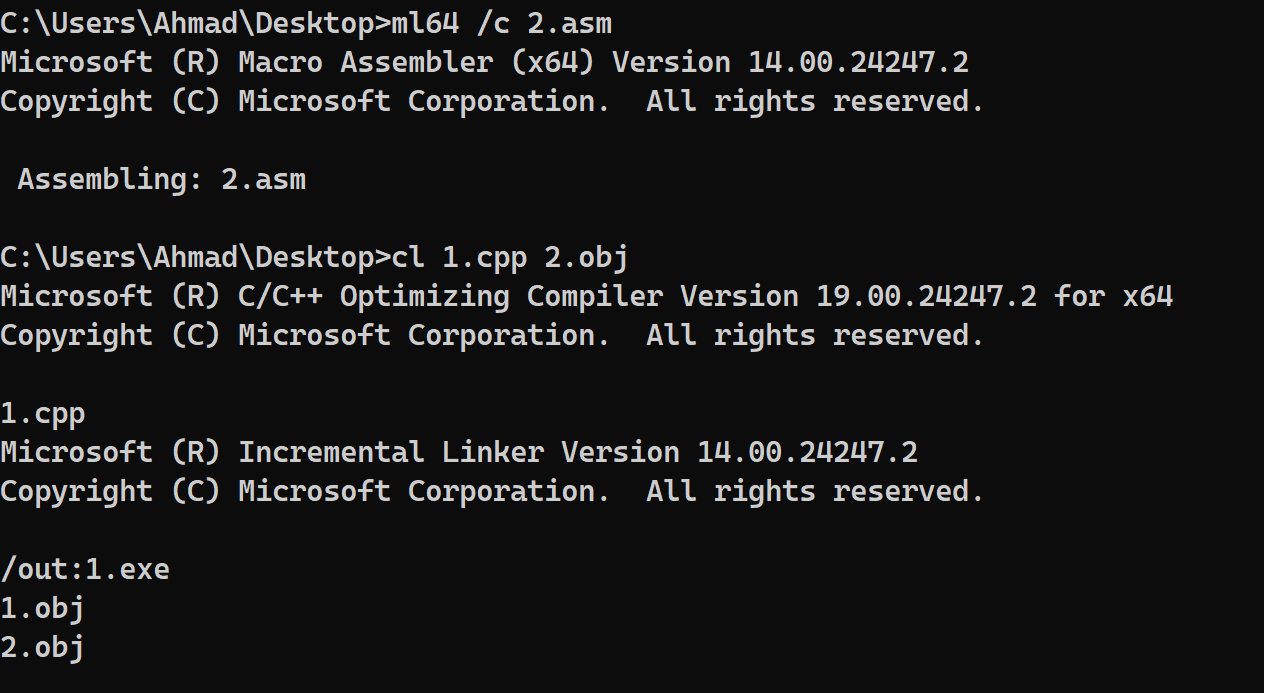
****

**2.asm**

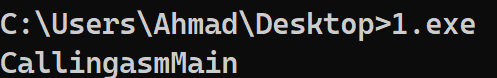
****

**في 2.asm تحتوي على ثلاثة متغيرات واثنان منهم جديدة option statement و public statement . وهذا ماذا يعني ؟ الـ option statement تشير الى MASM لجعل كافة الرموز حساسة لحالة الاحرف. يعد هذا ضروريا لان MASM افتراضيا غير حساس لهاذي لحالة الاحرف ويقوم بتعين كافة المعرفات الى احرف كبيرة بحيث asmfunc تصبح ASMFUNC . C/++ هي حساسة لحالة الاحرف تتعامل مع asmfunc و ASMFUNC كحالتين مختلفتين. الـ public statement يعلن انه عام هذا المعرف asmfunc وسيكون مرئيا خارج source/object file الخاص في MASM . بدون هذا البيان يمكن الوصول الى asmfunc فقط داخل MASM Module وسوف يشكل تحويل البرمجي في C++ الى معرف غير محدد . والفرق الثالث تم تغير main الى asmfunc لانه اذا تم استخدام نفس الاسم في كلتا الملفين 2.asm & 1.cpp لهذا سيتم خلط لانه يوجد اسمين.**

**لتجميع هاذي الملفات المصدريه وتشغيلها بهذا الامر :**

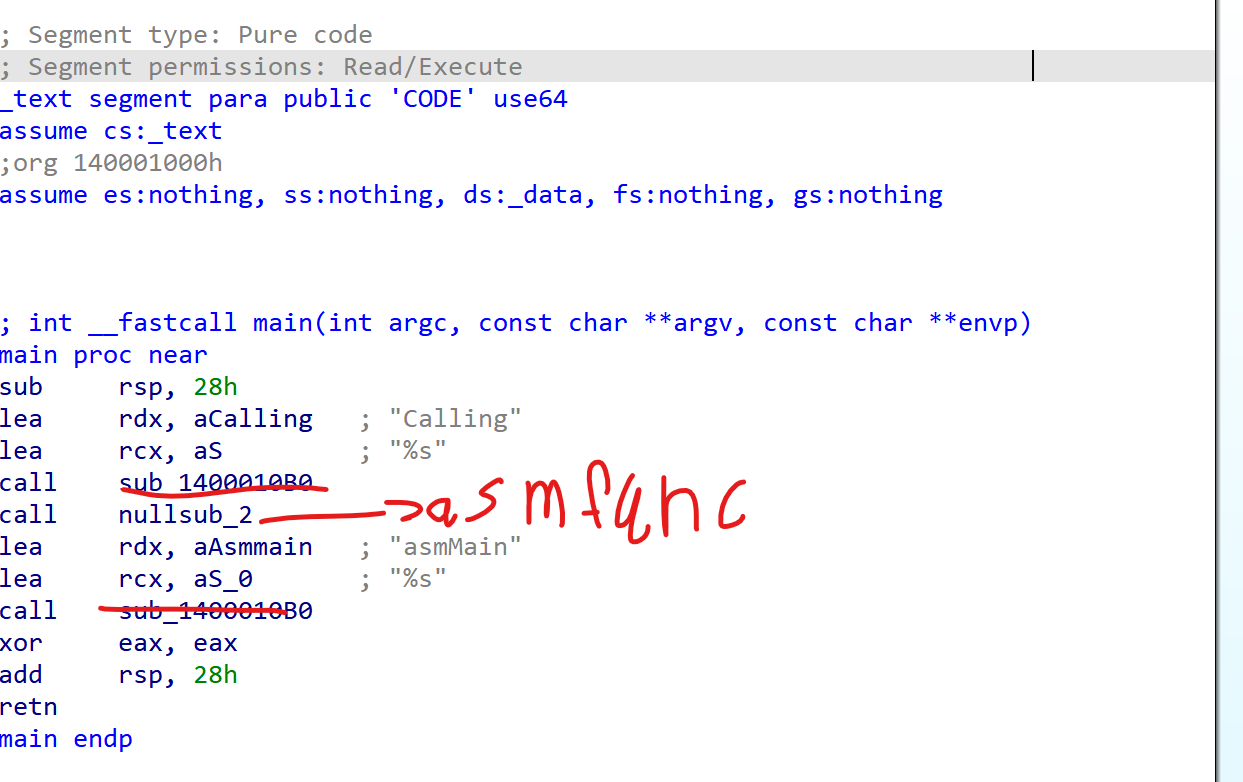
****

**المخرجات :**

****

**تم تشابك الاسماء بسبب عدم استخدام new line character (\n) .**

**صورة من داخل IDA :**

****

**الـ ml64 تم استخدام /c option الذي يرمز الى تحويل البرمجي فقط ولا يحاول تشغيل الـ linker لانه سيفشل لانه برنامج غير مستقل. Output من MASM هو ملف رمز كائن (object code) 2.obj والذي يعمل كادخال الى برنامج Compiler (Microsoft Visual C++ (MSVC)) . يقوم الامر cl بتشغيل MSVC compiler على ملف 1.cpp والارتباطات موجودة في التعليمات البرمجية الي صار لها assembled code في 2.obj . الاخراج من برنامج MSVC compiler هو الملف القابل للتنفيذ 1.exe .**

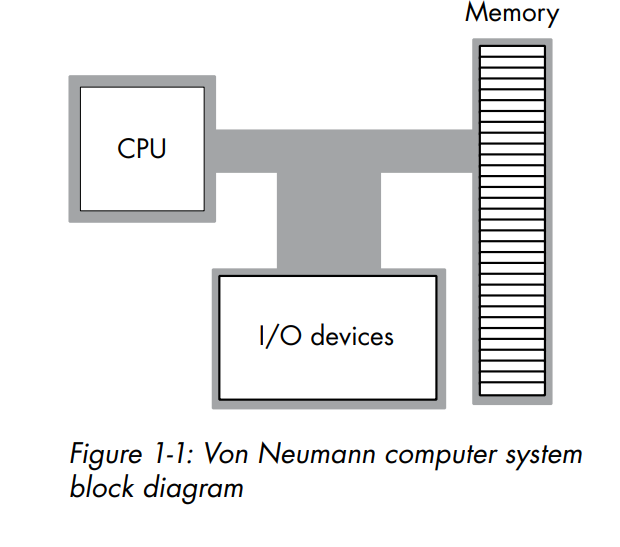
**An Introduction to the Intel x86-64 CPU Family**

حتى الان قمنا بتجميع وتشغيل برنامج واحد MASM وهو فقط يقوم باعادة التحكم الى Windows . قبل ان نتقدم اكثر وتعلم لغة التجميع من الضروري ان نلقي نظره على البنية الاساسية في Intel x86-64 CPU اذ لم تفهم هاذي البنية الـ machine instructions لن يكون لها اي معنى.

الـ Intel CPU family تصنف من von Neumann architecture machine الـ Von Neumann تحتوي على ثلاث وحدات اساسية هي : CPU , Memory & I/O . ترتبط جميع هاذي الناقلات عبر Data Bus ترتبط هاذي المكونات عبر system bus الذي يتكون من (address, data, and control buses).

لمعلومات اكثر تفصيلا :

<https://github.com/AhmaddF/Computer-Architecture/tree/main/Computer%20Architecture%20%26%20Organization/InputOutput>

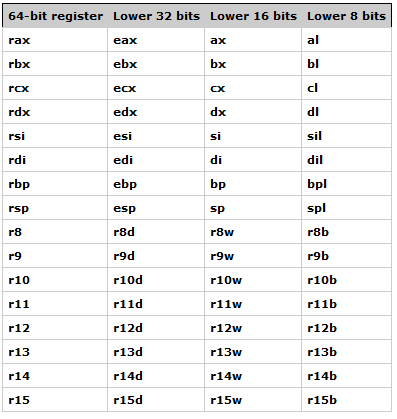


الصورة هاذي توضح هاذي العلاقة.

داخل الـ CPU يتم استخدام مواقع خاصة تعرف بالسجلات (Registers) لمعالجة البيانات يمكن تقسيم هاذي السجلات الى في x86-64 CPU الى اربع فئات : general-purpose registers و special-purpose application-accessible registers و segment registers و special-purpose kernel-mode registers .

الـ special-purpose kernel-mode registers مخصصة لكتابة انظمة التشغيل و debuggers و system-level tools اخرى. Intel family توفر العديد من السجلات للاغراض العامة للتطبيقات :

* Sixteen 64-bit registers that have the following names: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, and R15
* Sixteen 32-bit registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, and R15D
* Sixteen 16-bit registers: AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, and R15W
* Twenty 8-bit registers: AL, AH, BL, BH, CL, CH, DL, DH, DIL, SIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, and R15B



لسوء الحظ هاذي جميعها ليست سجلات مستقله بل مقسمه الى بتات. مثال سجل 64 ينقسم الى 32 وينقسم الى 16 وينقسم الى 8 . بمعنى انها ليست مستقله تغير سجل واحد مثال من سجل ذو 64 بت يمكن ان يغير ثلاثة سجلات مع بعض مثال قمت بتعديل rdx الـ edx,dx,dl ستتغير. اعتمادا على البيانات. احد الاخطاء الشائعة التي من الممكن الشخص يتعلم برمجة اسمبلي يصبح عنده (register value corruption) تلف قيمة السجل بسبب عدم الفهم الشامل للسجلات . بالاضافة الى السجلات العامة ايضا يوجد 8 سجلات خاصة الفاصلة العائمة (floating-point) يتم تنفيذها في وحدة floating-point unit (FPU) . شركة Intel قامت بتسمية هاذي السجلات ST(0) – ST(7) هكذا. على عكس general-purpose registers لا يمكن للبرنامج الوصول اليها مباشرة. بدلا من ذالك يتعامل البرنامج مع floating-point register file باعتباره eight-entry-deep stack مكون من 8 entries ويصل الى مدخل او مدخلين علوين. يبلغ كل سجل فاصلة عائمة الى 80-bit wide . ويحمل قيمة حقيقة ذات دقة ممتدة (extended precision).

على رغم Intel اضافت سجلات الفاصلة العائمة الاخرى الى x86-64 CPUs الان ان سجلات FPU لا تزال استخداما جيدا وشائع لانها تدعم 80-bit floating-point format .

في التسعينات قدمت Intel مجموعة من MMX register set وتعليمات لدعم single instruction والبيانات المتعددة (multiple data (SIMD) operations).

الـ MMX register عبارة عن مجموعة من ثمانية سجلات 64 بت تتراكب مع سجلات

ST(0) – ST(7) على FPU . الافضل لك الان الافضل ان تستخدم XMM registers (and instruction set) وترك FPU . الـ RFLAGS او الـ FLAGS هو سجل سجل 64 بت يحتوي على single-bit Boolean قيم. ثمانية من هاذي الـ FLAGS او (البتات) تهم مبرمجي التطبيقات الذين يكتبون في Assembly يوجد interrupt disable و 2 sign و auxiliary carry و parity و carry flags . وايضا هاذي الاعلام تتيح لك حالات لاختبار نتيجة الحساب السابقة.

يوجد الكثير التعارضات الذي ظهرت وتم حلها لكن لن اطرق اليها جميعها موجودة في مجلدات Intel .

**The Memory Subsystem**

**الـ memory subsystem يحتفظ ببيانات البرنامج مثل متغيرات البرنامج والثوابت وتعليمات الجهاز ومعلومات اخرى. يتم تنظيم الذاكرة عن طريق الخلايا (cells). يمكن لنظام دمج المعلومات والتعامل معها على هذا الاساس .**

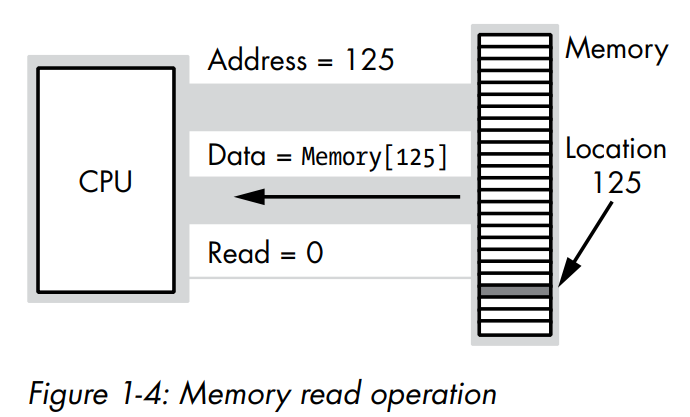
**لمعلومات اكثر تفصيلا :**

**x86-64 يدعم الذاكرة القابلة للعنونة بالبايت (byte-addressable memory) بمعنى ان وحدة الذاكرة الرئيسية هي عبارة عن بايتات. فكر بذاكرة كمصفوفة خطية من البايتات عنوان البايت الاول هو 0 وعنوان البايت الاخير هو 232 – 1 . بالنسبة للمعالج x86 مع الذاكرة بسعة 4 قيقا يعد تعريف المصفوفة كـ pseudo-Pascal array تقريبي جيد للذاكرة :**

**Memory: array[0….4294967295]**

**او كـ :**

**Byte Memory[4294967295];**

****

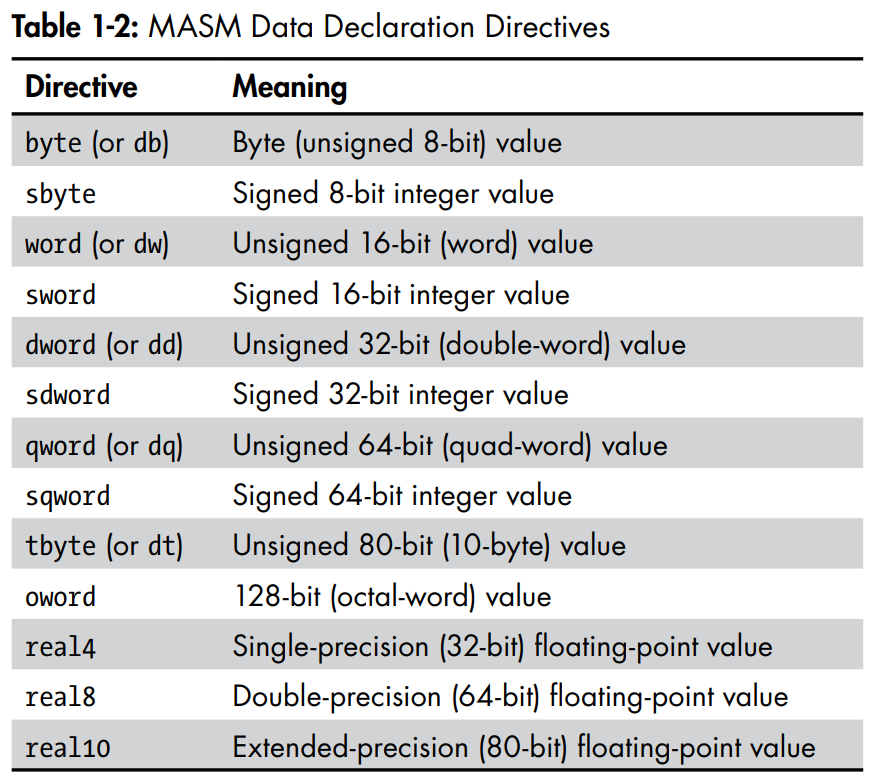
**على سبيل المثال لتنفيذ Memory[125] := 0 ; تضع الـ CPU قيمة 0 على Data Bus وتضع العنوان على address bus وتؤكد write line .**

**Declaring Memory Variables in MASM**

يمكن الاشارة الى عناوين الذاكرة باستخدام عناوين رقمية في لغة الاسمبلي الان ان القيام بذالك امر عرضه للخطأ وليس افضل شيء بدلا من ذالك يمكنك استخدام اسماء وتقم بتخزينها في متغيرات والاشارة عليها بمتغيرات استخدام اسماء المتغيرات يعد افضل بكثير من استخدام العناوين الرقمية للاشارة وايضا افضل في الكتابة والقراءة والصيانة. لأنشاء متغيرات بيانات يجب عليك وضعها في قسم البيانات data section في ملف المصدر في MASM يخبر هذا التوجيه MASM كافة العبارات التالية هي بيانات وسيتم تعرف عليها في البرنامج كـ Data . في .data section تسمح MASM لك بتعريف كائنات كـ متغيرات باستخدام (data declaration directives) الشكل الاساسي لتعريف البيانات هو :

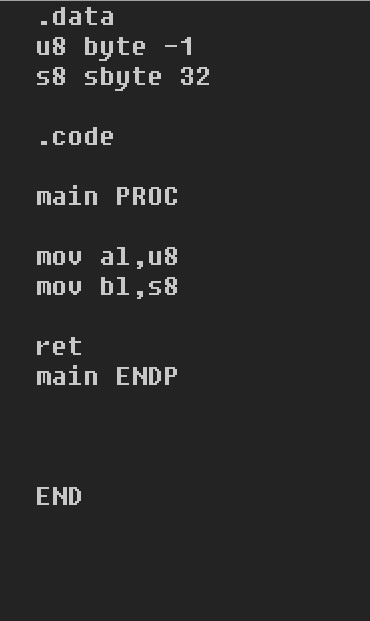
**`label directive ?`**

حيث الـ label هي معرف قانوني في MASM وهو احد التوجيهات (directives) التي تظهر في جدول هذا :



علامة الاستفهام (?) تخبر MASM ان الكائنات لن يكون له قيمة صريحة عند تحميل البرنامج في الذاكرة بمعنى انها صفر مبدئيا. اذا كنت ترغب بوضع قيمة صريحة فاستبدل علامة الاستفهام مثل :

**Initvalue sqword 1;**



MASM بتجاهل بغالب الاحيان بتجاهل بادئة S الي هي تعبر عن Signed او Unsigned . بل هي تعتمد اكثر على الـ machine instructions تفرق بين Signed or Unsigned لا يهتم اذا كان المتغير يحمل قيمة Signed او لا في هذا الكود يظهر ان لا يوجد مشكله ويمكن ان يعمل هذا الكود ويحمل قيمة Signed ويمكن عرض هذا من خلال خبرتك في الـ Software Debugging .

من الممكن حجز تخزين لقيم بيانات متعددة (multiple data) في single data declaration directive . الـ string يعد multi-valued data type يمكنك انشاء سلسلة من الاحرف منتهية بقيمة خالية في الذاكرة باستخدام byte directive على سبيل المثال :



**Declaring (Named) Constants in MASM**

MASM يسمح لك باعلان constants باستخدام = . الـ constant هو symbolic name (identifier) يقرنه MASM بقيمة. في كل مكان يظهر الـ symbol في البرنامج مباشرة يقوم MASM باستبدال قيمة هذا الرمز بالرمز المعرف. مثال :

label = expression -> dataSize = 256 or dataSize equ 256 (equates);

**Some Basic Machine Instructions**

الـ x86-64 CPU family توفر ما يزيد عن مئات الى الاف من machine instruction اعتمادا على كيفية تحديد machine instruction . لكن معظم الـ assembly language programs تستخدم بين 30 – 50 تعليمة وهنا سنشرح كتابة العديد من البرامج ذات معنى القليل سنوفر مجموعة صغيرة من تعليمات الـ machine instructions حتى تتمكن من البدء في كتابة البرنامج MASM بسيط.

**The mov Instruction**

مبدئيا بدون اي شك تعليمة mov هي التعليمة الاكثر استخداما في برامج الاسمبلي في اي برنامج نموذجي ما بين 25% - 40% من التعليمات عبارة عن mov . من اسمها يمكنك فهمها تقوم بنقل البيانات من موقع الى اخر كيفية بناء هاذي الجملة :

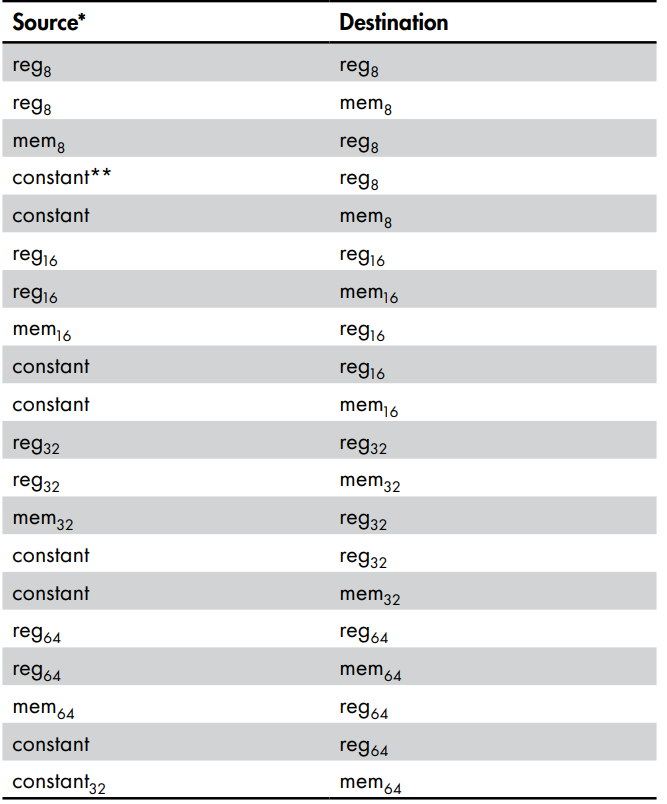
**mov destination\_operand, source\_operand**

يمكن ان يكون الـ source\_operand عبارة عن سجل للاغراض العامة (general-purpose) او متغير في الذاكرة (memory variable) او ثابت (constant) . وقد يكون الـ destination\_operand عبارة عن سجل او متغير ذاكرة. في لغات مثل C/++ تكون mov instruction مكافئة لهذا البيان التالي :

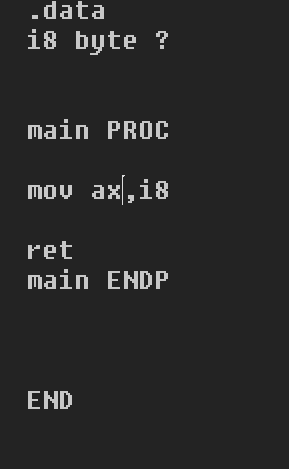
**destination\_operand = source\_operand**

يجب ان تكون الـ mov instruction’s operands يجب ان يكون بنفس الحجم. اي انه يمكنك نقل البيانات بين byte (8-bit) objects او word (16-bit) objects او double-word (32-bit) او quad-word (64-bit) objects .

**يوضح هذا الجدول ما اقصد به**



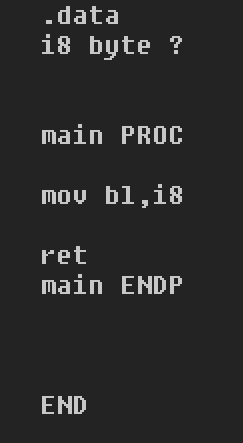
MASM يفرض بعض التحقيقات على المعاملات التعليمات. يجب ان يتوافق حجم المعاملات التعليمات. على سبيل المثال هذا خطأ :



المشكلة هي انك تحاول عمل Load لـ 8-bit variable الى 16-bit register . نظرا لان احجامها غير متوافقة يفترض MASM انه خطأ منطقي في البرنامج ويقوم بالابلاغ عن المشكلة :



الصح :



**The add and sub Instructions**

تعليمات الـ add & sub تقوم بعمل عملية الاضافة(add) او الطرح (subtract) لمعاملين الـ syntax :

**add destination\_operand, source\_operand -> destination\_operand = destination\_operand + source\_operand**

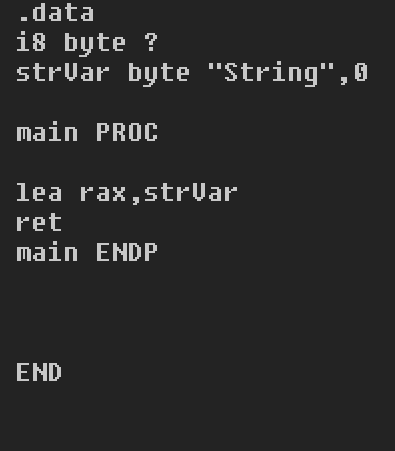
**sub destination\_operand, source\_operand -> destination\_operand = destination\_operand - source\_operand**

**The lea Instruction**

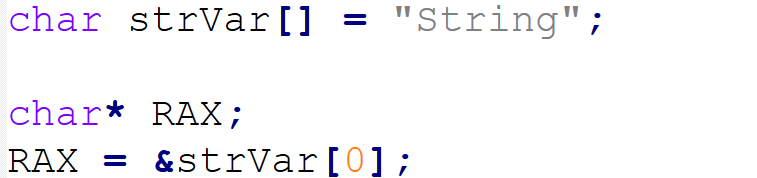
بعض الاحيان تحتاج الى عمل load لعنوان معين على سبيل المثال متغير وعلى سبيل المثال ايضا موجود في سجل بدلا من القيمة نفسها. يمكنك استخدام lea (load effective address) لهذا الغرض. الـ syntax :

**lea reg64, memory\_var**

الـ reg64 هو سجل general-purpose 64-bit و memory\_var (ليس من الضروري ان يكون المتغير qword variable كما هو الحال مع sub,add & mov). كل متغير (variable) له عنوان في الذاكرة مرتبط به ويكون هذا العنوان في بنيات x64 دائما 64-bit مثال بسيط :



هذا المثال يقوم بتحميل في الـ RAX Register عنوان اول حرف في strVar . في تمثيل لغة C++ :



**The call and ret Instructions and MASM Procedures**

لأجراء function calls وبالاضافة كتابة وظائفك البسيطة تحتاج الى تعليمات call & ret . الـ ret instruction نفس الـ return الي موجودة في C/++ فهي تعيد التحكم من assembly language procedure (الـ assembly language functions تسمى procedures في لغة اسمبلي). في هذا الشرح سنستخدم ret لوحدها لا تحتوي على معاملات :

**ret**

الـ call instruction تكون هكذا :

**call proc\_name**

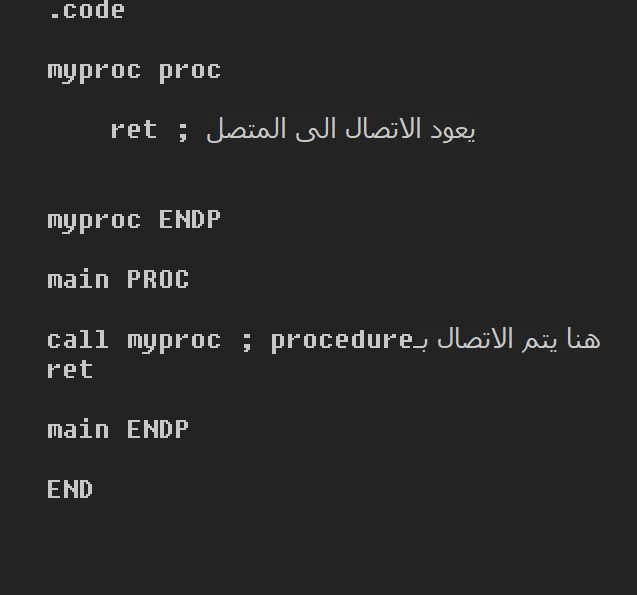
الـ proc\_name هو اسم procedure التي تريد عمل لها call . بعد كتابة القليل من البرامج الجدا بسيطة MASM procedure يتكون من هذا السطر :

**proc\_name proc**

**;any**

**proc\_name endp**

هكذا تبدأ وهكذا تنتهي من عند endp وفي نصف يمكنك كتابة التعليمات البرمجية التي تخص هذا procedures .



الكود هذا يوضح MASM procedure .

**Calling C/C++ Procedures**

على الرغم ان كتابة procedure واستدعاءها امر مفيد للغاية. فان سبب ادخال الـ procedures في هذه المرحلة ليس السماح لك بكتابة الـ procedures الخاص بك بل لاعطائك القدرة على استدعاء الـ procedures المكتوبة بـ C/C++ . تعد كتابة الـ procedures الخاصة بك لتحويل البيانات واخراجها الى الـ Console لحد الان هي تعتبر معقدة قليلا. بدلا من ذالك يمكنك استدعاء دالة printf() الموجودة في لغة C/++ لانتاج مخرجات البرنامج والتحقق ان برنامجك تقوم بالفعل بشيء عند تشغيلها . لسوء الحظ اذا قمت باستدعاء printf() في لغة اسمبلي الخاص بك دون توفير printf() procedure مجمع MASM سيعطي خطأ ان الرمز غير محدد (undefined symbol). لأستدعاء اجراء خارج الـ Source Code تحتاج الى استخدام " externdef directive " بناء الجملة على النحو التالي :

**externdef symbol:type**

الـ symbol هذا هو رمز خارجي الذي تريد تعريفه والـ type هو نوع الـ Symbol والذي سيكون (proc for external procedure definitions) لتحديد رمز printf() هكذا سيكون :

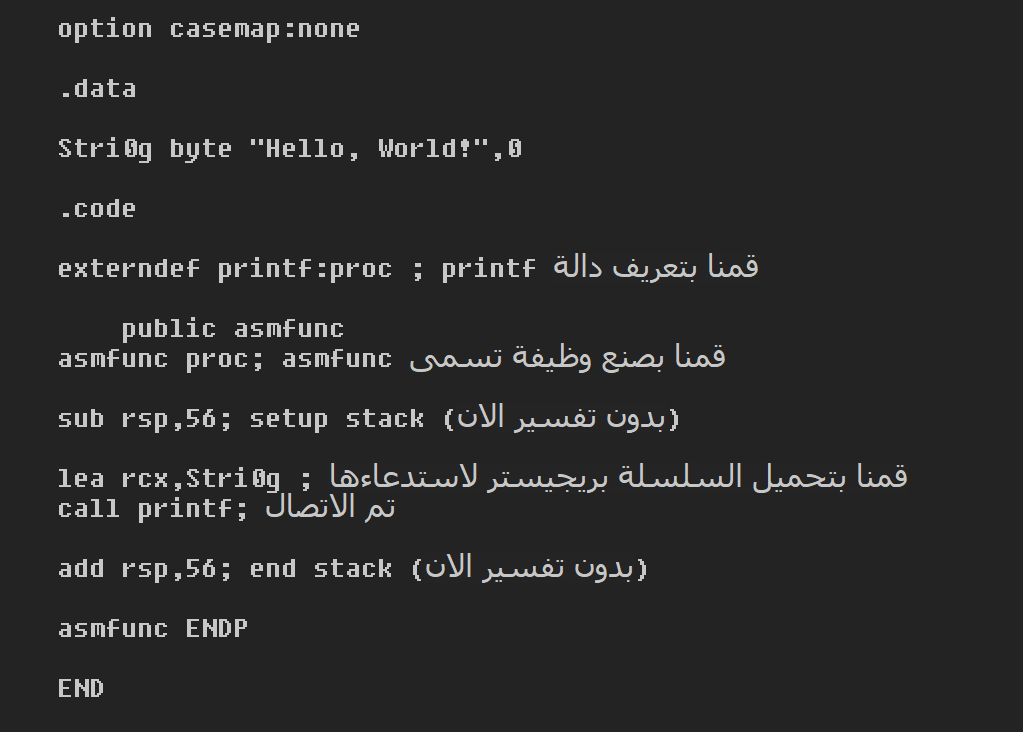
**externdef printf:proc**

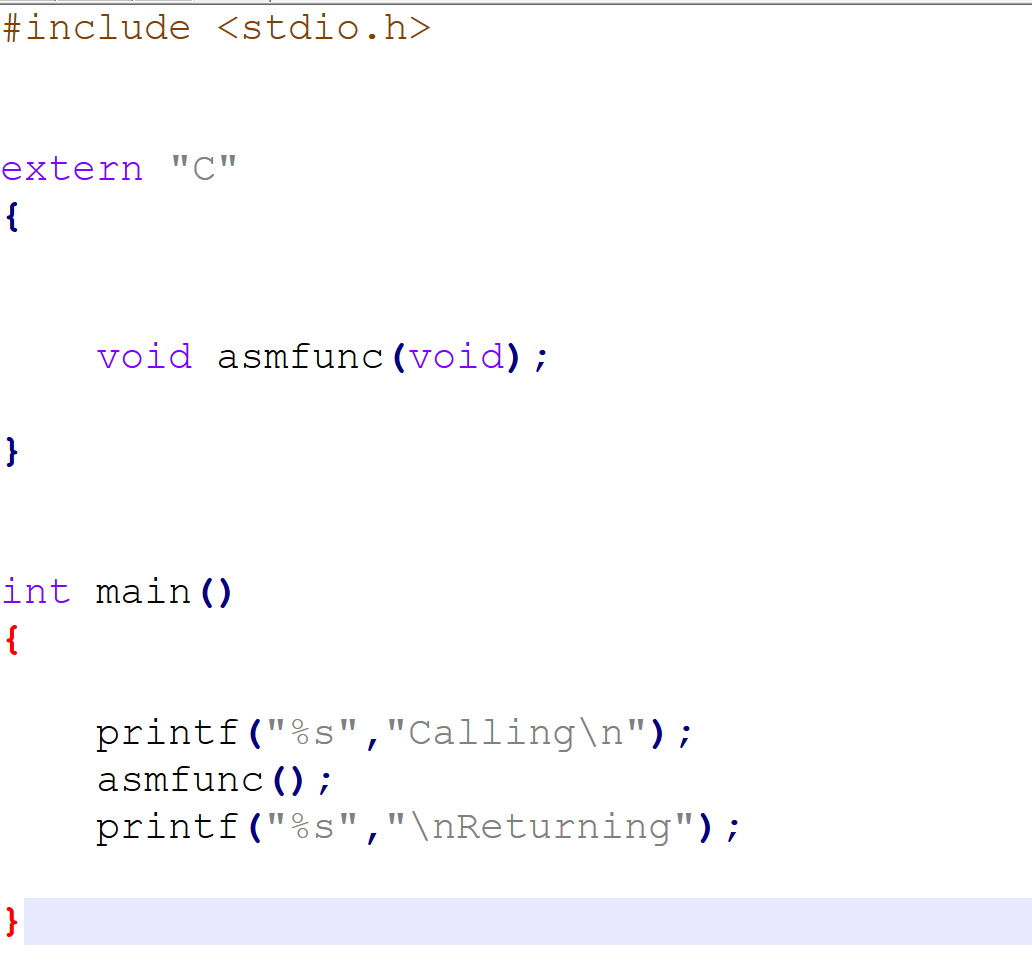
الـ externdef لا يتيح لك تحديد معلمات لتمريرها الى اجراء printf() ولا توفير تعليمات لتحديد الـ (parameters) . بدلا من ذالك يمكنك تمرير ما يصل الى 4 معلمات الى دالة printf في سجلات RCX,RDX,R8 & R9 . تتطلب دالة printf() ان تكون المعلمة الاولى هو الـ format string .

لذالك يجب تحميل(Load) RCX بعنوان سلسلة منتهية بصفر قبل استدعاء printf() . اذا كانت سلسلة التنسيق تحتوي على احد محددات تنسيق (على سبيل المثال مثل %s or %d) فيجب عليك تمرير قيم المعلمات المناسبة في RDX , R8 & R9 .

**Hello, World!**

الان لدينا المعلومات الكافية لكتابة Hello, World! .

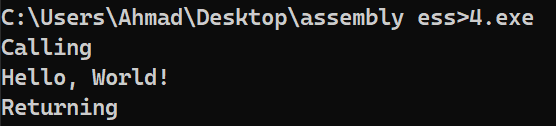


=

عملية التجميع :

**1 - ml64 /c 3.asm**

**2 - cl 4.cpp 3.obj**

**3 – output 4.exe :** ****

**يمكنك اضافة Stri0g byte "Hello, World!",10,0 ايضا 10 لعمل new line لكن انا قمت بعمل new line عن طريق printf("%s","\nReturning"); .**

**Returning Function Results in Assembly Language**

شرحنا كيفية تمرير تعليمات الى procedure مكتوب بلغة اسمبلي هنا سنصف عملية معاكسة : ارجاع قيمة الى التعليمات البرمجية التي استدعت احد procedures الخاص بك. في pure assembly language حيث يستدعي احد الاجراءات اجراء اخر يعد تمرير المعلمات (passing parameters) ونتائج الوظائف المرتجعة (returning function) عبارة عن اتفاقية تتشاركها اجراءات المتصل والمستدعي مع بعض. يمكن للمتسدعى (calle) (الاجراء الذي يتم استدعاؤه) او المتصل (الاجراء الذي يقوم بالاتصال) (caller) اختيار مكان ظهور النتائج. من وجهة نظر الـ calle يحدد الاجراء الذي يعيد القيمة المكان الذي يمكن للمتصل العثور على نتيجة الوظيفة فيه ومن يستدعي تلك الوظيفة يجب ان يحترم هذا الاختيار . على سبيل المثال اذا قام احد الـ procedure بارجاع قيمة في سجل XMM0 سجل شائع لحفظ قيمة الـ Return الفاصلة العائمة (floating-point). فيجب على من يستدعي هذا الاجراء ان يتوقع العثور على النتيجة في XMM0 . قد يؤدي اجراء اخر حفظ النتيجة في سجل RBX . من وجهة نظر الـ caller’s يتم عكس الاختيار. تتوقع التعليمات البرمجية الموجودة ان تقوم الوظيفة بارجاع نتيجتها في موقع معين ويجب ان تحترم الوظيفة التي يتم استدعاؤها . في النهاية من المهم ان تعرف عندما تكتب اسمبلي فان طريقة تمرير البيانات الى الاجراء . المره الوحيدة التي يجب ان تقلق هو بشان الالتزام بـ ABI هي عندما تتصل برمز خارج عن سيطرتك اذا كان هذا الرمز الخارجي يتصل بكودك. سنغطي هنا كتابة لغة التجميع ضمن Microsoft Windows هو كود لغة الاسمبلي الذي يتفاعل مع MSVC . لذالك عند التعامل مع التعليمات البرمجية الخارجية (Windows and C++ code) يجب عليك استخدام Windows/MSVC ABI . ينص Microsoft ABI ايضا على ان الوظائف (الاجراءات) ترجع قيما صحيحة ومؤشرا (pointer) تتناسب مع X64 في سجل RAX . لذالك اذا كانت بعض التعليمات C++ تتوقع ان يقوم اجراء الخاص بك بارجاع قيمة صحيحة فيمكنك تحميل (Load) النتيجة العدد الصحيح الى RAX مباشرة قبل العودة (returning) الى الاجراء الخاص بك.

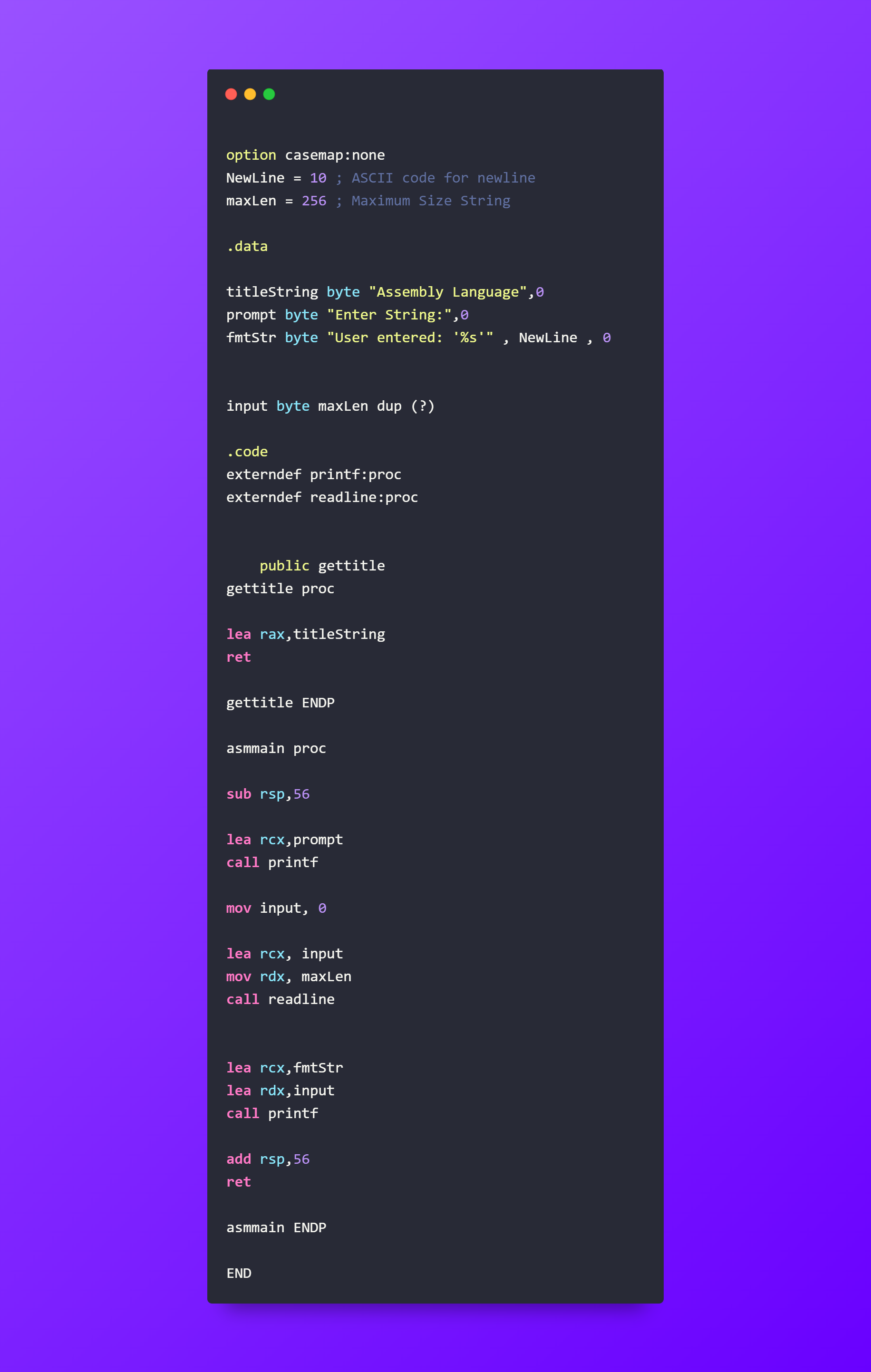


اول شيء الـ try..catch تلتقط اي استثناءات ينشئها كود اسمبلي بحيث تحصل على نوع الاشارة اذا تم احباط هذا البرنامج . هذا الكود يوفر مفاهيم جديدة ابرزها عن الـ return . تقوم دالة getitle بارجاع pointer الى سلسلة سيطبعها كود C .

في قسم الـ .data ستجد هذا :

**input byte maxLen dup (?)**

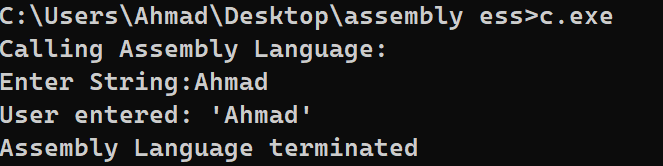
الـ maxLen dup (?) تخبر MASM يفعل تكرار (duplicate) بعدد مرات ? (اي بايت غير مهيأ).

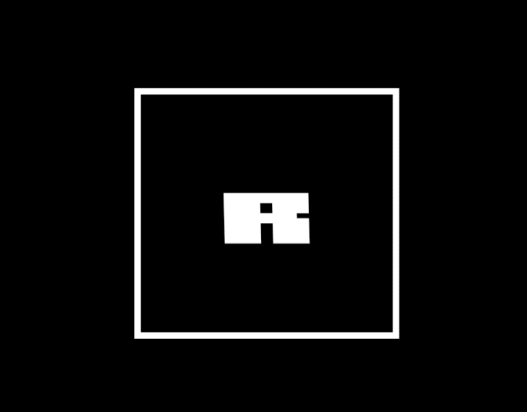


لنقوم بعمل compile :

**ml64 /c new.asm**

**cl /EHa c.cpp new.obj**

****

****

Twitter : <https://twitter.com/dr_retkit>

YouTube : https://www.youtube.com/@retkit1823