مقدمة عن C

::مفاهيم وأساسيات::

بما أن C# تم إنشائها خصيصاً لتمثل إحدى لغات الـ .NET. فسنحتاج لمعرفة ماهية .NET أولاً.

ماهي الـ .NET.؟

كثيراً ما يتردد على مسامعنا كلمة .NET. سواء في الإنترنت أو الصحف والمجلات التقنية أو حتى في الجامعات! فما هي الـ.NET. ؟

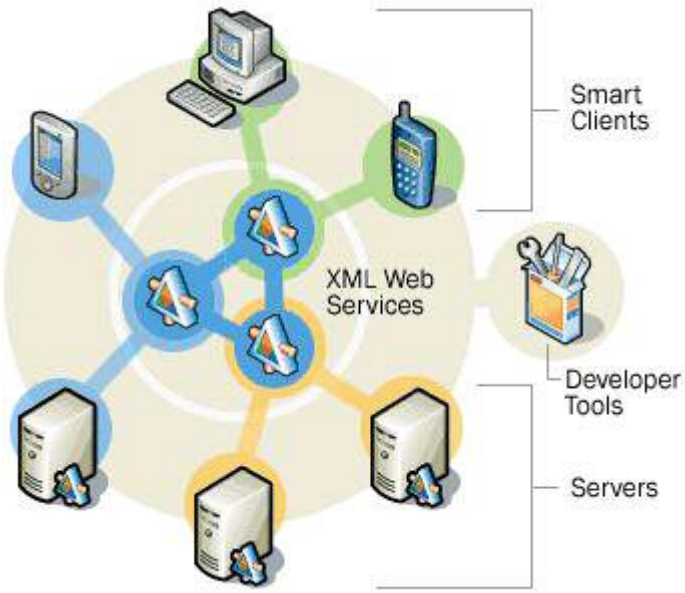
هل هو برنامج ضخم يتم شراءه؟ أم لغة برمجة؟ أم خدمة يتم الاشتراك بها؟!

الهدف الذي أرادات شركة مايكروسوفت تحقيقه من وراء الـ .NET. هو أن تجعل أجهزة الكومبيوتر و برامجه و الأجهزة الأخرى كالطابعات والماسحات الضوئية ومواقع الويب تعمل معاً وتوفر للمستخدم حلولاً أقوى لمشكلاته، بدلاً من أن تتصل هذه الأجهزة و البرامج بالإنترنت بشكل منفصل.

فالـ .NET. عبارة عن مجموعة من تقنيات برمجية تمثل البنية التحتية والتي تسمى بـ .NET FRAMEWORK. والتي تجعل المعلومات، المستخدمين، التطبيقات، ومختلف الأجهزة، والأنظمة تتصل ببعضها البعض من خلال ما يسمى بخدمات الويبweb services.

Web Services:

هي عبارة عن برامج يتم وضعها على الجهاز الخادم server لتعمل من خلال الإنترنت، هذه البرامج تم بناءها بواسطة لغة XML لتسمح لمختلف التطبيقات بتبادل المعلومات بغض النظر عن اللغة التي كُتبت بها هذه التطبيقات، أو أي نظام تشغيل تعمل عليه، أو حتى نوعية الأجهزة التي توجد عليها هذه التطبيقات.



هذه الصورة مأخوذة من موقع مايكروسوفت

.NET FRAMEWORK:

يحتوي على مكتبة NET. وهذه المكتبة عبارة عن أكواد جاهزة مصنفة تصنيفات مختلفة فبعضها خاص ببرمجة الشبكات وبعضها خاص ببرمجة تطبيقات الويندوز، والبعض الآخر مختص ببرمجة مواقع الويب.

ويحتوي الـ NET FRAMEWORK. أيضاً على ما يسمى بـ Common Type System(CTS) وهي عبارة عن الأنواع المتاحة للغات الـ NET. لتمثيل البيانات في ذاكرة الكومبيوتر.

ويحتوي هذا الـ FRAMEWORK أيضاً على Common Language Runtime(CLR) وهو عبارة عن محرك يقوم بتنفيذ جميع البرامج المكتوبة بأحد لغات NET.

لغات NET.:

- Visual basic.net
- Visual C#.net
- Visual c++.net
- Visual j#.net
- Jscript.net
- Cobol.net
- Perl.net
- Eiffel.net
- Python.net
- Pascal.net
- Mercury.net
- Mondrian.net
- Oberon.net
- Salford FTN95.net

C#:

C# كما ذكرنا في البداية أنها إحدى لغات NET. من إنتاج شركة مايكروسوفت، وتنطق "سي شارب".

تم إنشائها بواسطة فريق عمل بقيادة أندرس هيجلزبرج، وقدر ركز هذا الفريق في بناء هذه اللغة على نقاط القوة في اللغات الأخرى وتجنب نقاط الضعف فيها بالإضافة إلى بعض المميزات الجديدة التي أضيفت إلى هذه اللغة! لذلك فهي تتميز بالقوة والمرونة، وكباقي لغات NET. فيمكن استخدامها لإنشاء تطبيقات الويندوز، وتطبيقات الويب و أجهزة الترجمة و غيرها. وتعتمد البرمجة بسي شارب على مفهوم البرمجة باستخدام الكائنات بصورة كاملة!

::نقاط مهمة في #C::

كل لغة برمجة لها بعض القيود أو دعنا نسميها أساليب لكتابة البرامج بها. و لهذا فنحن بحاجة لوضع هذه النقاط في الاعتبار عند البرمجة، حتى لا نقع في أخطاء بسيطة قد تعطل عملنا حتى نكتشفها!!

عند كتابة برامج C# نأخذ في الاعتبار أن :

- **لغة C# لغة حساسة لحالة الأحرف، أي أن الكلمة Console تختلف عن الكلمة console.**
- **يجب أن ينتهي كل سطر برمجي بعلامة الفاصلة المنقوطة (؛).**
- **لإضافة تعليقات للكود نستخدم الرمز // لإضافة تعليقات في سطر واحد، بينما نستخدم الرمز /* */ ونضع التعليقات بينهما لإضافة تعليقات في أكثر من سطر. مثال:**

```
//this line to display a welcome message on the screen

Console.WriteLine("welcome to the first C# Program");
```

- عند تنفيذ برامج #C فإنه يتم تجاهل المسافات البيضاء. و المسافات البيضاء هي الأسطر الفارغة التي قد نستخدمها في تنسيق شكل البرنامج لتسهل قراءته !

أنواع البرامج في #C:

قبل أن ننتقل لنتعرف بشكل أعمق على #C، يهمني أن تعرف أنواع البرامج التي يمكن برمجتها مستخدماً #C، حيث سنتعرض لهذه الأنواع خلال الدروس القادمة بإذن الله:

- برامج نصية: وهي البرامج التي تعرض نتائجها في صورة نصية على سطر الأوامر (الدوس) تماماً كما في المثال السابق.

- برامج الويندوز: هي برامج ذات واجهة رسومية، كما في العديد من البرامج التي تستخدمها كبرنامج وورد مثلاً.
- خدمات الويب: هي برامج يمكن استدعاؤها عبر الويب. مثل MSN Messenger للمراسلة الفورية عبر الإنترنت.
- نماذج ويب: وهي صفحات ويب يتم معالجتها على الجهاز الخادم.

::البرنامج الأول في C#::

هذا الدرس سيكون المدخل الأول لتعلم C#، حيث سنتعلم فيه كتابة أول برنامج (برنامج بسيط للغاية) الغرض منه هو وضع أقدامنا على بداية الطريق..

قد يدور في ذهنك الآن، كيف وأين أكتب برامج C# ؟!

سؤال منطقي، بكل سهولة يمكننا كتابة برامج C# باستخدام برنامج المفكرة وحفظها بالامتداد cs. وقد ذكرنا في الدرس السابق أن NETFRAMEWORK. يحتوي على محرك لتنفيذ برامج NET. بالإضافة إلى مكتبة برامج NET. ، فإذاً كل ما نحتاج إليه لتنفيذ برامج C# هي NETFRAMEWORK. وستجدها في موقع مايكروسوفت www.microsoft.com

ولكن شركة مايكروسوفت طرحت أداة تطوير جديدة تسمى Visual Studio.NET و اختصارها VS.NET وهي عبارة عن بيئة تطوير متكاملة IDE تحتوي على التالي:

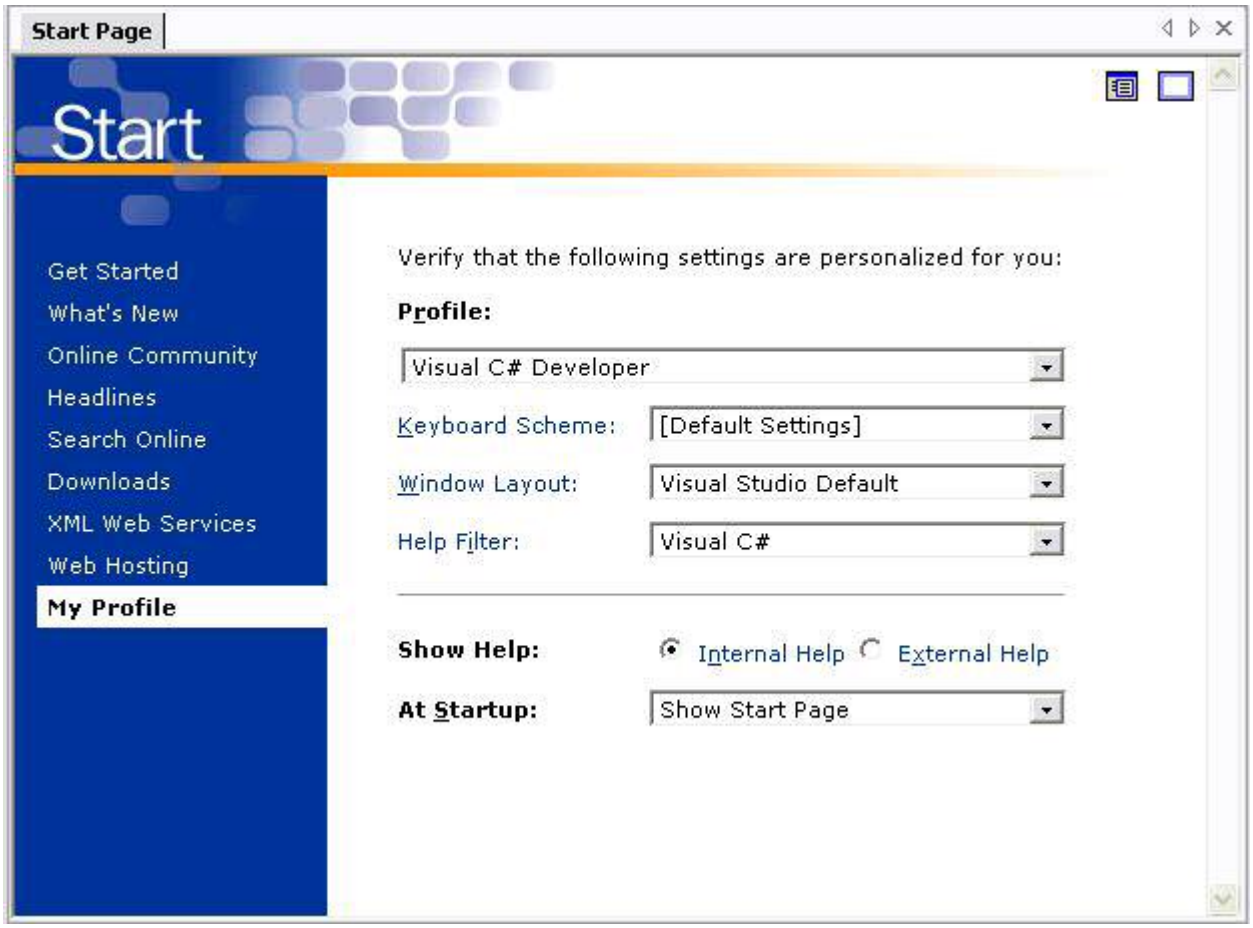
- **محرر نصوص، لكتابة برامج .NET.**
- **ترجمة وتنفيذ برامج .NET.**
- **يمكنك من تصميم الواجهات والنماذج بسهولة .**
- **يتمتع بخاصية ترقيم أسطر لبرنامجك.**
- **يمكنك تصفح الإنترنت من خلال برنامج انترنت اكسبلورر الموجود ضمن بيئة VS.NET.**
- **بالإضافة إلى أن واجهته مصورة و سهلة الاستخدام.**

لذلك فمن الأفضل لنا استخدام VS.NET لكتابة وتنفيذ برامجنا.

لنبدأ إذن:

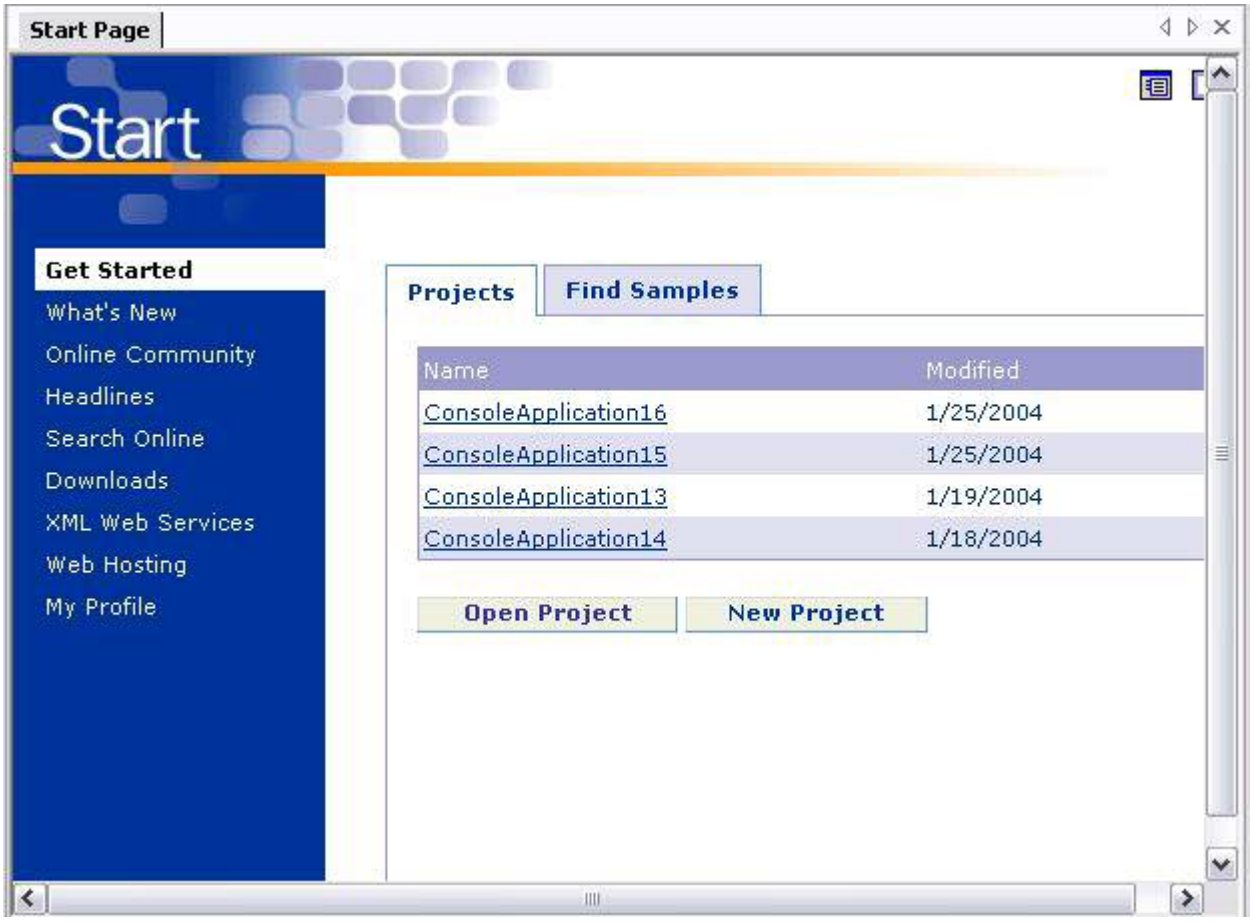
لنفتح برنامج VS.NET ونتبع الخطوات التالية:

١. إذا كانت هذه المرة الأولى التي تفتح فيها البرنامج، فاختر My Profile، وذلك لتحديد لغة .NET. التي تريدها كما في الشكل التالي:

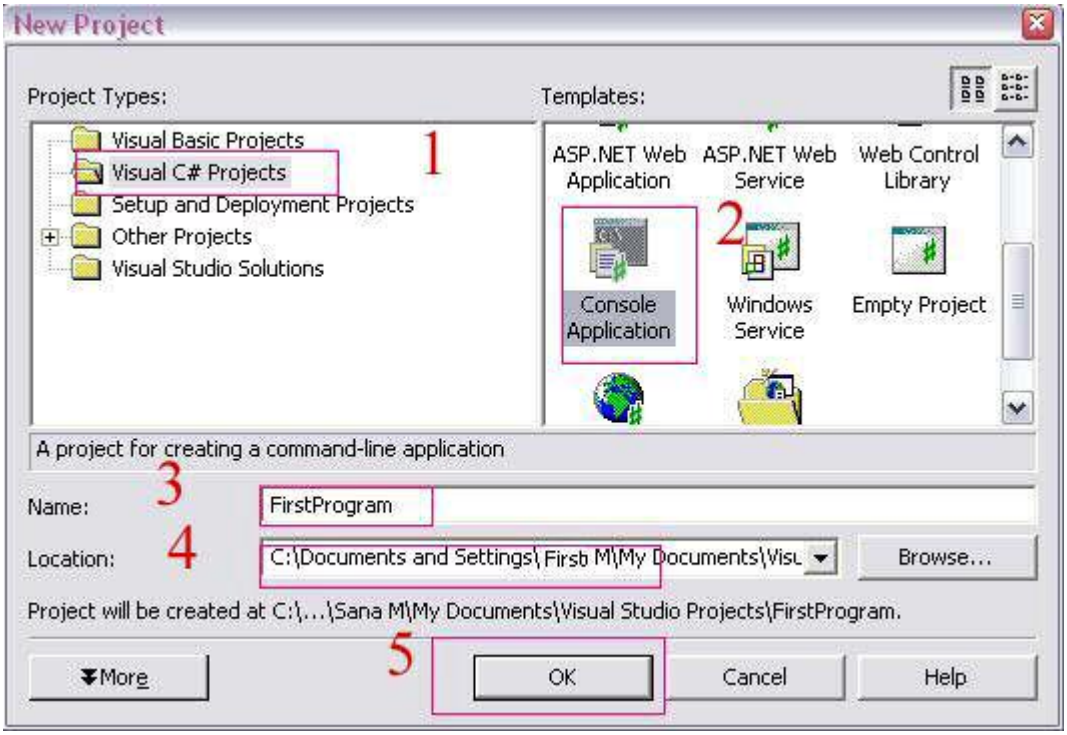


ثم اختر Visual C# Developer من القائمة المنسدلة.

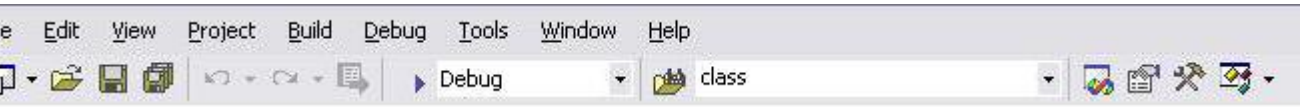
٢. ثم مرة أخرى من القائمة الموجودة على اليسار اختر Get Started، ثم اختر زر New Project كما في الشكل التالي:



٣. في الخطوة الأولى اختر Visual C# من القائمة الموجودة على اليسار، وفي الخطوة الثانية اختر Console Window من النافذة الموجودة على اليمين، وفي الخطوة الثالثة قم بتسمية المشروع بأي اسم تريد هنا قمنا بتسميته FirstProgram ولك الحرية المطلقة في التسمية، وفي الخطوة الرابعة اخترنا ملف لحفظ المشروع، في هذا المثال حفظنا المشروع في الملف الافتراضي لبرنامج VS.NET، وفي الخطوة الخامسة والأخيرة نضغط على الزر OK . كما يبين الشكل التالي:

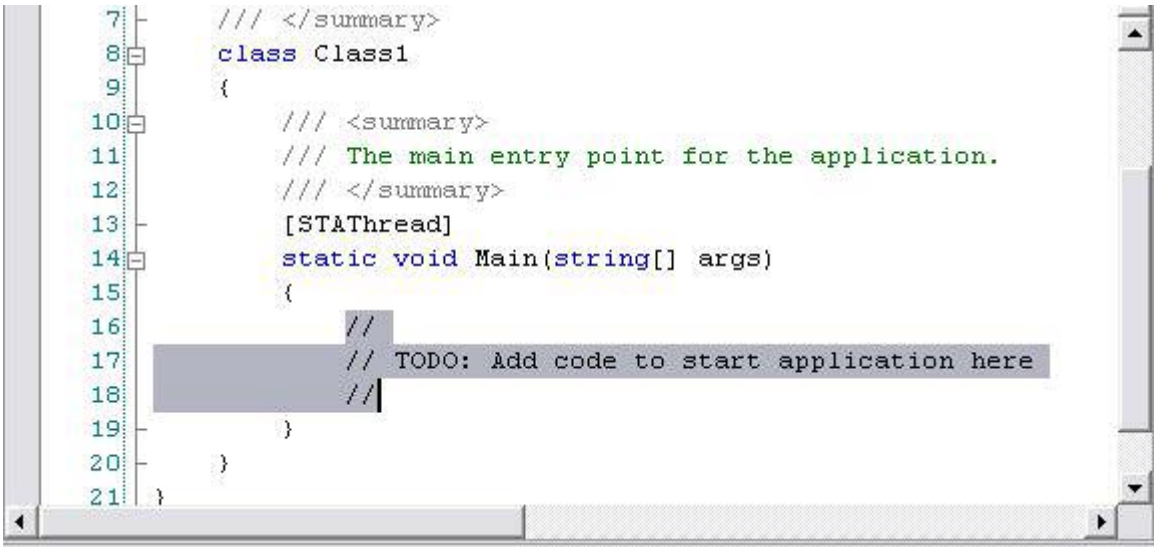


٤. بعد الضغط على OK ستفتح لنا نافذة البرنامج وسترى فيها محرر النصوص مكتوب بداخله أوامر C# (لا تقلق لست ملزماً بفهم هذه الأوامر في الوقت الحالي)، و في الجهة العليا ستجد شريط القوائم والأدوات كما هو مبين في الشكل :



٥. أما في الجهة السفلى فستجد نافذة المخرجات، وهي النافذة التي تبين لك ما إذا كان تنفيذ برنامجك قد تم بنجاح أم أن هناك أخطاء في البرنامج !

٦. هذه نظرة سريعة على الواجهة المرئية لبرنامج VS.NET، لنكتب أول برنامج لنا! في محرر النصوص، امسح الجزء المضلل في الشكل التالي:

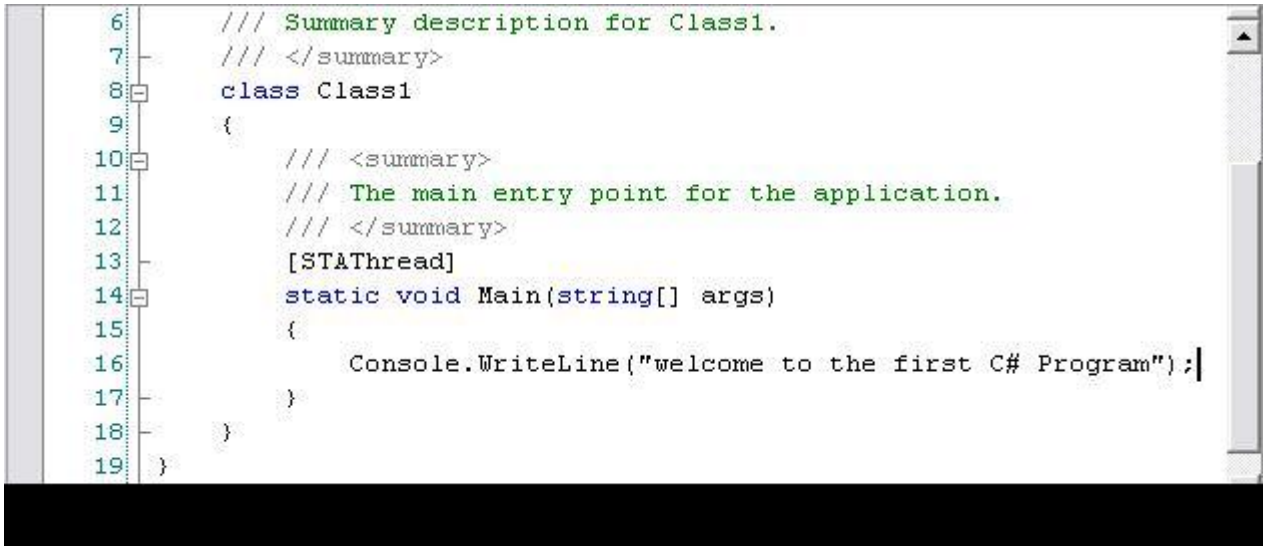


```
7     /// </summary>
8     class Class1
9     {
10        /// <summary>
11        /// The main entry point for the application.
12        /// </summary>
13        [STAThread]
14        static void Main(string[] args)
15        {
16            //
17            // TODO: Add code to start application here
18            //
19        }
20    }
21 }
```

٧. ثم اكتب السطر التالي بدلاً منه:

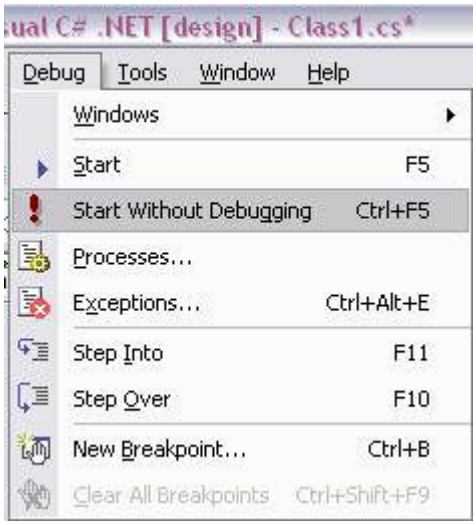
```
Console.WriteLine("welcome to the first C# Program");
```

كما في الشكل التالي:



السطر السابق يؤدي إلى طباعة الجملة الموجودة بين علامتي التنصيص "" على شاشة الدوس السوداء.

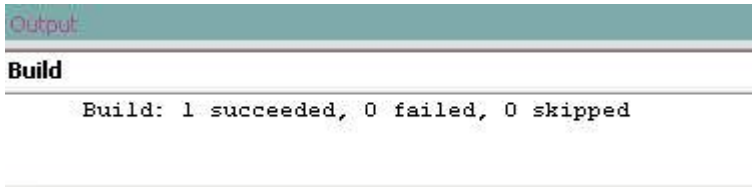
٨. لنشاهد النتيجة علينا تنفيذ الكود أولاً، من شريط القوائم الموجود أعلى محرر النصوص اختر Debug | Start without debugging (أو اضغط على المفتاحين ctrl+F5) كما في الشكل:



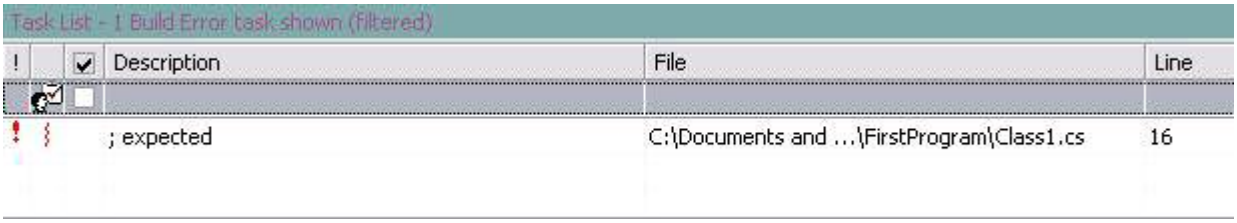
بعد ذلك سترى شاشة الدوس السوداء قد ظهرت ومكتوب عليها النص السابق كما في الشكل التالي:

اضغط على أي مفتاح لإخفاء هذه الشاشة.

٩. هل تلاحظ أي تغير في نافذة المخرجات؟ بالطبع! فأثناء تنفيذ البرنامج، ظهرت على هذه النافذة أن البرنامج قد نفذ بنجاح ولا يوجد أخطاء. كما في الشكل التالي:

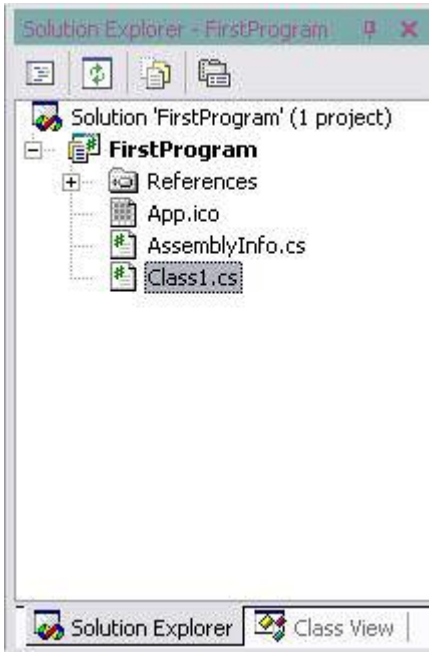


أنظر ما سيحدث في حال كتابة أخطاء في الكود السابق، كعدم كتابة علامة الفاصلة المنقوطة في آخر السطر السابق:

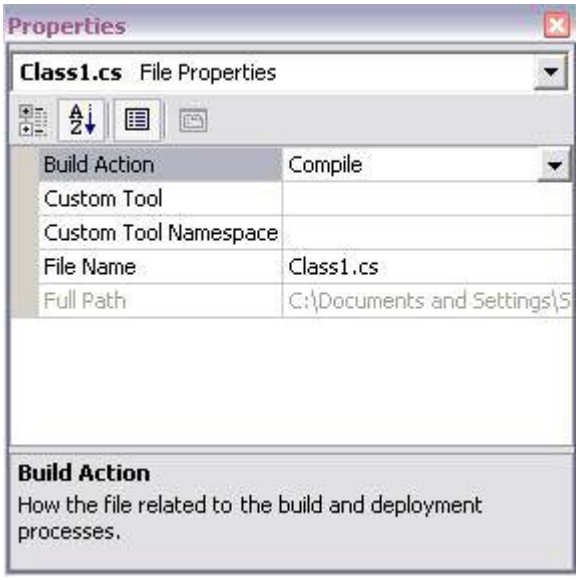


سيظهر الخطأ والسطر الذي يوجد به هذا الخطاء، ولن تظهر النتيجة على الشاشة السوداء حتى يتم تعديل هذا الخطأ.

١٠. **إذا أردت معرفة الملفات التي يتضمنها هذا المشروع، فمن شريط الأدوات الموجود أعلى المحرر، اختر الرمز**  **وستفتح لك نافذة Solution Explorer يمين محرر النصوص، سترى في هذه النافذة أربع ملفات References و App.ico و assemplyInfo.cs و class1.cs. ما يهمنا هو class1.cs حيث أنه الملف الذي اضفنا له سطر C# السابق. وهو يحمل الإمتداد cs. وهو امتداد جميع ملفات البرامج المكتوبة بلغة C#. كما في الشكل:**



١١. عند اختيار أي ملف من هذه الملفات قم بالنقر على الرمز في شريط الأدوات، وسيفتح لك نافذة الخصائص Properties وستظهر خصائص هذا الملف كالاسم والمسار الذي يوجد به هذا الملف. كما في الشكل:



حسناً، كل ما قمنا به حتى الآن هو التعرف على بيئة التطوير VS.NET و كتابة أول برنامج لنا مستفيدين من المميزات السهلة التي توفرها هذه البيئة.

وقبل أن أتركك تعبث في واجهة VS.NET، أود أن أضيف معلومات مبسطة حول الكيفية التي تمت بها تنفيذ برنامجنا السابق.

- **في البداية يوجد لدينا ملف به كود مكتوب بأحد لغات NET. وهو C# في هذه الحالة.**

§ بعد ذلك تتم ترجمة الكود إلى لغة وسيطة ليست من لغات ذات المستوى الأعلى ولا من اللغات ذات المستوى الأدنى، وتسمى هذه اللغة MSIL وهي اختصار لـ Microsoft Intermediate Language حيث يصبح هذا الكود غير معتمد على جهاز معين ولا نظام تشغيل معين.

§ وباستخدام نوع من المترجمات يأتي مع NETFRAMEWORK. يسمى JIT Compiler وهو اختصار لـ Just In Time Compiler، وهو حسب ما يدل عليه اسمه مترجم لحظي أي يستخدم في كل مرة أريد تشغيل البرنامج وتحويله إلى لغة يفهمها جهاز الكومبيوتر وينفذها، وهذه اللغة تسمى Native Code.

ولذلك فمن مميزات تطبيقات NET. أنها غير معتمده على نظام تشغيل أو جهاز معين! فقط نستخدم JIT Compiler مناسب لنظام التشغيل والجهاز لدي ومن ثم يمكنني تشغيل أي تطبيق من تطبيقات NET.!

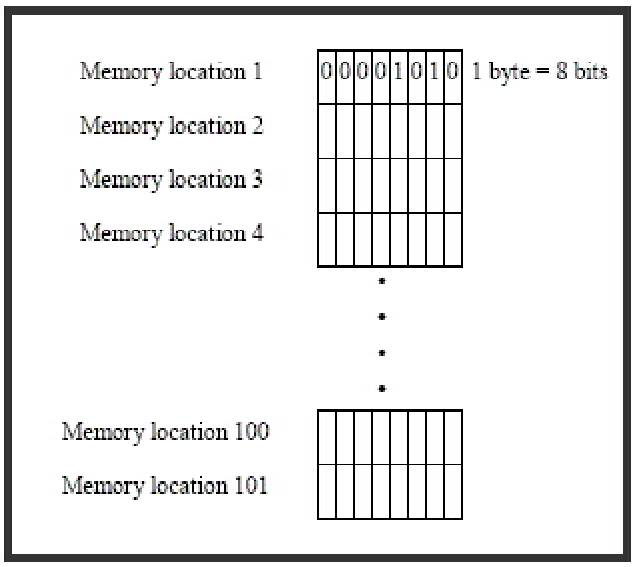
وبذلك يمكنني استدعاء برنامج مكتوب بلغة Visual Basic.NET من برنامج مكتوب بلغة C# أو العكس، وذلك لأن هذه البرامج قد تم ترجمتها إلى اللغة الوسيطة MSIL. وهذه من مميزات تقنية .NET.

أنواع البيانات – Data Types

C#تعتمد بشكل كبير على ما يسمى بالأنواع، فكل شئ في هذه اللغة له نوع، هذه الأنواع تحدد المساحة التخزينية المتاحة للبيانات وطريقة تخزينها في ذاكرة الكومبيوتر.

قبل أن نتطرق لشرح الأنواع المختلفة في C#، دعنا نتعرف قليلاً على ذاكرة الكومبيوتر.

نستطيع أن نتخيل ذاكرة الكومبيوتر بأنها خزانة بها عدد من الأرفف، كل رف من هذه الأرفف يسمى بالـ "موضع" فذاكرة الكومبيوتر عبارة عن عدد معين من المواضع، وكل موضع من هذه المواضع عبارة عن سلسلة من الخانات الثنائية وكل خانة تسمى بتbit ؛ وتحتوي هذه الخانة إما على ٠ أو ١ بحيث أن كل ٨ - بت تمثل بايت byte واحد. وبهذا فإن جميع البيانات تخزن داخل هذه المواضع في صورة ٠ أو ١ فقط ،كما هو موضح في الشكل التالي:



الشكل1-

الأنواع في C# تنقسم من حيث وجودها إلى قسمين:

- **أنواع جاهزة**
- **أنواع غير جاهز**

الأنواع الجاهزة هي الأنواع الموجودة ضمن مكتبة .NET FRAMEWORKوتسمى بالـ Built-In Types، وهي إما أن تكون أنواعاً رقمية أو غير رقمية.

الأنواع الرقمية:

النوع	الحجم بالبايت	الوصف
byte	1	عدد صحيح من 0 إلى 255
sbyte	1	عدد صحيح من –128 إلى 127
short	2	عدد صحيح من –32768 إلى 32767
ushort	2	عدد صحيح من 0 إلى 65535
int	4	عدد صحيح من – 2147483648 إلى 2147483647
uint	4	عدد صحيح من 0 إلى4294967295
long	8	عدد صحيح من - 9223372036854775808 إلى 9223372036854775807
ulong	8	عدد صحيح من 0 إلى 18446744073709551615
Float	4	عدد عشري من $+/-1.5 * 10^{-45}$ إلى $+/-3.4 * 10^{38}$
double	8	عدد عشري موجب من $+/-5.0 * 10^{-324}$ إلى $+/-1.8 * 10^{308}$
decimal	12	عدد عشري من موجب 0 إلى 296 وسالب من – 26 إلى 0

الجدول - ١

الأنواع الغير الرقمية:

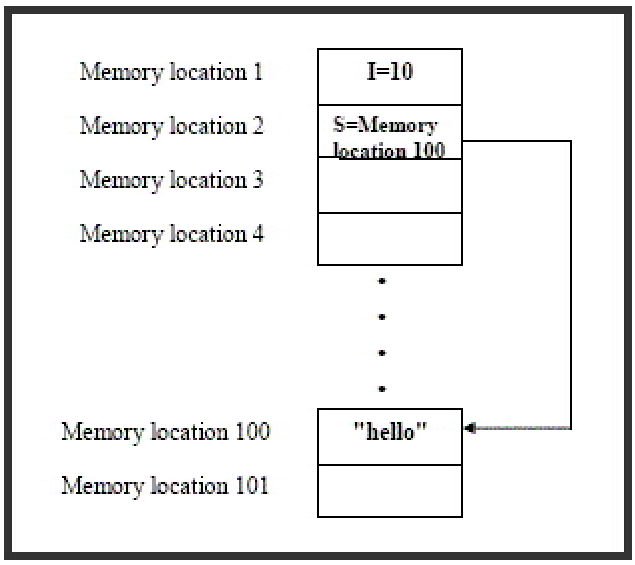
النوع	الحجم بالبايت	الوصف
char	2	رمز يخزن كعدد صحيح من 0 إلى 65535
bool	1	يأخذ قيم true أو false
string	20 على الأقل	مجموعة من الرموز

الجدول - ٢

والقسم الثاني - الأنواع الغير جاهزة؛ هي أنواع يُعرِّفها المستخدم، وتسمى بالـ User-Defined Types. وهي ما سنتعرف عليه لاحقاً مثل: class, struct, delegate, interface.وarray,
تنقسم أيضاً الأنواع في C# من حيث طريقة التخزين إلى قسمين:

- Value Type
- Reference Type

جميع الأنواع الجاهزة التي تعرفنا عليها هي Value Type ما عدا النوع string فهو Reference Type.
وجميع الأنواع الغير جاهزة هي Reference Type ما عدا النوع struct فهو Value Type.
و الفرق بين هذين القسمين أنه في القسم الأول Value Type فإنه يتم تخزين القيمة مباشرة في الموضع المحجوز في ذاكرة الكومبيوتر مباشرة، فمثلاً لو عرفنا متغير من النوع int فإن قيمة هذا المتغير تحفظ في الذاكرة مباشرة، بينما في القسم الثاني Reference Type فإن موضع الذاكرة المحجوز بأحد أنواع هذا القسم المذكورة- كالنوع - string يحمل عنوان موضع آخر حيث توجد به القيمة المخزنة.
والشكل التالي يوضح الفرق بين القسمين، حيث يبين الطريقة التي تخزن فيها البيانات. لدي متغيرين الأول I=10 من النوع int والثاني s="hello" من النوع string.



الشكل2-

إذا كنت مبتدئ في البرمجة، فلا تقلق بشأن ما تعنيه كلمة " متغير "، لأن هذا ما سنتعرف عليه في الدرس القادم بإذن الله.

المتغيرات -Variables

من المُسلم به، أن كل برنامج يتطلب وجود بيانات إما أن يدخلها المستخدم أو أن تكون مخزنة في ذاكرة الكومبيوتر. حيث يتم تشغيل هذه البيانات وإجراء عمليات عليها لنحصل على معلومات والتي هي مخرجات البرنامج.

فإذا تخيلنا - كما قلنا في الدرس السابق - أن ذاكرة الكومبيوتر عبارة عن خزانة بها عدد من الأرفف، فإن المتغيرات تمثل الصناديق التي توضع على هذه الأرفف. فكما أن لكل صندوق اسم معين يصف محتوياته، بالإضافة إلى حجم معين؛ فكذلك المتغيرات لها اسم و نوع لتتمكن من تخزين مختلف البيانات في ذاكرة الكومبيوتر. وسمي

بالمتغير لأن البيانات التي يحملها يمكن تغييرها وليست ثابتة.

وبذلك، فإننا نحتاج عند استخدام أي متغير من تعريفه أولاً، وذلك يكون عن طريق اختيار اسم مناسب له و ذكر نوعه كالتالي:

```
<Variable Type> <Variable Name>;
```

بالنسبة لنوع المتغير فقد وضحنا في الدرس السابق الأنواع المختلفة في لغة C#.

أما بالنسبة لاسم المتغير، فهناك شروط لكتابة أسماء المتغيرات:

- أن يبدأ المتغير إما بحرف أو الرمز _ أو الرمز @ يتبع ذلك سلسة من الأحرف أو الأرقام أو الرمز _
- أن لا يكون اسم المتغير مشابهاً لأحد الكلمات الأساسية في اللغة، ككلمة string أو struct.
- يستحسن أن يكون اسم المتغير يوضح من الوهلة الأولى ماهية البيانات التي تحتويها.
- من الأفضل، وكعادة برمجية حسنة، أن يكون لك شكل معين في تسمية المتغيرات، فمثلاً هناك طريقتين لكتابة أسماء المتغيرات:

Pascal Casing:

في هذه الطريقة، إذا كان اسم المتغير مكون من كلمتين مثل: studentname" "فإن جميع أحرف الكلمتين تكتب بالأحرف الإنجليزية الصغيرة ماعدا الحرف الأول من كل كلمة فيكتب بالأحرف الكبيرة، لتصبح على الصورة: StudentName.

Camel Casing:

في هذه الطريقة، إذا كان اسم المتغير مكون من كلمتين فإن جميع أحرف الكلمتين تكتب بالأحرف الإنجليزية الصغيرة ما عدا الحرف الأول من الكلمة الثانية فيكتب

بالأحرف الكبيرة، لتصبح الكلمة السابقة على الصورة:
studentName.
وكما ذكرنا قبل قليل، فإن هاتين الطريقتين ليست إجبارية، وإنما فقط لتكون برمجتك مفهومة ومقروءة، وليكون لك أسلوب برمجي مفهوم حتى لدى الآخرين!
وسنعتمد على الطريقة الثانية في تسمية المتغيرات في هذه الدروس بإذن الله.
حسناً، لنفرض أن لدي متغير عبارة عن عمر طالب، وعمر الطالب عبارة عن عدد صحيح. كل ما نحتاج إليه لتعريف هذا المتغير هو ذكر نوع المتغير و كتابة اسم مناسب له.
وليكن اسم المتغير هو:studentAge
ونوع المتغير هو:int
إذن سيكون كود C# هو:

```
int studentAge;
```

بعد ذلك نستطيع تعيين قيمة للمتغير لتحفظ في ذاكرة الكومبيوتر، وذلك باستخدام علامة المساواة كالتالي:

```
studentAge=7;
```

أو نستطيع تعيين القيمة أثناء تعريف المتغير:

```
int studentAge=7;
```

وبالطبع يجب أن تكون القيمة مناسبة لنوع المتغير، فلا نضع قيمة عبارة عن string في متغير من النوع!int
ولكن ماذا إذا كان لدي متغيرين من نوعين مختلفين وأردت أن أضع قيمة أحد هذين المتغيرين في المتغير الآخر؟!
كأن يكون لدي متغير من النوع int و آخر من النوع ،long وأردت أن أضع القيمة المخزنة في المتغير من النوع int إلى المتغير الآخر - من النوع - long ؟
لعمل ذلك فهناك طريقة تسمى تحويل الأنواع، أي نحول القيمة من نوع إلى نوع آخر، وذلك بنقلها لمتغير جديد وهناك طريقتين لعمل ذلك وهي ما سنتعرف عليه في الدرس التالي بإذن الله .

تحويل الأنواع - التحويل الضمني-Implicit Conversion

بعد أن تعرفنا على المتغيرات وكيفية تعريفها بأحد الأنواع المختلفة في سي شارب. سنتعرف في هذا الدرس على عملية تحويل المتغيرات من نوع إلى آخر، وهذه العملية تسمى عملية تحويل الأنواع، وهي طريقتين؛ التحويل الضمني و التحويل العلني.

التحويل الضمني:

كما ذكرنا في الدرس الماضي، بأن المتغيرات بمثابة الصناديق التي نحتفظ فيها بالأشياء داخل الخزانة، والأشياء في الكومبيوتر هي القيمة التي يحتفظ بها المتغير ! فمن البديهي أن نختار الصندوق المناسب للقيمة الموجودة لدي! فلا يمكن وضع قيمة كبيرة في صندوق صغير! بينما يمكننا وضع قيمة صغيرة في صندوق أكبر من الصندوق المناسب لهذه القيمة!

وكذلك عندما نريد نقل قيمة من الصندوق المناسب إلى صندوق آخر،فلا بد أولاً من أن يكون حجم الصندوق الجديد مناسب لهذه القيمة أو أكبر منها لنستطيع حفظها فيه.

نستخدم طريقة التحويل الضمني إذا كان حجم المتغير الذي نريد التحويل له يساوي أو أكبر من حجم المتغير الموجود لدي، ويمكننا التعرف على ذلك من الجدولين ١ و ٢ في درس أنواع البيانات.

فمثلاً المتغيرين التالين:

```
int a;

long b;

a=10;
```

```
b=a;

Console.WriteLine("a = {0}", a);

Console.WriteLine("b = {0}", b);
```

في هذا المثال، عرفنا متغيرين؛ a من النوع int و b من النوع long، ثم حولنا المتغير a من النوع int إلى النوع long وذلك بجعل المتغير b يحمل قيمة المتغير a. وستُظهر النتيجة بعد تنفيذ الكود السابق أن كلا المتغيرين سيحملان القيمة10.

سمي هذا النوع من التحويل بالتحويل الضمني، لأنه لم يلزمنا أي كود إضافي للتحويل! فكل ما قمنا به هو تعيين قيمة للمتغير b فقط! وذلك لأن حجم المتغير a وهو من النوع int يساوي ٢ بايت، بينما حجم المتغير b من النوع long هو ٤ بايت، وبذلك فأي قيمة من النوع int نستطيع وضعها بسهولة في متغير من النوع long وذلك لأن المساحة التخزينية لهذا المتغير (٤ بايت) أكثر من كافية بالنسبة لمتغير من النوع int.

والجدول التالي يوضح الأنواع التي يمكن تحويلها لأنواع أخرى بالطريقة الضمنية:

النوع	يمكن تحويله ضمنياً إلى
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

ولكن ماذا إذا أردنا أن نحول متغير من النوع long إلىint ؟

بمعنى آخر، لو كان لدي صندوقين، أحد هذين الصندوقين صغير ويسع كتاب طوله ١٠ سم أو أقل، والآخر صندوق كبير ويسع كتاب طوله ٢٠ سم أو أقل!

فيمكن بسهولة أن أضع محتويات الصندوق الصغير (١٠ سم أو أقل) في الصندوق الكبير (٢٠ سم أو أقل).

ولنفرض أن الصندوق الكبير يحتوي على كتاب طوله ٥ سم، بينما الصندوق الصغير فارغ! وأردت أن أنقل هذا الكتاب للصندوق الصغير، فهل يمكنني ذلك؟!

بالطبع يمكننا ذلك! ونفس الشئ مع المتغيرات يمكننا ذلك باستخدام الطريقة الثانية من طرق تحويل الأنواع وهي طريقة التحويل العلني. كما سنتعرف عليه في الدرس القادم إن شاء الله.

التحويل العلني-Explicit Conversion

في الدرس السابق تعرفنا على أول طريقة من طرق تحويل الأنواع، وهي طريقة التحويل الضمني، وذكرنا بأنه يستخدم لتحويل المتغير إلى نوع يحتاج لمساحة تخزينية أكبر مما هو عليه!

أما في النوع الثاني الذي سنتعرف عليه في هذا الدرس، فيستخدم للتحويل إلى نوع يحتاج لمساحة تخزينية أقل مما هو عليه! ولكن بشرط أن يكون حجم المتغير مناسب للمساحة الجديدة. ومثال الكتاب في الدرس السابق يوضح فكرة هذه الطريقة جيداً.

من الجدولين ١ و ٢ في درس أنواع البيانات، نجد أن المساحة التخزينية المتاحة لمتغير من النوع int هي 2 بايت، ويشمل هذا جميع الأعداد الصحيحة -2147483648 وحتى 2147483647 ولمتغير من النوع long هي ٤ بايت ويشمل هذا جميع الأعداد الصحيحة من ٩٢٢٣٣٧٢٠٣٦٨٥٤٧٧٥٨٠٨ - وحتى 9223372036854775807.

وهذا يؤكد إمكانية تحويل أي متغير long إلى int ولكن بشرط أن تكون قيمة المتغير long ضمن نطاق المتغير int.ولكن التحويل هنا علني أي يحتاج لكتابة كود إضافي كما في المثال التالي:

```
int a;

long b;

b=20;

a=(int) b;

Console.WriteLine("a = {0}", a);

Console.WriteLine("b = {0}", b);
```

هذا المثال عكس المثال السابق، فقد حولنا من long إلى ،int وذلك بنقل القيمة الموجودة في المتغير b من النوع long إلى المتغير a من النوع ،int وهذا ممكن حيث أن القيمة ٢٠ موجودة ضمن نطاق النوعين long و ،int ويتم ذلك بكتابة نوع المتغير الذي نريد التحويل إليه بين قوسين كما هو موضح أعلاه.

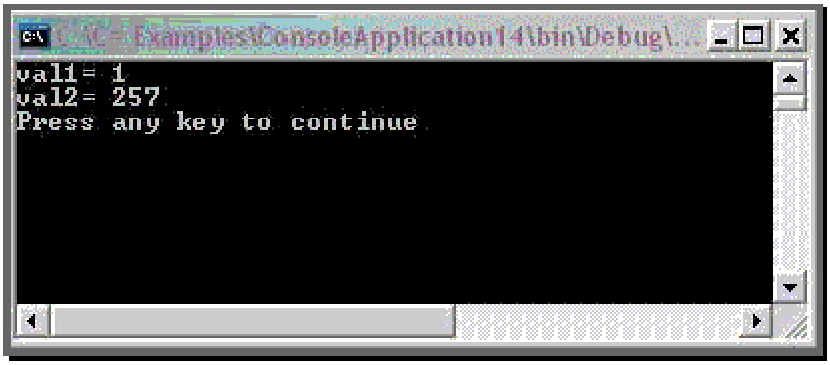
هذا في حال أن القيمة موجودة ضمن نطاقي المتغيرين، ولكن ماذا لو كانت القيمة أكبر من نطاق المتغير الذي نود التحويل إليه؟

كأن نحول من النوع short) مساحته التخزينية ٢ بايت ويشمل هذا أي عدد صحيح ضمن النطاق من ٣٢٧٦٨- إلى ٣٢٧٦٧) إلى النوع byte) مساحته التخزينية ١ بايت ويشمل هذا أي عدد صحيح ضمن النطاق من ٠ إلى ٢٥٥) وذلك بنقل القيمة ٢٥٧ من متغير نوعه short إلى متغير نوعه byte ؟

طبّق معنا هذا المثال لنرى النتيجة:

```
byte val1;

short val2;

val2=257;

val1= (byte) val2;

Console.WriteLine("val1 = {0}", val1);

Console.WriteLine("varl = {0}", val2);
```

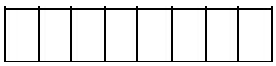
لاحظ أن النتيجة ستظهر كالتالي:



نلاحظ أن القيمة قد تغيرت بعد نقلها إلى المتغير ،val1 أي بعد تحويلها من النوع short إلى النوع ،byte ومن البديهي أن يحصل مثل هذا الأمر، والذي يعتبر من خطأ المبرمج وليس خطأ الكومبيوتر!

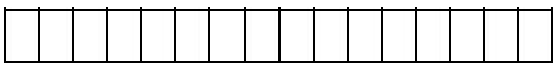
فكما عرفنا أن الكومبيوتر يحتفظ بالبيانات في صورتها الرقمية، أي في صورة سلسة من ٠ أو ١ والتي يتم تخزين كل منها في خانة عشرية.bit

والنوع byte يحجز مساحة قدرها ١ بايت أي ٨ خانات عشرية (١ بايت = ٨ خانات عشرية) لتخزين قيمة المتغير من النوع byte فيها، بينما النوع short فيحجز مساحة قدرها ٢ بايت أي ١٦ خانة عشرية لتخزين قيمة المتغير من النوع short فيها.



المساحة التخزينية المحجوزة للمتغيرval1

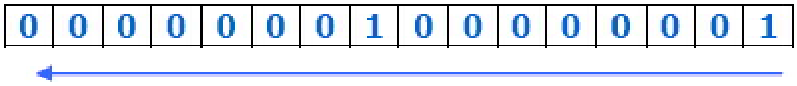
من النوعbyte



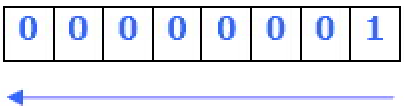
المساحة التخزينية المحجوزة للمتغيرval2

من النوعshort

في المثال السابق، قمنا بتخزين القيمة ٢٥٧ في المتغير val2وهو من النوع short وتم تخزينه كما يلي في ذاكرة الكومبيوتر:



وعند نقل هذه القيمة إلى المتغير val1 وهو من النوع byteيتم تخزين البيانات بدءً من اليمين إلى اليسار وحتى ٨ خانات فقط وهي المساحة التخزينية المتاحة لهذا النوع! كالتالي:



وهذه القيمة الثنائية مساوية للواحد، ولهذا فالنتيجة ظهرت مساوية للواحد بعد نقل القيمة للمتغير val1 في المثال السابق.

هذا الأمر قد يسبب مشاكل في البرنامج، إذا لم ينتبه المبرمج لذلك أثناء البرمجة!

وأفضل طريقة لتجنب حدوث مثل هذا الأمر من دون انتباه المبرمج، هي إضافة الأمر التالي لتنبيه المستخدم برسالة خطأ إذا انتقلت القيمة بصورة غير كاملة إلى متغير ذو مساحة تخزينية أقل من احتياجه. وهناك طريقتين لذلك؛ كتابة كود إضافي، أو بتفعيل هذه الخاصية ضمن بيئة الدوت نت.

الطريقة الأولى:

```
checked(expression);
```

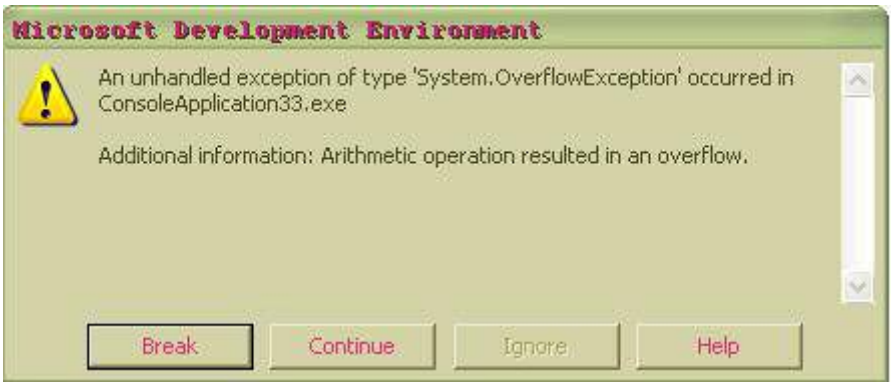
أو العبارة التالية لعدم التنبيه:

```
unchecked(expression);
```

وهذا المثال السابق بعد استخدام أمر التحقق من أن المتغير الجديد مناسب للقيمة المنقولة إليه:

```
byte val1;

short val2;

val2=257;

val1=checked((byte) val2);

Console.WriteLine("val1 = {0}", val1);

Console.WriteLine("varl = {0}", val2);
```

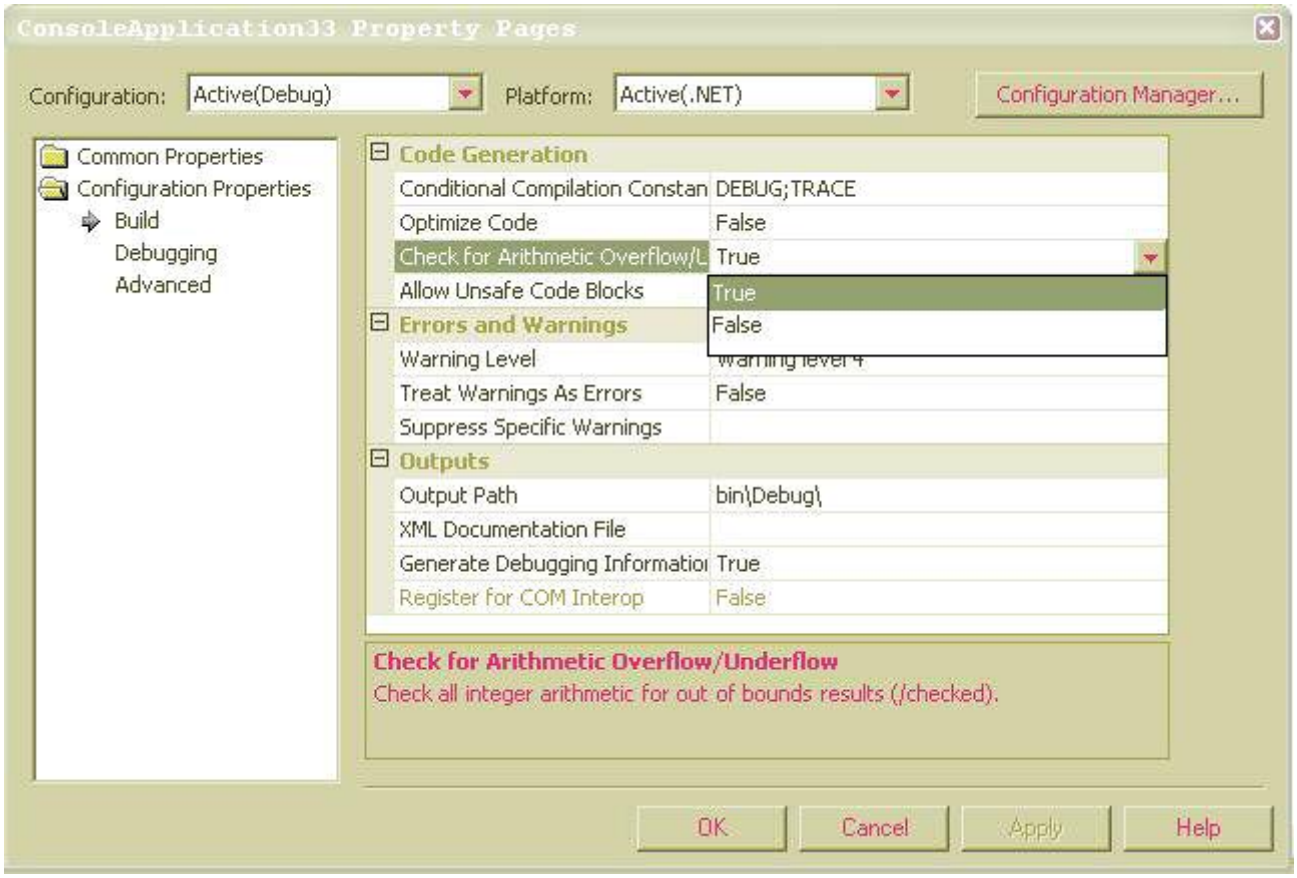
بعد تنفيذ هذا الكود ستظهر رسالة الخطأ التالية:



والطريقة الأخرى كما ذكرنا ضمن بيئة VS.NET تجعل المترجم ينبه المبرمج عن هذا الخطأ دون الحاجة لكتابة الأمر السابق، وذلك بالنقر على اسم المشروع في نافذة Window Explorer بالزر الأيمن، وأختيار Properities، لتظهر نافذة جديدة، اختر Configuration Properties من القائمة الموجودة على يسار هذه النافذة، ثم أختر build. ثم من القائمة الموجودة على يمين هذه النافذة غير في خصائص الخاصية Check for Arithmetic

Overflow/Underflowواجعلها True كما هو موضح في الشكل التالي:

خطأ!



مدينة صنعاء للكمبيوتر

-ماذا يجب أن تعرف أولا؟

قبل أن نبدأ، و حتى تحصل على الفائدة المرجوة من المقال التالي، يجب أن تكون عندك معرفة جيدة بالأمور التالية:

- **ما هي المؤشرات(Pointers) ؟ اقرأ هنا**
- **ما هي C# وNet. ؟**
- **كيفية كتابة برنامج بسيط في C# و ترجمته (Compile).**

المواضيع السابقة ليست محل بحث هذا المقال، لكنه يفترض بك المعرفة الجيدة بها، و إن كنت تشعر أن عندك شيئا من القصور في أحد المواضيع السابقة، أرجو أن لا تتردد في القراءة عنه؛ لتحصل على الفائدة المنشودة من المقال التالي، و لا تنس أنه من السهل أن تجد في الإنترنت ضالتك المنشودة.

-محتويات المقال:

- **المقدمة**
- **كيف و متى تستخدم المؤشرات(Pointers) ؟**
- **أنواع المؤشرات (Pointers)**
- **مثال جيد**
- **الخلاصة**

-المقدمة:

شاع استخدام المؤشرات (Pointers) في لغات البرمجة،

بالخصوص C و ،C++ حيث لم يكن لها بديل لأداء الكثير من الوظائف و العمليات الحيوية التي لا غنى عنها في العديد من البرامج، مثل المصفوفات غير محددة الحجم (Dynamic Array)و مصفوفة المصفوفات أو ما يسمى بـ (Jagged Arrays)و منها مصفوفة السلاسل الحرفية (Strings Array)و كذلك في عملية (Late or Dynamic Binding)و غيرها من الحالات التي لا غنى عن استخدام المؤشرات (Pointers) فيها.
هذا الاستخدام و إن قدم خدمات جليلة للمبرمجين و سرع من أداء البرامج و التطبيقات، إلا أنه بالمقابل زاد العبء على المبرمج الذي أصبح يخصص جزءا لا بأس به من وقته لمراقبة و إدارة موارد النظام و لاسيما الذاكرة يدويا - عن طريق برنامجه طبعا - حيث ما من وسيلة لعمل ذلك آليا ،(Dynamically) و بسبب هذا الوقت "الضائع" و الذي يأخذ جزءا كبيرا من الوقت المخصص لإنجاز البرامج أو التطبيقات، و حاجة الشركات إلى إنجاز مشاريعها بأسرع ما يمكن، إضافة إلى وقوع المبرمجين - هاويهم و محترفهم - في أخطاء "قاتلة" قد تؤدي إلى فشل البرنامج في إنجاز مهامه التي صمم من أجلها، تم طرح البدائل في لغات البرمجة التي تلت C و C++ أو اشتقت منها، و قد يكون المثال الأبرز هنا هو لغة جافا (Java) التي يقوم فيها (Garbage Collector) أو ما يعرف اختصارا بـ (GC) بعملية إدارة الذاكرة و تهيئة المتغيرات فيها و إزالتها منها عند انتفاء الحاجة لها و في الوقت المناسب.
هذه البدائل سرعت من عملية التطوير و إنجاز المشاريع، و قللت من الأخطاء التي كانت تحصل نتيجة الإدارة اليدوية للذاكرة، و أصبح المبرمج يركز اهتمامه على عمل البرنامج بدل التركيز على أفضل السبل لإدارة الذاكرة.
لكن مما يؤسف له أن هذه البدائل نظرت إلى المؤشرات (Pointers)على أنها "شر كلها"، و منعت المبرمج من التعامل المباشر مع الذاكرة، و حرمته من أحد أهم الميزات التي كانت متوفرة في C أو ،C++ و التي كانت تمكنه بحرية و مرونة من تحديد ما يريد من برنامجه أن

يعمل، و هذا - بالتالي - أثر سلبا على قدرات و إمكانيات البرامج و التطبيقات في التعامل مع الذاكرة، و أثر كذلك على سرعة أداء التطبيقات التي تحتاج إلى القيام بعمليات كثيرة في الذاكرة، أو القيام بـ "رحلات" متكررة من و إلى الذاكرة. لاحظ معي المثال التالي - المكتوب بلغة- C#

```
// NormalCopy.cs


using System;


public class NormalCopy

{

    public static void CopyArray(byte []
Src, byte [] Dst)

    {

        for (int j = 0; j < 10000; ++j)

            for (int i = 0; i < Src.Length;
++i)

                Dst[i] = Src[i];

    }



    public static void Main()
```

```
    {

        byte [] MySrcArray = new
byte[1000];

        byte [] MyDstArray = new
byte[1000];



        for (int i = 0; i <
MySrcArray.Length; ++i)

            MySrcArray[i] = (byte) i;



        CopyArray(MySrcArray, MyDstArray);



        Console.Read();

    }

}
```

ملاحظة مهمة جدا: المثال السابق و كل الأمثلة التالية ما هي إلا وسيلة لإيصال فكرة معينة و ليس مثالا يحتذى في الطريقة المثلى للبرمجة

في المثال السابق، يتطلب نسخ مصفوفة (Array) مكونة من ١٠٠٠ عنصر من نوع (byte) لأخرى "رحلات منتظمة" ذهابا و إيابا من و إلى الذاكرة، و ما يصاحب ذلك من عمليات قراءة و كتابة و حجز و تفريغ و ما شابه ذلك،

مما يؤثر سلبا على سرعة أداء البرنامج. البرنامج السابق استغرق من الوقت حوالي (0.02923) ثانية لتنفيذه أو ما يقارب الثلاثة أجزاء بالمائة من الثانية، وقت قصير، أليس كذلك؟ احفظ هذا الرقم لأننا سنعود إليه لاحقا. أرجو الانتباه هنا إلى أن لا فائدة من التكرار (Loop) الأول غير زيادة الوقت المستغرق لتنفيذ البرنامج، و نحن بحاجة لذلك في مثالنا لحصول على زمن معقول يفيدنا في عملية المقارنة.

و بسبب الحاجة لاستخدام المؤشرات (Pointers) و الحاجة لإدارة الذاكرة آليا تم بناء لغة برمجة تجمع ما بين الإثنين، و هي ،C# و تم فيها توفير الإمكانيات لإدارة الذاكرة آليا عن طريق - GC كما في جافا - (Java) إضافة إلى إمكانية الإدارة شبه اليدوية للذاكرة باستخدام المؤشرات ،(Pointers) حيث يكون المبرمج مسؤولا عن تعرف و إدارة متغيرات الذاكرة - المؤشرات - (Pointers) دون أن يكون مسؤولا عن عمليات التفريغ و التنظيف كما هو الحال في C و.C++

-كيف و متى تستخدم المؤشرات(Pointers) ؟

بسبب صعوبة الجمع بين الإدارة الآلية و اليدوية للذاكرة، إضافة إلى طبيعة بنية و هدف ،C# كان لزاما على مصممي اللغة وضع شروط و ضوابط تحكم و تحد استخدام المؤشرات في ،C# و لكن قبل أن نتعرف على هذه الشروط و الضوابط يجب الانتباه إلى أن استخدام المؤشرات (Pointers) في C# غير مستحسن، إلا في حالة أن يكون استخدامها يزيد من سرعة أداء برنامجك بنسب معقولة، أو أن تكون بحاجة لاستخدام مكتبات ربط ديناميكي (DLL) و ما شابهها أو كائنات (COM) و غيرها، و التي لا تكون خاضعة لنظام إدارة الذاكرة التابع لـ Net.

ففي هذه الحالات يكون استخدام المؤشرات (Pointers) منطقيا أو أمرا لا بد منه، أما ما سوى ذلك فلا، لأن C# و إضافة إلى أنها وفرت على المبرمج أداء العمليات الاعتيادية المرتبطة بالذاكرة، و التي كان عليه إنجازها يدويا في C و ،C++ و بسرعة تضاهي سرعة برامج

هاتين اللغتين، فهي لا تعاني من البطء الذي تعاني منه برامج لغات أخرى مثل جافا ،(Java) و أضف إلى هذا و ذاك المجموعة الهائلة من المكتبات (Libraries) و الفئات (Classes)التي تشكل البنية التحتية (Infrastructure) لـ .Net و التي تغني المبرمج - في الغالب - من الحاجة الغوص بنفسه في أعماق النظام، و بالتالي الحاجة لاستخدام المؤشرات.(Pointers)
عند استخدام المؤشرات (Pointers) في ،C# يجب استخدامها ضمن العبارة (unsafe) و هي كلمة محجوزة (Reserved word or Keyword)تحدد جزء الشيفرة (Code)الذي يتضمن استخداما للمؤشرات ،(Pointers) و هذه الكلمة يمكن استخدامها بمفردها أو عند تعريف الأعضاء (Members) سواء كانت من الخصائص (Properties)أو الوظائف (Functions) أو المشيدات (Constructors)أو غيرها و كذلك عند تعريف الفئات (Classes)و ما إلى ذلك، لاحظ الأمثلة التالية:

```
// Using 'unsafe' as block in a function
void MyFunction()

{

...

    unsafe

    {

        // Unsafe code to be here

    }

...

}
```

```
// Using 'unsafe' in function declaration
unsafe void MyFunction()

{

    // Do something

}


// Using 'unsafe' in class declaration
unsafe class MyClass

{

    // Class body to be here

}
```

و السبب الداعي لاستخدام (unsafe) هو أن Net. تتبع سياسة معينة في الأمان، فلا يتم تنفيذ أي ملف أو جزء منه إلا إن كان ذلك التنفيذ آمنا - بعد قيام Net. بالتأكد من ذلك - و هذا يتحقق في حالة عدم استخدام المؤشرات (Pointers)، أما عند استخدام (unsafe) فلا يتم التنفيذ إلا في حالة توفر بيئة موثوقة، لأن Net. لا تقوم بالتأكد من كون التنفيذ آمنا.
عند ترجمة الشيفرة (Code) المتضمنة عبارة (unsafe) إلى (Intermediate Language) أو ما يعرف اختصارا بـ (IL)و تحويل الشيفرة المصدرية (Source Code) إلى (Assembly)ملف تنفيذي (exe) أو مكتبة (dll) و غيرها - يتم التعامل مع (Assembly) كاملا باعتباره ،(unsafe) لأن (unsafe) في Net. يعرّف على مستوى.(Assembly)

يجب أن تعلم أخيرا أنه عند ترجمة شيفرة C# أو (Compile C# Code)تحتوي على عبارة (unsafe) يجب أن ترسل الأمر ((unsafe/إلى مترجم C# أو (C# Compiler)كما في المثال التالي:

```
csc /unsafe MyFile.cs
```

حيث إن (csc) هو مترجم C# أو (- C# Compilerكما هو معلوم -، و (MyFile.cs) هو الملف المراد ترجمته (Compile)إلى(IL)

-أنواع المؤشرات:(Pointers)

أنواع المؤشرات (Pointer Types) محدودة في C#، و الأمر ليس مطلقا كما هو الحال في C و C++، و ليس كل نوع من البيانات (Data Type) يمكن الإشارة إليه باستخدام المؤشر (Pointer)، و سبب ذلك يعود إلى محدودية الحالات التي يتطلب استخدام المؤشرات (Pointers)فيها، و الحفاظ على طابع و بنية C# الآمنة (Type Safe)، إضافة إلى أن طبيعة تعرف الكائنات (Objects)في C# يقوم على مبدأ تعريف مؤشر لكائن (Pointer to Object)الموجود في C++، لاحظ المثال التالي:

```
MyClass MyObject = new MyClass();    // in
C#
MyClass * MyObject = new MyClass(); // in
C++
```

المؤشرات في C# على نوعين:

- void *: و هو المؤشر (Pointer) غير محدد النوع.
- type *

و (type) هو أحد الأنواع التالية:

- أنواع القيم (Value Types) و هي: bool, sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, enum
- أنواع المؤشرات (Pointer Types): أي المؤشرات على المؤشرات (Pointer to Pointer)
- الأنواع المعرفة من قبل المستخدم (User-Define Types): و هي أي أنواع من قبيل التراكيب أو السجلات (Structures) و ليست الفئات (Classes)، فالأولى تعتبر من أنواع القيم (Value Types) و التي يمكن الإشارة إليها (Pointing to)، و الثانية من الأنواع المرجعية (Reference Types) و التي لا يمكن الإشارة إليها بأي حال للسبب المذكور في الفقرة السابقة، و لكن تستثنى المصفوفات (Arrays) من الأنواع المرجعية (Reference Types) التي لا يمكن الإشارة إليها، حيث إن الإشارة للمصفوفات (Arrays) ممكنة لكن بشرط سنأتي على ذكره لاحقا. و أخيرا يجب الانتباه إلى أن التراكيب أو السجلات (Structures) يجب أن لا تحتوي في تركيبها على أي من الأنواع المرجعية (Reference Types) و إلا لن يكون بالإمكان الإشارة إليها.

تأمل معي الأمثلة التالية و التي توضح النقاط السابقة، و طريقة تعريف المؤشرات، و إسناد القيم إليها، و قراءة القيم التي تشير إليها:

```
byte * pMyByte;      // Pointer to byte
bool * pMyBool;      // Pointer to bool
int * * pMyInt;      // Pointer to pointer
to int
long * [] pMyLong;   // Array of pointers to
long
```

```
void * pMyVoid;      // Pointer to unknown
type
char * pC1, pC2;     // Two Pointers to char


// string * pMyString; // Error, 'string'
is reference type

// The compiler generates the following
error message:

// Indirection to managed type is not valid

byte MyByte = 10;    // byte variable

pMyByte = & MyByte; // pMyByte now points
to MyByte (i.e. The value
                     // of pMyByte is the
address of MyByte)

// Array of bool
bool [] MyBool = {true, false, false,
true};
// pMyBool = MyBool     // Error, arrays is
also reference types

// The compiler generates the following
error message:

// Cannot implicitly convert type 'bool[]'
to 'bool*'

// To point to an array use 'fixed'
statement
fixed (bool * pB = MyBool)
```

```
{

    for (int i = 0; i < MyBool.Length; ++i)

        Console.WriteLine((pB + i)-
>ToString());



    // pB++;    // Error, pB is fixed

    // The compiler generates the following
error message:

    // Cannot assign to 'pB' because it is
read-only

}


// Print the value of MyByte (Will prints
10)

Console.WriteLine("MyByte is: {0}", *
pMyByte);


// Another way to get the value of MyByte
(Will prints 10)
Console.WriteLine("MyByte is: {0}",
pMyByte->ToString());
```

و مما تقدم و من الأمثلة السابقة يمكننا ملاحظة التالي:

- (*)هي جزء من اسم النوع (Type Name) و ليست سابقة لاسم المتغير (Variable Name) كما هو الحال في C و ،++C و يتضح ذلك في المثال (char pC1, pC2) *حيث تم تعريف مؤشرين من نوع (char ،(*أما في C و ++C فنتيجة العبارة السابقة هي تعريف مؤشر (Pointer) لـ (char) و هو (pC1) و متغير من نوع (char) هو (pC2)
- تستخدم (*) أو ما يعرف بـ (Pointer Indirection Operator)للحصول على القيمة التي يشير إليها المؤشر (Pointer) مثلما هو الحال في C و ،++C لكن الاختلاف في #C يكمن في أنه لا يمكن الحصول على قيمة المؤشر نفسه - عنوان المتغير الذي يشير إليه -
- المؤشر من نوع ((* Tيحتوي - كقيمة له - عنوان متغير من نوع (T)
- إن كان نوع الكائن (Object) من أنواع المؤشرات (Pointer Types)يتم استخدام المعامل (Operator) (<-)بدل المعامل (.) (Operator) للوصول إلى أعضاء هذا الكائن كما هو واضح في المثال: (pMyByte->ToString())
- لأن المصفوفات (Arrays) من الأنواع المرجعية (Reference Types)فهي تخضع نظام الإدارة الآلية للذاكرة، و بالتحديد لـ ،(GC) و هي معرضة في أي وقت لتغيير عنوانها - عنوان العنصر الأول فيها - أو للإزالة نهائيا من الذاكرة - عند انتفاء الحاجة إليها - و لذلك عند الإشارة للمصفوفات (Pointing to Arrays) نستخدم العبارة (fixed) التي تحمي الكائن - (Object)مؤقتا - من عمليات ،(GC) و تثبته في محله. و لأن استخدام العبارة (fixed) لا يتناسب مع طبيعة و هدف ،(GC) لا يمكن حجز و تثبيت الكائن (Object)إلا لفترة قصيرة، و لإنجاز عمليات سريعة تتطلب وجود الكائن (Object) في محله حتى الانتهاء منها. و هنا يجب الانتباه إلى أن المؤشر

(Pointer)الذي يتم تعريفه في العبارة (fixed) ثابت القيمة - المكان الذي يشير إليه -، و لا يمكن تغيير قيمته.

-مثال جيد:
الآن و بعد أن عرفنا كيف و متى نستخدم المؤشرات (Pointers)، نستطيع العودة لمثالنا الأول (NormalCopy.cs) و التعديل عليه حتى نسرع من عمليه نسخ المصفوفة (Array)، و فيما يلي المثال بعد التعديل (الفكرة مقتبسة من مكتبةMSDN):

```
// FastCopy.cs

using System;


public class FastCopy

{

    public static unsafe void
CopyArray(byte [] Src, byte [] Dst)

    {

        fixed (byte * pSrc = Src, pDst =
Dst)

        {

            byte * pS = pSrc;

            byte * pD = pDst;
```

```
            int Count = Src.Length;


            for (int j = 0; j < 10000; ++j)
            {
                for (int i = Count >> 2; i
!= 0; --i)
                {
                    * ((int *) pD) = *
((int *) pS);
                    pD += 4;
                    pS += 4;
                }


                for (Count &= 3; Count !=
0; --Count)
                {
                    * pD = * pS;
                    pD++;
                    pS++;
```

```
                }

            }

        }

    }


    public static void Main()

    {

        byte [] MySrcArray = new byte[100];
        byte [] MyDstArray = new byte[100];


        for(int i = 0; i <
MySrcArray.Length; ++i)

            MySrcArray[i] = (byte) i;


        CopyArray(MySrcArray, MyDstArray);


        Console.Read();

    }

}
```

في المثال السابق تم استخدام مؤشر (Pointer) من نوع ((* intللقيام بنسخ كل أربعة عناصر معا، و هذا يقلل من عدد المرات التي نحتاج فيها للقراءة و الكتابة من و إلى الذاكرة. لاحظ في التكرار (Loop) الثاني استخدام المعامل ((<<أو (Right-Shift Operator) الذي يقوم بإزاحة العدد الثنائي - العامل (Operand) الأيسر - نحو اليمين بالمقدار المعطى في العامل (Operand) الأيمن. ففي مثالنا (Count << 2) تم إزاحة الرقم ١٠٠٠ و هو (١١١١١٠١٠٠٠) بالثنائي خانتين نحو اليمين - أزيل منه رقمان من اليمين - ليصبح (١١١١١٠١٠) و هو ٢٥٠ بالعشري، إي إن إزالة رقمين من اليمين يقسم العدد - قسمة صحيحة - على أربعة (هل تستطيع أن تخمن نتيجة الإزاحة خانة أو ثلاث؟) و أخيرا في التكرار (Loop) الثالث استخدمنا المعامل ((&أو (Bitwise AND) الذي يقوم بضرب كل خانة من الرقم الثنائي - العامل (Operand)الأيمن - مع نظيرتها في الرقم الثنائي الآخر - العامل (Operand) الأيسر -. و في مثالنا (Count &= 3) تكون النتيجة باقي قسمة الرقم ١٠٠٠ على ٤. إن الهدف من التكرار (Loop) الثاني هو نسخ العناصر أربعا أربعا، أما التكرار (Loop) الثالث فهدفه نسخ ما تبقى من عناصر، إن كان عدد عناصر المصفوفة (Array) لا يقبل القسمة على أربعة، و لا ننسى أن التكرار (Loop) الأول فقط لزيادة وقت تنفيذ البرنامج.

على الرغم من أن المثال السابق أطول من المثال الأول، و قد يبدو معقدا كذلك، إلا أنه استغرق (٠,٠٠٠٨٧) ثانية لإتمام التنفيذ، و بالمقارنة مع الزمن الذي حصلنا عليه في المرة السابقة، يبدو فارق السرعة بين المثالين واضحا جدا لصالح المثال السابق، فالزمن الذي سجله المثال الأول أطول بما يزيد عن ٣٦ مرة، و هذا الفارق أكثر من كاف للمبرمج ليتخذ قراره باستخدام المؤشرات (Pointers)إن كان برنامجه يقوم بالكثير من هذه

العمليات التي تستهلك وقتا يمكن توفيره لصالح سرعة الأداء.
في الإنترنت تجد العديد من الأمثلة الحقيقية و الأكثر تعقيدا و التي تجعل من استخدام المؤشرات (Pointers) أمرا لا بد منه. كما تجد في مكتبةMSDN مثالا جميلا عن استخدام المؤشرات (Pointers) في معالجة الصور، و ما توفره لك من إمكانيات و سرعة أداء لا يمكن الوصول إليها بدون استخدام المؤشرات.(Pointers)

-الخلاصة:

و خلاصة القول أن المؤشرات (Pointers) في C# لا تختلف كثيرا عن نظيراتها في C و ،C++ و لكن و على الرغم من توفر ميزة استخدام المؤشرات (Pointers) في ،C#إلا أن تجنب استخدامها أولى من استخدامها، ما لم تكن تؤدي للمبرمج أعمالا و تنجز له أمورا بحيث لا يمكن الاستغناء عنها بغيرها.