الفصل الحادى والعشرون
تعقب الأخطاء وتصحيحها

تعتبر عملية تعقب الأخطاء وتصحيحها **Debugging** من العمليات الأساسية التى تأخذ حيزاً كبيراً من اهتمام المبرمج عند تطوير تطبيقاته، الصغير منها والكبير على حدٍ سواء. فكما تعرف مسبقاً بأن مراحل إنشاء التطبيق تتمثل فى مرحلة تصميم التطبيق ثم مرحلة تمثيله ثم المرحلة الثالثة والأخيرة والتى تتمثل فى تعقب أخطاء التطبيق وتصحيحها.

فى هذا الفصل ستتعرف على:

- استخدام أنماط المترجم
- تعيين خيارات تعقب الأخطاء ومستوياتها
- تصحيح الأخطاء أثناء التشغيل

كما هو متوقع، تحتوى لغة Visual C++ على بيئة شاملة لتعقب الأخطاء وتصحيحها كما تحتوى على مجموعة من الأدوات التى تساعدك على اكتشاف هذه الأخطاء أثناء مراحل إعداد التطبيق، حيث يمكنك اكتشاف الأخطاء بسرعة متناهية ومشاهدة محتويات متغيراتك، كما يمكنك تتبع خط سير البرنامج سواءً من خلال الكود الذى قمت بإدخاله أو كود دوال وتصنيفات مكتبة MFC. كما يمكنك أيضاً استخدام بعض الأدوات الأخرى كبرنامج Spy++ للتعرف على الرسائل المتبادلة بين نظام التشغيل وتطبيقك أثناء تشغيل البرنامج، كما يمكنك أيضاً التعرف على المورد المستخدم حالياً من قبل التطبيق. ابق معنا فى هذا الفصل الشيق من فصول الكتاب وستجد ما يسرك.

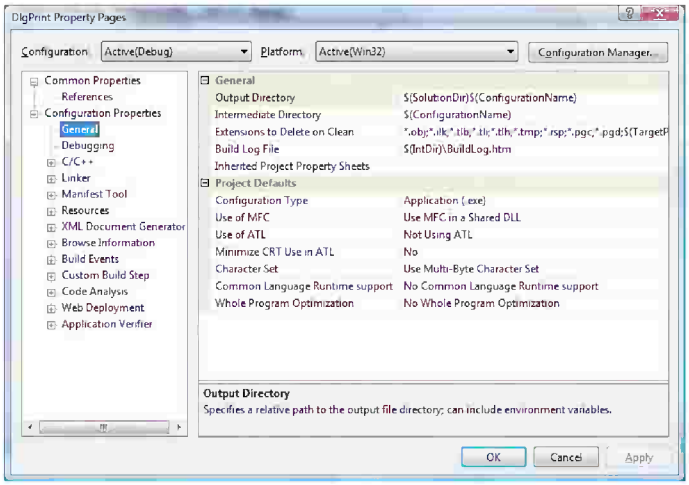
استخدام أنماط المترجم

ذكرنا فى بداية هذا الكتاب أن هناك نمطين أو شكلين أساسيين يستخدمهما المعالج عند بناء التطبيقات. الأول هو نمط تصحيح الأخطاء Debug mode والثانى نمط النشر Release mode. وحينما تقوم بإنشاء تطبيق جديد، يمكنك استخدام أى من النمطين، إلا أن كل منهما يتسبب فى إنشاء كود مختلف عن الآخر. فإذا قمت ببناء التطبيق فى نمط تصحيح الأخطاء، تتسبب عملية بناء التطبيق فى إنتاج تطبيق تنفيذى كبير الحجم وبطيء فى التنفيذ وذلك بسبب تضمين العديد من معلومات تصحيح الأخطاء. أما إذا قمت ببناء نفس التطبيق فى نمط النشر، فسيتم إنتاج تطبيق تنفيذى صغير الحجم وسريع فى التنفيذ مقارنةً بالحالة الأولى، إلا أنك لن تستطيع الخطو خلال الكود الأساسى للتطبيق أو تلقى أى رسائل عند حدوث الأخطاء أثناء تنفيذه.

والمتبع دائماً عند تطوير تطبيقاتك، أن تترك نمط تصحيح الأخطاء كما هو حتى يمكن التعرف على أخطاء الكود الموجودة بالتطبيق، وبمجرد انتهائك من تطوير التطبيق وعزمك على نشره إلى مستخدميك، يمكنك تغيير نمط البناء إلى نمط النشر. للتحكم فى نمط البناء بعد إنشاء التطبيق، تابع معنا إحدى الطرق الآتية:

الطريقة الأولى:

١. تأكد من تنشيط اسم المشروع داخل نافذة الحل ثم افتح قائمة **Project** من شريط القوائم واختر **Properties** من القائمة المنسدلة، يظهر المربع الحوارى **Property Pages** (انظر شكل ٢١-١).



شكل ٢١-١اختيار نمط البناء من المربع الحوارى **Property Pages**

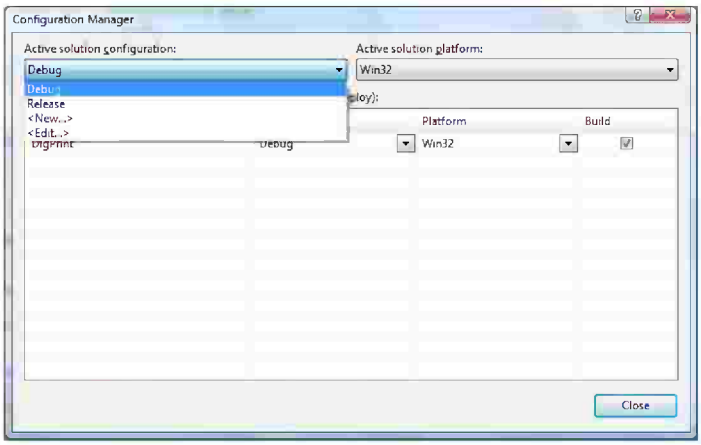
٢. تأكد من اختيار **Configuration Properties** من الجانب الأيسر بالمربع الحوارى ثم اختر نمط البناء المناسب من مربع السرد والتحرير **Configuration**.

٣. بمجرد اختيارك لأحد الأنماط، سيتم تطبيق الخيارات التى ستقوم بتعيينها داخل التبويبات الأخرى على هذا النمط فقط. فإذا أردت تطبيق هذه الخيارات على جميع الأنماط فى آنٍ واحد، اختر **All Configurations** من مربع السرد والتحرير **Configuration**.

الطريقة الثانية:

١. افتح قائمة **Build** من شريط القوائم ثم اختر **Configuration Manager** من القائمة المنسدلة، يظهر المربع الحوارى **Configuration Manager** (انظر شكل ٢١-٢).

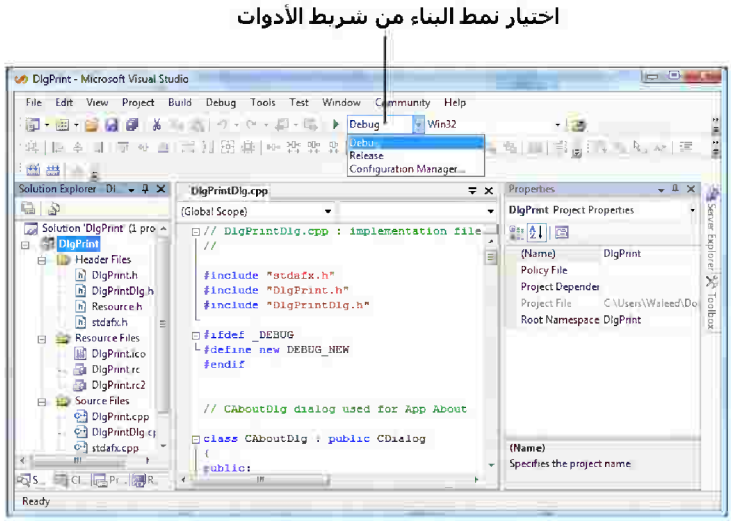
٢. اختر النمط المطلوب من مربع السرد Active Solution Configuration ثم انقر زر Ok لإغلاق المربع الحوارى.



شكل ٢١-٢ اختيار نمط البناء من قائمة Build

الطريقة الثالثة:

من شريط الأدوات القياسى، تلاحظ وجود مربع سرد وتحرير خاص بأنماط البناء (انظر شكل ٢١-٣). قم باختيار النمط المطلوب من هذا المربع.



شكل ٢١-٣ اختيار نمط البناء من شريط الأدوات القياسى

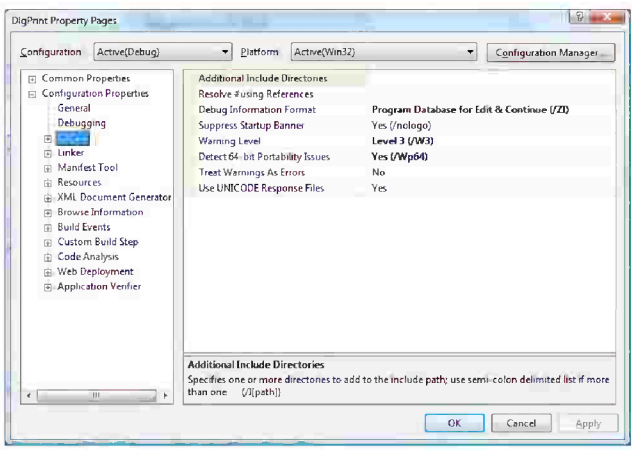
تعيين خيارات تعقب الأخطاء ومستوياتها

يمكنك التحكم فى العديد من خيارات تعقب الأخطاء ومستويات الأخطاء نفسها من خلال التبويب C/C++ فى المربع الحوارى Property Pages الخاص بالمشروع، وذلك كما يلى (انظر شكل ٢١-٤):

- يمكنك من خلال مربع السرد والتحرير Warning Level اختيار مستوى رسائل تحذيرات المترجم التى يقوم بإظهارها خلال عملية الترجمة. يمكنك اختيار إحدى القيم الموضحة بالجدول ٢١-١.

جدول ٢١-١ مستويات التحذير

المستوى	التحذيرات الناتجة
Off	لا يوجد تحذيرات
Level 1	التحذيرات الخطيرة جداً فقط
Level 2	التحذيرات الأقل خطورة
Level 3	المستوى الافتراضى ويحتوى على جميع الرسائل المعقولة
Level 4	جميع التحذيرات



شكل ٢١-٤ التبويب C/C++ داخل المربع الحوارى Property Pages

وكما تلاحظ فإن المستوى الثالث هو المستوى الافتراضى، إلا أن الكثير من المبرمجين يفضلون المستوى الرابع للتأكد من عدم وجود أى خلل فى تطبيقاتهم مهما كانت درجته.

- قم بتخصيص القيمة Yes للخاصية Treat Warnings As Errors إذا أردت اعتبار التحذيرات أخطاء وبالتالى عدم بناء التطبيق فى حالة وجود أى تحذير.
- يمكنك من مربع السرد والتحرير Debug Information Format تعيين مستوى إظهار معلومات تصحيح الأخطاء باختيار إحدى القيم الموضحة بالجدول ٢١-٢ التالى.

جدول ٢١-٢ مستويات معلومات تصحيح الأخطاء

المستوى	المعلومات الناتجة
Disabled	عدم إظهار أى معلومات وتستخدم غالباً مع نمط النشر
C 7 Compatible	إظهار معلومات تعقب الأخطاء المتوافقة مع برنامج Microsoft C 7.0
Program Database	إنشاء ملف بالامتداد pdb يحتوى على أعلى مستوى لمعلومات تعقب الأخطاء ولكنه لا يحتوى على معلومات Edit وContinue
Program Database for Edit & Continue	وهو الخيار الافتراضى ويقوم بإنشاء ملف بالامتداد pdb يحتوى على أعلى مستوى لمعلومات تعقب الأخطاء بما فيها معلومات Edit وContinue
<inherit from parent or project defaults>	استخدام الخيار المعرف بالمشروع الأبوى أو خيارات المشروع الافتراضية المعرفة داخل Visual C++

- يمكنك من خلال مربع السرد Whole Program Optimization الاختيار بين سرعة تنفيذ التطبيق وبالتالى تكبير حجمه أو تصغير حجم التطبيق وبالتالى بطئ تنفيذه.
- يمكنك من خلال التبويب Browse Information جعل المترجم يقوم بإظهار المعلومات التى تساعدك على التعرف على العلاقات بين المتغيرات والدوال

والتصنيفات المختلفة فى حالة حدوث أى أخطاء، إلا أن هذه المعلومات تقلل من سرعة عملية الترجمة.

- يمكنك من خلال الخاصية **Preprocessor Definition** بالتبويب **Resources** تعيين تعريفات واضحة يتم تعريفها بمجرد ترجمة التطبيق، ومن ثمَّ يمكنك استخدامها مع التركيب **#endif , #else , #ifDef** لترجمة أجزاء من الكود فى نمط معين. فمثلاً التعريف **_DEBUG** يتم تعيينه تلقائياً فى نمط تصحيح الأخطاء ويمكنك استخدامه لترجمة كود معين فى نمط تصحيح الأخطاء فقط هكذا:

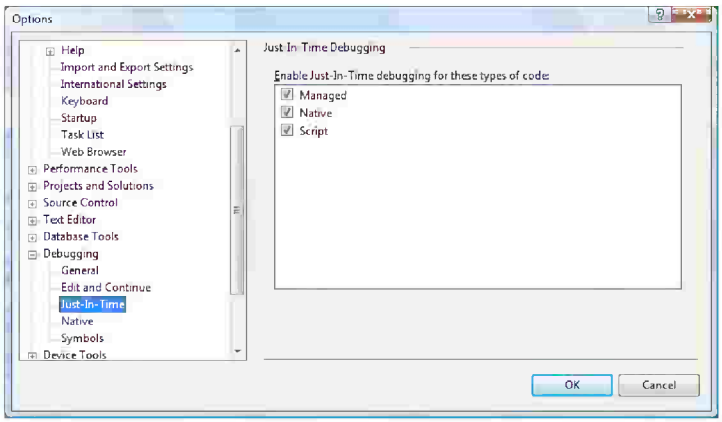
```
int a = b * c/d + e;
#ifDef _DEBUG
CString strMsg;
StrMsg.Format("Result of sum was %d,a);
AfxMessageBox(strMsg);
#endif
```

حيث يتم ترجمة مربع الرسالة فى حالة بناء التطبيق فى نمط تصحيح الأخطاء فقط.

تصحيح الأخطاء أثناء التشغيل

يتيح لك خيار تصحيح الأخطاء أثناء التشغيل **Just-in-time- debugging** تصحيح أخطاء تطبيق يعمل بصورة صحيحة، إلا أن مشكلةً ما قد ظهرت به أثناء التشغيل. فى حالة تمكين هذا الخيار ووجود نسخة من **Visual C++** مثبتة على حاسبك، يتم تحميل التطبيق الذى قام بإحداث الخطأ داخل بيئة تطوير خاصة للبدء فى إصلاح هذا الخطأ. لتمكين هذا الخيار، تابع معنا الخطوات الآتية:

١. افتح قائمة **Tools** من شريط القوائم ثم اختر **Options** من القائمة المنسدلة، يظهر المربع الحوارى **Options** (انظر شكل ٢١-٥).



شكل ٢١-٥ تعيين خيارات تصحيح الأخطاء أثناء التشغيل

٢. نشط التبويب **Debugging** بالجانب الأيسر من المربع الحوارى ثم اختر العنصر الفرعى **Just-In-Time** إذا لم يكن هو التبويب النشط.

٣. نشط مربعات الاختيار الموجودة بالجانب الأيمن من المربع الحوارى ثم انقر زر **Ok** لإغلاقه.

تتبع كود التطبيق

من السمات الهامة داخل بيئة تصحيح الأخطاء اقتفاء أثر الكود وتنفيذه خطوة بخطوة وسطر بسطر ومن ثمَّ مشاهدة محتويات المتغيرات مع كل خطوة. كما يمكنك استخدام نقاط التوقف **Breakpoints** إذا أردت تشغيل التطبيق حتى يصل إلى نقطة من هذه النقاط وحينئذٍ يقف عندها حتى يأخذ الأمر منك بالاستمرار. كما يمكنك أيضاً استخدام المختزلات **Trace** و**Assert** و**Verify** لاكتشاف الأخطاء أثناء التشغيل. وهذا ما سنتعرض له بشيءٍ من التفصيل فى الأجزاء التالية من الفصل، فكن معنا.

استخدام المختزل TRACE

يمكنك إضافة المختزلات **TRACE** فى أماكن متعددة من تطبيقك للتأكد من تشغيل أجزاء الكود المختلفة بشكل صحيح أو محتويات المتغيرات فى هذه الأماكن، حيث يتم ترجمة هذه المختزلات إلى كودك فى نمط تصحيح الأخطاء ويتم إظهار نتائجها داخل نافذة العرض

بالتبويب Debug حينما تقوم بتشغيل تطبيقك من خلال مصحح الأخطاء. وفى حالة بناء التطبيق فى نمط النشر، يمكنك ترك هذه المختزلات، حيث يتم تجاهلها فى الملفات الناتجة. تتشابه طريقة عمل المختزل TRACE كثيراً مع طريقة عمل الدوال Printf() و Format(). للتعرف على طريقة عمل المختزل TRACE، قم بإنشاء تطبيق أحادى الوثيقة باسم مناسب وليكن Debug ثم قم بإضافة الكود التالى داخل تصنيف الوثيقة CDebugDoc وقبل كود الدالة OnNewDocument():

```
1.    void Swap(CUIntArray* pdwNumbers,int i)
2.    {
3.      UINT uVal = pdwNumbers->GetAt(i);
4.      pdwNumbers->SetAt(i,pdwNumbers->GetAt(i+1));
5.      pdwNumbers->SetAt(i+1,uVal);
6.    }
7.    void DoSort()
8.    {
9.      CUIntArray arNumbers; int i = 0;
10.     for(int i=0;i<10;i++)
11.     arNumbers.Add(1+rand()%100);
12.     TRACE("Before Sort\n");
13.     for(i=0;i<arNumbers.GetSize();i++)
14.       TRACE("[%d] = %d\n",i+1,arNumbers[i]);
15.     BOOL bSorted;
16.     do
17.     {
18.       bSorted = TRUE;
19.       for(i=0;i<arNumbers.GetSize()-1;i++)
20.       {
21.         if(arNumbers[i] > arNumbers[i+1])
22.         {
23.             Swap(&arNumbers,i);
24.              bSorted = FALSE;
25.         }
26.       }
27.     } while(!bSorted);
28.     TRACE("After Sort\n");
29.     for(i=0;i<arNumbers.GetSize();i++)
30.     TRACE("[%d] = %d\n",i+1,arNumbers[i]);
```

```
31.  }
```

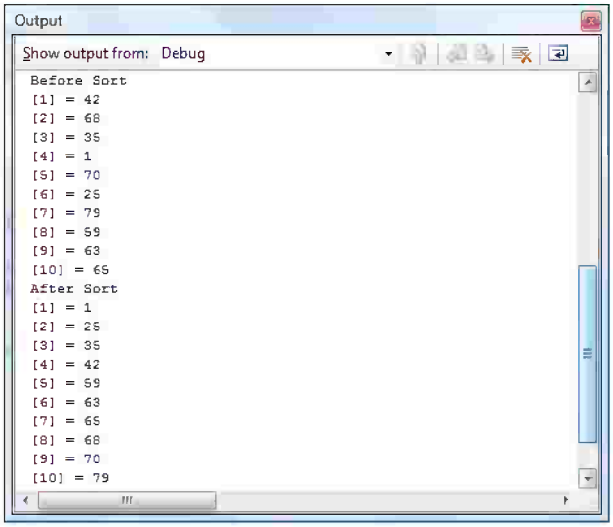
وعن هذا الكود، نوضح ما يلى:

- نقوم فى هذا المثال بتنفيذ كود بسيط لترتيب مجموعة من الأرقام.
- تحتوى السطور من ١إلى ٦ على كود الدالة **()Swap** التى سنستخدمها لترتيب كل رقمين متتاليين. حيث يتم استخدام الدالة **()SetAt** لتبديل الرقمين.
- تحتوى السطور من ٧ إلى ٣١ على كود الدالة **()DoSort** المستخدمة فى عملية الترتيب.
- فى السطور من ٩ إلى ١١، يتم إنشاء مصفوفة وإضافة ١٠ أرقام عشوائية إليها.
- فى السطر رقم ١٢، تم استدعاء المختزل **TRACE** لإظهار النص **Before Sort**.
- فى السطور ١٣ و١٤، تم استدعاء المختزل **TRACE** لإظهار بيانات المصفوفة قبل الترتيب.
- فى السطور من ١٥ إلى ٢٧ يتم استخدام الدالة **()Swap** والمتغير المنطقى **bSorted** لترتيب عناصر المصفوفة.
- فى السطر رقم ٢٨، تم استدعاء المختزل **TRACE** لإظهار النص **After Sort**.
- فى السطور ٢٩ و٣٠، تم استدعاء المختزل **TRACE** لإظهار بيانات المصفوفة بعد الترتيب.
- يجب أن تقوم باستدعاء الدالة **()DoSort** داخل الدالة **()OnNewDocument**.

كى تشاهد بنفسك نتيجة الكود السابق، تابع معنا الخطوات الآتية:

١. تأكد من تنشيط نافذة الإخراج، وإلا افتح قائمة **View** من شريط القوائم ثم اختر **Output** من القائمة المنسدلة الناتجة أو اضغط الاختصار **Alt+2** من لوحة المفاتيح.

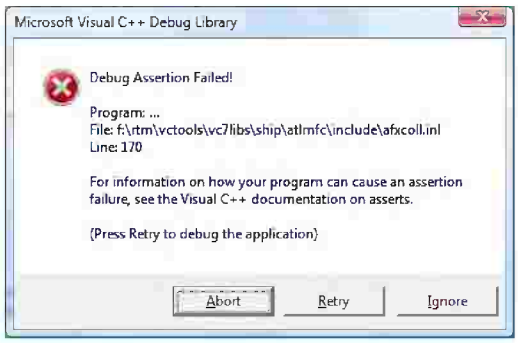
٢. اضغط مفتاح **F5** من لوحة المفاتيح أو افتح قائمة **Debug** من شريط القوائم ثم اختر **Start Debugging** من القائمة المنسدلة، تلاحظ ظهور بيانات المصفوفة قبل الترتيب وبعده (انظر شكل ٢١-٦).



شكل ٢١-٦ استخدام المختزل TRACE

استخدام المختزل ASSERT

يمكنك استخدام المختزل **ASSERT** للتأكد من صحة شرط معين، حيث يحتوى المختزل على معامل منطقى واحد فقط. فإذا كانت قيمة هذا المعامل **TRUE**، فكأن شيئاً لم يكن. أما إذا كانت قيمته **FALSE**، فيتم إيقاف البرنامج ويظهر المربع الحوارى **Debug Assertion Failed** (انظر شكل ٢١-٧) الذى يستحثك على تخطى التطبيق **Abort** أو محاولة تنفيذ الكود مرةً أخرى **Retry** أو تجاهل الرسالة **Ignore**، كما يقوم المربع الحوارى بإظهار اسم التطبيق واسم الملف ورقم السطر الذى يحتوى على الخطأ. اختر **Abort** إذا أردت تخطى طور تصحيح الأخطاء، اختر **Retry** إذا رغبت فى تعديل الكود الذى حدث عنده الخطأ، أما إذا كنت تعرف مسبقاً عدم تأثير هذا الخطأ على سير التطبيق أو كنت لا تعبأ به مطلقاً، فقم باختيار **Ignore** واستمر فى تشغيل التطبيق.



شكل ٢١-٧ المربع الحوارى الناتج عن تشغيل المختزل ASSERT

ومن الاستخدامات الشهيرة للمختزل **ASSERT** أيضاً التأكد من صحة المعاملات المدخلة للدالة. فمثلاً فى الكود السابق، إذا أردت اختبار صحة المعاملات المدخلة للدالة **()Swap**، قم بإضافة الكود التالى لبداية الدالة:

```
ASSERT(pdwNumbers);
ASSERT(i>=0 && i<10);
```

حيث يتم التأكد فى السطر الأول من أن مؤشر مصفوفة البيانات لا يشير إلى القيمة صفر، وفى السطر الثانى يتم التأكد من ترتيب العنصر الذى سيتم تبديله من **0** إلى **9**. فإذا لم يتحقق أى من الشرطين، يتم إيقاف التطبيق ويظهر المربع الحوارى **Debug Assertion Failed** وبذلك يمكنك استخدام هذه التقنية فى تجنب الأخطاء الغير متوقعة والتى تنجم عن تمرير معاملات غير صحيحة للدالة.

هناك بعض المختزلات الأخرى المنبثقة من المختزل **ASSERT** والتى أهمها على الإطلاق ما يلى:

- المختزل **ASSERT_VALID** ويستخدم مع التصنيفات المنبثقة عن التصنيف **CObject** ويقوم بعمل اختبار شامل على العنصر ومحتوياته للتأكد من أنه يعمل بحالة صحيحة ومقبولة، حيث يتم تمرير مؤشر العنصر المراد اختباره كما يلى:

```
ASSERT_VALID(pdwNumbers);
```

- المختزل **ASSERT_KINDOF** ويستخدم مع التصنيفات المنبثقة عن التصنيف **CObject** للتأكد من أن عناصرها تحتوى على نوع التصنيف الصحيح. فمثلاً يمكنك التأكد من أن مؤشر العرض ينتمى إلى التصنيف المناسب هكذا:

```
ASSERT_KINDOF(CMyDebugView,pMView);
```

فإذا لم يكن المؤشر **pMView** من النوع **CMyDebugView**، يتم إظهار المربع الحوارى **Debug Assertion Failed**.

استخدام المختزل VERIFY

ذكرنا منذ قليل أنه يتم تجاهل كود المختزل **ASSERT** عند بناء التطبيق فى نمط النشر **Release Mode**، لذا ينبغى ألا تستخدم الكود الضرورى لعمل التطبيق الطبيعى داخل المختزل **ASSERT**. لتوضيح ذلك، دقق معى سريعاً فى الكود التالى:

```
1.    int a = 0;
2.    ASSERT(++a > 0);
3.    if(a>0)
4.       MyFunc();
```

إذا قمت ببناء التطبيق فى نمط تصحيح الأخطاء، يحدث ما يلى:

- فى السطر رقم ١ يتم إعطاء القيمة **0** للمتغير الصحيح **a**.
- فى السطر رقم ٢ يتم زيادة المتغير **a** لتصبح قيمته **1** وبالتالى تكون نتيجة استدعاء المختزل **ASSERT** هى **TRUE**.
- فى السطر رقم ٣ يتم اختبار قيمة المتغير **a** باستخدام عبارة **if**، ولأنها **0<**، يتم استدعاء الدالة **()MyFunc** فى السطر رقم ٤.

أما إذا قمت ببناء التطبيق فى نمط النشر، فسيحدث ما يلى:

- فى السطر رقم ١ يتم إعطاء القيمة **0** للمتغير الصحيح **a**.
- يتم تجاهل السطر رقم ٢ لأنه لا يعمل فى نمط النشر.
- فى السطر رقم ٣ يتم اختبار قيمة المتغير **a** باستخدام عبارة **if** ، ولأنها ليست **0<** ، لا يتم استدعاء الدالة **()MyFunc** فى السطر رقم ٤.

للتغلب على هذه المشكلة، يمكنك استخدام المختزل **VERIFY** الذى يقوم فى نمط تصحيح الأخطاء بإظهار المربع الحوارى **Debug Assertion Failed** فى حالة إرجاع القيمة **0**، أما فى نمط النشر فيقوم بأداء نفس المهام إلا أنه لا يقوم بإظهار المربع الحوارى فى حالة إرجاع القيمة **0**. لذا يجب استخدام المختزل **VERIFY** حينما ترغب فى إيجاد قيمة تعبير ما دائماً، أم المختزل **ASSERT** فيستخدم للاختبار أثناء عملية تصحيح الأخطاء فقط.

فإذا قمت الآن باستبدال المختزل **ASSERT** بالمختزل **VERIFY**، سيعمل التطبيق بشكل صحيح عند بنائه فى نمط النشر.

من الاستخدامات الشهيرة أيضاً للمختزل **VERIFY** اختبار القيم المرجعة من الدوال هكذا:

```
VERIFY(MyFunc() != FALSE);
```

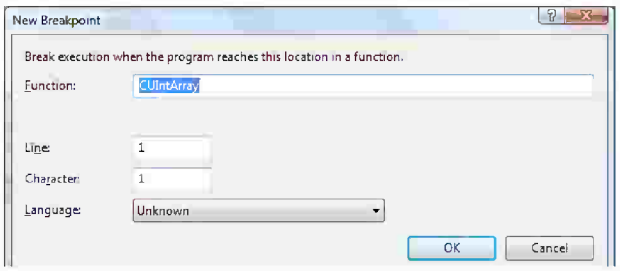
حيث يتم التأكد من صحة القيمة المرجعة من الدالة **MyFunc()**.

استخدام نقاط التوقف

تعتبر نقاط التوقف **Breakpoints** من أدوات تصحيح الأخطاء المؤثرة المستخدمة لتعقب وتصحيح معظم الأخطاء الموجودة بالبرنامج، حيث يمكنك وضع نقطة توقف فى أى مكان داخل كود تطبيقك ثم تشغيل التطبيق داخل مصحح الأخطاء. وحينما تصل إلى نقطة التوقف، يتم الإيقاف المؤقت للتطبيق ويكون لديك الخيار بالتحرك داخل الكود خطوة بخطوة أو الاستمرار أو حتى إنهاء التطبيق.

لإضافة نقطة توقف لمكانٍ ما داخل الكود، تابع معنا الخطوات الآتية:

١. انقر بزر الفأرة داخل سطر الكود الذى ترغب فى وضع نقطة التوقف عنده.
٢. اضغط زر **F9** من لوحة المفاتيح.
٣. يمكنك بدلاً من الخطوة السابقة فتح قائمة **Debug** من شريط القوائم ثم اختيار **New Breakpoint** من القائمة المنسدلة ثم **Break at a function** لإظهار المربع الحوارى **New Breakpoint** (انظر شكل ٢١-٨).



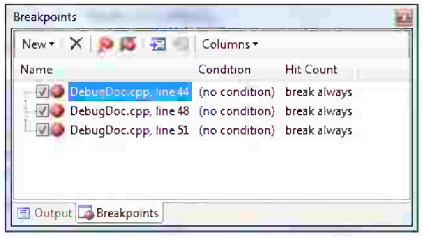
شكل ٢١-٨ إضافة نقطة التوقف باستخدام المربع الحوارى **New Breakpoint**

٤. بمجرد إضافتك لنقطة التوقف، تظهر دائرة حمراء صغيرة بجوار السطر الذى قمت بتحديده. يتم دائماً إضافة نقاط التوقف بجوار سطور الكود الصحيحة، وإلا تقوم بيئة التطوير نيابةً عنك بنقل نقطة التوقف إلى أقرب سطر كود صحيح.

لحذف نقطة التوقف، اضغط مفتاح **F9** من لوحة المفاتيح أو انقر نقطة التوقف.

أما إذا أردت الإبقاء على نقطة التوقف مع تعطيلها، فقم بتعطيل مربع الاختيار المجاور لنقطة التوقف داخل نافذة **Breakpoints** (انظر شكل ٢١-٩). ولتنشيطها مرةً أخرى، قم باتباع أى من الطرق التالية:

- قم بتنشيط مربع الاختيار المجاور لنقطة التوقف داخل نافذة **Breakpoints** .
- اضغط مفتاح **F9** من لوحة المفاتيح.



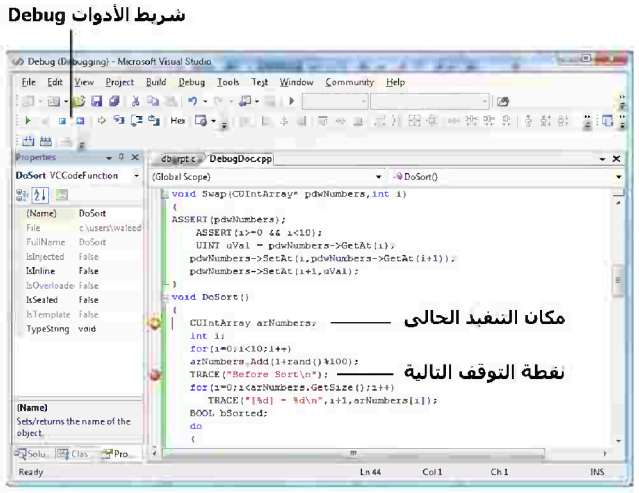
شكل ٢١-٩ نافذة نقاط التوقف **Breakpoints**

بعد أن قمت بتعيين نقاط التوقف، يمكنك تشغيل التطبيق من خلال مصحح الأخطاء باتباع إحدى الطرق التالية:

- اضغط مفتاح **F5** من لوحة المفاتيح.
- انقر الرمز من شريط الأدوات **Debug** (انظر شكل ٢١-١٠).

- افتح قائمة **Debug** ثم اختر **Start Debugging** من القائمة المنسدلة.

باتباع أي من هذه الطرق يبدأ البرنامج في العمل الطبيعي له حتى يصل إلى نقطة التوقف، وحينئذ يتوقف البرنامج ويظهر سهم صغير داخل الدائرة الحمراء (انظر شكل ٢١-١٠) وبذلك يمكنك استخدام شريط الأدوات **Debug** للتنقل عبر الكود كما سنرى بعد قليل. كما يمكنك أيضاً التعرف على محتويات المتغيرات بتمرير مؤشر الفأرة بضع ثوانٍ على المتغير، حيث يظهر تلميح به محتوياته.



شكل ٢١-١٠ توقف المصحح عند نقطة التوقف

التنقل عبر الكود

بمجرد أن يتوقف البرنامج عند إحدى نقاط التوقف، يمكنك التنقل عبر الكود باستخدام شريط الأدوات **Debug** كما في جدول ٢١-٣ أو خيارات قائمة **Debug**.

جدول ٢١-٣ أزرار شريط الأدوات **Debug**

الزر	الاختصار	العمل
	**F11**	يقوم المصحح بتنفيذ السطر الحالي وإذا كان المؤشر فوق اسم دالة، سيقوم المصحح بالدخول إلى هذه الدالة
	**F10**	يقوم المصحح بتنفيذ السطر الحالي وإذا كان المؤشر فوق

الزر	الاختصار	العمل
		اسم دالة، يقوم المصحح بتشغيل الدالة بسرعتها العادية ويتوقف بمجرد استرجاع قيمة هذه الدالة
	Shift+F11	يقوم المصحح بتنفيذ الجزء الباقى من الدالة الحالية بسرعتها العادية ويتوقف بمجرد استرجاع قيمة الدالة إلى الدالة التى قامت باستدعائها
	———	يقوم المصحح بالتوجه إلى العبارة التالية
	Ctrl+Shift+F5	يتسبب هذا الخيار فى بدء تنفيذ البرنامج من البداية
	Shift+F5	يتسبب هذا الخيار فى إيقاف تنفيذ البرنامج

باستخدام الخيارات الموضحة بالجدول السابق، يمكنك مشاهدة تدفق سير كود البرنامج ومشاهدة محتويات متغيراته، وفى كل مرة يشير السهم الأصفر داخل نافذة المحرر إلى العبارة التالية التى سيتم تنفيذها.

تحرير الكود داخل طور تصحيح الأخطاء

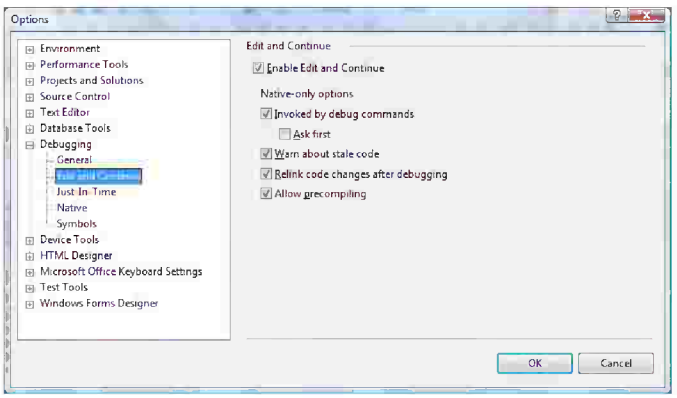
من السمات الجديدة التى بزغت بظهور الإصدار السابق من **Visual C++** إمكانية تحرير الكود داخل طور تصحيح الأخطاء ثم الاستمرار فى عملية تنفيذ البرنامج وهو ما يطلق عليه **Edit and Continue** . فإذا قمت بتعديل كود تطبيقك أثناء توقف مصحح الأخطاء، تلاحظ تمكين الخيار **Apply Code Changes** داخل قائمة **Debug** بحيث يمكنك اختيار هذا الخيار لترجمة تغييرات الكود الجديدة والاستمرار فى تصحيح أخطاء بقية الكود. وعلى ذلك فاستخدامك لهذه السمة يمكّنك من تصحيح الأخطاء أثناء العمل فى طور تصحيح الأخطاء ثم الاستمرار من نفس مكان الكود الذى كنت تقف عنده مع الاحتفاظ بنفس إعدادات متغيراتك وهو ما يكون أكثر إفادة عند العمل مع البرامج الكبيرة والمركبة. لتمكين عملية التحرير والاستمرار أثناء طور التصحيح، تابع معنا الخطوات الآتية:

١. افتح قائمة **Tools** من شريط القوائم ثم اختر **Options** من القائمة المنسدلة، يظهر المربع الحوارى **Options**.

٢. انقر المجلد **Debugging** ومنه اختر **Edit and Continue** من العمود الأيسر بالمربع الحوارى.

٣. نشط مربع الاختيار **Enable Edit and Continue** من العمود الأيمن بالمربع الحوارى (انظر شكل ٢١-١١).

٤. انقر زر **Ok** لإغلاق المربع الحوارى.



شكل ٢١-١١ تمكين عملية التحرير والاستمرار أثناء طور التصحيح