دليل المبتدئين فى المؤشرات

اعداد وترجمة/ محمد ابوزايد

Anyhelpit@gmail.com

فهرس المحتويات

ما هى المؤشرات *pointers*؟

المؤشرات في الأساس مشابه لأية متغير آخر. ومع ذلك ، تختلف عن المتغيرات العادية فى انها بدلا من ان تحتوي على بيانات فعلية فأنها تحتوي على مؤشر إلى موضع المتغير في الذاكرة حيث يمكن الوصول الى المعلومات. هذا مفهوم هام. الكثير من البرامج والأفكار تعتمد على المؤشرات كأساس لتصميمها مثل القوائم الموصولة وطوابير البيانات.

البداية.

كيف يمكن تعريف مؤشر؟ بكل سهوله، كتعريف اى متغير آخر ، باستثناء إضافة نجمة قبل اسمه.مثال، الكود التالي ينشئ مؤشران، وكلاهما يشير إلى متغير عددي صحيح *int* :

```
int* pNumberOne;
int* pNumberTwo;
```

لاحظ البادئة "*p*" فى بداية اسم كل مؤشر؟ هذةعادة يستخدمها المبرمجين للإشارة إلى أن المتغير هو مؤشر لسهولة التفرقة بينة وبين المتغير العادى. الآن ، لنجعل هذه المؤشرات تشير إلى شيء ماء فعلي :

```
pNumberOne = &some_number;
pNumberTwo = &some_other_number;
```

ينبغي أن تقرأ العلامة & على أنها "عنوان الـ" عنوان المغير في الذاكرة بدلا من ارجاع المتغير نفسه(أى قيمة المتغير).في هذا المثال، تم ضبط *pNumberOne* ليساوي عنوان *some_number* ، ولذلك الآن *pNumberOne* يشير الى *some_number*.

تذكران: المتغير العادى له قيمة قد تكون 1 او او *a* او *asd* واية لة عنوان فى الذاكرة يمكن الوصول الى هذة القيمة من خلال عنوانة هذا ما تفعلة المؤشرات انها تحاول ان تغير قيمة المتغير عن طريق عنوانة ولعلك استنتجت انة المؤشر لة ايضا عنوان فى الذاكرة وله قيمة وقيمتة هنا هى عنوان المتغير الذى تريد الوصول الية.

فلو اننا الان نريد الحصول على عنوان *some_number* بأمكاننا استخدام المؤشر *pNumberOne*. ولو اننا نريد الحصول على قيمة *some_number* من خلال *pNumberOne* فأن ذلك عن طريق أضافة النجمة مثل *pNumberOne** . وان الـ * تقرأ على أنها " مكان الذاكرة المشار الية بـ " . بأستثناء وجودها فى الاعلان كما فى السطر *int *pNumber*.

مثال على ما تعلمناة الى الآن. هل استغرق وقت طويل ،اذا واجهتة صعوبة فى فهم هذة المفاهيم نوصيك ان تعيد القرأة مرة أخرى. المؤشرات موضوع معقد ويمكن أن يستغرق بعض الوقت لاتقانه. هنا مثال يوضح الأفكار التي طرحت أعلاه. بكود السى وليس سي++ .

```
#include <stdio.h>
```

```
void main()
{
   // الاعلان عن المتغيرات:

   int nNumber;
   int *pPointer;
   // الآن اعطائها القيم:
   nNumber = 15;
   pPointer = &nNumber;
   // طباعة nNumber:

   printf("nNumber is equal to : %d\n", nNumber);

   // او تعديل الـ nNumber من خلال pPointer:
   *pPointer = 25;

   // . نثبت ان  nNumber قد تغير نتيجة الكود السابق
   // طباعة قيمتة مرة اخرى:
   printf("nNumber is equal to : %d\n", nNumber);
}
```

من خلال قراءة وترجمة المثال السابق. تأكد من فهمك كيف يعمل جيدا. ثم عندما تكون مستعدا واصل القراءة!

فخ يا ذكى.

هل يمكنك معرفة ما إذا كان هناك خطأ في البرنامج التالى ام لا؟ :

```
#include <stdio.h>

int *pPointer;
void SomeFunction();
{
   int nNumber;
   nNumber = 25;
   //  pPointer يشيرالى nNumber:
   pPointer = &nNumber;
}

void main()
{
   SomeFunction(); //  pPointer يشير الى شي ماء
   // لماذا يفشل الكود التالى أى لايعطى القيمة الصحية؟
   printf("Value of *pPointer: %d\n", *pPointer);
```

```
}
```

هذا البرنامج أولا يستدعى الدالة *SomeFunction*، التى تنشئ المتغير *nNumber* و تجعل *pPointer* يشير إليه. ولكن أين توجد المشكلة. عندما تنتهى الدالة من عملها ، يتم حذف *nNumber* لأنه متغير محلي. دائما يتم حذف المتغيرات المحلية التى بداخل الدالة عندما تنتهى الدالة من عملها او يخرج تنفيذ الكود من كتلة الكود الحالية(كتلة الكود هى بين الاقواس { *var1 code* ..} .

وهذا يعني عندما يرجع التنفيذ من *SomeFunction* الى *main* () ، يتم حذف المتغير *nNumber*. وحينها ان *pPointer* يشير في الوقت ذاته الى *nNumber* الذى لم يعد موجودا فى هذا البرنامج(أو ان هذا المكان فى الذاكرة المشار الية لم يعد يخص برنامجنا الحالى). إذا كنت لا تفهم هذا ، قد يكون من الحكمة ان تراجع على المتغيرات المحلية والعامة ومداها (*Local and global variables and scope*) . هذا المفهوم مهم أيضا.

كيف يمكن حل هذه المشكلة؟ الجواب هو باستخدام تقنية تعرف باسم التخصيص الديناميكي للذاكرة *dynamic memory allocation*. انتبة هذا هو الفرق بين سي و سي + +. ونظرا لأن معظم المطورين يستخدمون الآن سي + + فأن الكود التالى يستخدم تقنيات التخصيص الديناميكى للذاكرة فى السي++.

التخصيص الديناميكى *Dynamic Allocation*.

التخصيص الديناميكي ربما يكون شئ اساسي بالنسبة للمؤشرات. يتم تخصيص الذاكرة دون الحاجة إلى تعريف متغيرات ثم نجعل المؤشرات تشير الى المكان المخصص ديناميكيا فى الذاكرة. على الرغم من أن المفهوم قد يبدو مربك، فإنه بسيط . الكود التالي يوضح كيفية تخصيص الذاكرة لعدد صحيح *integer* :

```
int *pNumber;
pNumber = new int;
```

في السطر الأول نعلن عن المؤشر *pNumber*. السطر الثاني يتم تخصص الذاكرة لعدد صحيح *integer* ومن ثم نجعل *pNumber* يشير لهذا المكان الجديد فى الذاكرة. هنا مثال آخر، وهذه المرة باستخدام رقم من نوع *double* :

```
double *pDouble;
pDouble = new double;
```

الصيغة هي نفسها في كل مرة وصعب ان تخطأ في كتابتها. الجزء المختلف فى التخصيص الديناميكي فقط، مع التخصيص الديناميكي لا يتم حذف الذاكرة التي خصصت عندما تنتهى الدالة من عملها، أو عندما يخرج التنفيذ من كتلة الكود الحالي لذا لو إعادنا كتابة المثال أعلاه باستخدام التخصيص الديناميكي للذاكرة ، يمكننا أن نرى أنه يعمل بشكل جيد الآن :

```
#include <stdio.h>
int *pPointer;
void SomeFunction()
{
```

```
   // pPointer يشير الى منطقة ذاكرة جديد من الحجم int

   pPointer = new int;
   *pPointer = 25;
}

void main()
{
   SomeFunction(); // تنفيذ الدالة و pPointerيشير الى مكان الذاكرة المناسب
   printf("Value of *pPointer: %d\n", *pPointer);
}
```

من خلال قراءة وترجمة المثال السابق. تأكد من فهمك لماذا يعمل؟ . عندما يتم استدعاء الدالة *SomeFunction*، فإنه يتم تخصيص الذاكرة ويضبط *pPointer* ليشير إليها. هذه المرة، عندما ينتهى تنفيذ كود الدالة، يتم ترك الذاكرة الجديدة بدون حذف، لذلك *pPointer* لا يزال يشير إلى نفس المكان فى الذاكرة رغم انتهاء عمل الدالة. وهذا ما يسمى التخصيص الديناميكى للذاكرة! . واصل القراءة لمعرفة لماذا لا يزال هناك خطأ جسيم في الكود أعلاه.

ذاكرة جاية وذاكرة رايحة *Memory Comes, Memory Goes*.

هناك دائما خطأ بسيط يمكن أن يصبح خطيرا جدا، على الرغم من سهولة اصلاحة. المشكلة هي أنه على الرغم من ان الذاكرة التي حجزتها باستخدام التخصيص الديناميكي تظل سليمة بدون تغيير وفى الواقع لن تحذف تلقائيا وستظل محجوزة حتى تقوم بأغلاق الكمبيوتر ونتيجة لذلك انك ان لم تقول للكمبيوتر انا لست بحاجة الى هذة الذاكرة فللأسف ستضيع عليك فرصة استخدامها فى البرامج الاخرى. واذا تكرر ذلك سوف يودى فى نهاية المطاف الى توقف النظام عن العمل نتيجة نفاذ الذاكرة فأحذر هذة النقطة مهمة جدا. ان تحرير(حذف) الذاكرة عقب ما تنتهى من استخدامك لها يتم بشكل بسيط جدا فقط كالتالى:

```
delete pPointer;
```

هذا كل ما في الأمر. عليك أن تكون حذرا على ان تمرر مؤشر بشكل صحيح ،وهذا المؤشر يشير فعلا إلى ذاكرة كنت قد خصصتها لة وليس اى مكان ذاكرة عشوائي. ان محاولة حذف ذاكرة تم تحريرها بالفعل أمر خطير ويمكن أن يؤدي توقف البرنامج عن العمل.

نرجع الى مثالنا ولكن تم تعديلة بحيث يحرر الذاكرة التى حجزتها حتى لا تضيع على النظام او البرامج الاخرى فرصة استغلالها :

```
#include <stdio.h>
int *pPointer;

void SomeFunction()
{
// make pPointer point to a new integer

   pPointer = new int;
   *pPointer = 25;
```

```
}

void main()
{
   SomeFunction(); // make pPointer point to something
   printf("Value of *pPointer: %d\n", *pPointer);
   delete pPointer; // تم تحرير مكان الذاكرة بعد الانتهاء من استخدامها
}
```

الفرق سطر واحد من الكود البرمجى ولكنة فرق جوهرى . لو انك لم تحرر الذاكرة فأنة سوف يحدث ما يسمى "*memory leak*" تسرب الذاكرة . وعندما يتم تسرب الذاكرة تدريجيا فلن يتم استخدامها الا اذا أغلق البرنامج.

تمرير المؤشرات الى الدوال.

إن القدرة على تمرير المؤشرات إلى الدوال مفيد جدا ، ومن السهل تعلمها. فلو اننا كتبنا برنامج يأخذ وسيطة رقمية ويضيف اليها قيمة خمسة كما يلى:

```
#include <stdio.h>
void AddFive(int Number)
{
   Number = Number + 5;
}

void main()
{
   int nMyNumber = 18;
   printf("My original number is %d\n", nMyNumber);
   AddFive(nMyNumber);
   printf("My new number is %d\n", nMyNumber);
}
```

للأسف ان المشكلة فى *Number* وسيط الدالة *AddFive* هو نسخة من المتغير *nMyNumber* تم تمريرها الى الدالة وليس المتغير نفسة. ولذلك فأن سطر الكود: *Number = Number + 5* يضيف القيمة 5 الى نسخة من المتغير وذلك بدون تغيير المتغير الذى فى الدالة الرئيسية بأى شئ. حاول تشغيل البرنامج حتى تتأكد من ذلك.

لكى نحل هذة المشكلة يمكننا تمرير مؤشر حيث يوجد الرقم بالذاكرة الى الدالة، ولكن يجب علينا تعديل الدالة لكى تفهم اننا نريد مؤشر الى الرقم وليس نسخة الرقم نفسة. لكى تفعل ذلك نقوم بتغيير *void AddFive(int Number)* الى *void AddFive(int* Number)* وتلاحظ اضافة النجمة فقط. التالى البرنامج بعد عمل التغييرات اللازمة. نلاحظ يجب علينا تمرير عنوان الـ *nMyNumber* بدلا من الرقم نفسة؟ وذلك بأضافة علامة & التى تقرأ على انها عنوان المتغير *nMyNumber*.

```
#include <stdio.h>

void AddFive(int* Number)
{
   *Number = *Number + 5;
}

void main()
{
   int nMyNumber = 18;

   printf("My original number is %d\n", nMyNumber);
   AddFive(&nMyNumber);
   printf("My new number is %d\n", nMyNumber);
}
```

حاول عمل مثال خاص بك لإثبات ذلك. لاحظ اهمية أضافة الـ * قبل *Number* فى الـدالة *AddFive*. هذا يخبر المترجم بأننا نريد ان نضيف القيمة 5 الى الرقم المشار الية بواسطة المتغير *Number* بدلا من اضافة قيمة 5 الى المؤشر نفسة. الشئ النهائى الملاحظ عن الدوال ان بأمكانك ارجاع مؤشر من الدالة ايضاً مثل:

```
int * MyFunction();
```

فى هذا المثال ان الدالة *MyFunction* ترجع مؤشر الى قيمة من نوع *int*.

تذكر ان : من خلال دراستك للدول تعلمت ان الدالة يمكن ان ترجع قيمة فلا تستغرب عندما تعرف ان الدالة ترجع مؤشر ايضا.

مؤشرات الى الفئات

هناك عددا من الأمور ينبغي الانتباه اليها مع مؤشرات، ومنها مع الفئات. يمكن تعريف فئة كما يلي :

```
class MyClass
{
public:
   int m_Number;
   char m_Character;
};
```

ثم يمكنك تعريف متغير من نوع *MyClass* على النحو التالي :

```
MyClass thing;
```

يجب عليك ان تعرف هذا مسبقا. لما لا؟ من خلال قراءتك لهذا المقال. لتعريف مؤشر إلى *MyClass* يمكن عمل :

```
MyClass *thing;
```

كما تتوقع. ثم يجب عليك تخصيص حجم مناسب من الذاكرة وهذا المؤشر يشير الى الذاكرة المخصصة.

```
thing = new MyClass;
```

هذا هو مكان وجود المشكلة : كيف نستخدم هذا المؤشر مع الفئات؟ رائع ربما تريد ان تكتب الـ *thing.m_Number* ولكن لن تستطيع ذلك مع المؤشرات لان *thing* ليس الـ *MyClass* نفسها ولكن مؤشر اليها. لذا *thing* نفسة لا يحتوى على متغير يسمى *m_Number* انما يشيرالى هيكل بيانات يحتوى على *m_Number*. لذا يجب علينا استخدام شئ مختلف حتى نصل الى عناصر الفئة . وهو استبدال (.) النقطة بـ -> . المثال يوضح ذلك:

```
class MyClass
{
public:
   int m_Number;
   char m_Character;
};

void main()
{
   MyClass *pPointer;
   pPointer = new MyClass;

   pPointer->m_Number = 10;
   pPointer->m_Character = 's';

   delete pPointer;
}
```

المؤشرات الى المصفوفات

يمكن ايضا ان نجعل المؤشرات تشير الى المصفوفات. عن طريق:

```
int *pArray;
pArray = new int[6];
```

هذا سوف ينشئ مؤشرا الى *pArray* يشير الى مصفوفة من ستة عناصر. الطريقة الاخرى التى لا نستخدم التخصيص الديناميكى للذاكرة كما يلي:

```
int *pArray;
int MyArray[6];
pArray = &MyArray[0];
```

يمكن ان نستبدل *&MyArray[0]* بـــ *MyArray* . طبعا هذا ينطبق على الصفوفات وكنتيجة من طريقة تصميم المصفوفات فى لغة سي و سي ++. الخطأ الشائع هو كتابة *pArray = &MyArray*; ولكن هذا غير صحيح . لو كتبت هذا فأنك تقول انة مؤشر الى مؤشر الى مصفوفة. والذى ليس بالمرة ما نريد نحن نريد فقط مؤشرالى مصفوفة هل لاحظت الفرق؟

استخدام المؤشرات مع المصفوفات

بمجرد وجود مؤشر الى مصفوفة لديك، كيف يمكن استخدامة؟ حسنا لنفرض ان لديك مؤشرا الى مصفوفة مكونة من متغيرات من نوع *int*. هذا المؤشر بشكل مبدئى يشير الى اول قيمة فى المصفوفة، كما يوضح المثال التالى:

```
#include <stdio.h>
void main()
{
   int Array[3];
   Array[0] = 10;
   Array[1] = 20;
   Array[2] = 30;

   int *pArray;
   pArray = &Array[0]; // او  pArray = MyArray;
  printf("pArray points to the value %d\n", *pArray);
}
```

لكى تجع المؤشر يتحرك الى القيمة التالية فى المصفوفة، يمكن كتابة *pArray*++. ويمكننا ايضا وربما كما توقعت انت تكتب *pArray + 2* الذى سوف ينقل مؤشر المصفوفة بعدد قيمتين الى الامام . الشئ الذى يجب ان تحذر منة ان الحد الاقصى لعناصر المصفوفة هو 3 كما فى المثال وبمأن ان المترجمات لن تحذرك عندما تتخطئ حدود المصوفة لذا لا يجب عليك اطلاقا تجاوز حدود المصفوفة مع المؤشرات والا فى نهاية المطاف قد تتسب فى توقف نظام التشغيل. والمثال التالى يعرض الثلاثة قيم للمصفوفة:

```
#include <stdio.h>
void main()
{
   int Array[3];
   Array[0] = 10;
   Array[1] = 20;
   Array[2] = 30;

   int *pArray;
   pArray = &Array[0];

   printf("pArray points to the value %d\n", *pArray);
   pArray++;
   printf("pArray points to the value %d\n", *pArray);
   pArray++;
   printf("pArray points to the value %d\n", *pArray);
}
```

لاحظ انة كما كان بأمكانك استخدام ++ للأشارة الى العنصر التالى يمكنك استخدام -- ايضا بالعكس.
واستخدام المؤشرات الى المصفوفات مفيد جدا مثلا فى فى الدورات *loops* .

ولاحظ ايضا اذا كان لديك مؤشر الى قيمة مثل: *int* *pNumberSet* يمكنك التعامل معها كما فى مصفوفة. مثال: *pNumberSet[0]* مساوية لـ **pNumberSet*; وبالمثل ايضا ان *pNumberSet[1]* تساوي الـ **(pNumberSet + 1)*.

وكلمة اخير عن التحذير من المصفوفات المخصص لها ذاكرة عن طريق *new* كما فى المثال:

```
int *pArray;
pArray = new int[6];
```

يجب حذف هذة الذاكرة الديناميكية كالتالي:

```
delete[] pArray;
```

لاحظ [] بعد *delete*. هذا يخبر ويطلب من المترجم حذف كل المصفوفة بكل عناصرها وليس عنصر معين منها. يجب عليك استخدام ذلك فى كل وقت تستخدم المصفوفات بهذة الطريقة والا انت تعرف سيحدث ما يسمى بتسرب الذاكرة *a memory leak*.

المراجع *references*

لقد وصلت لى استفسارات كثيرة عن المراجع خلال اطلاع القراء على هذة المقالة ولكن سوف اطرحها فى مقالة منفصلة. على كل حال المراجع تتشابة الى حد كبير مع المؤشرات وفى العديد من الحالات تستخدم كطريقة سهلة بديلة عن المؤشرات. وان استرجعنا مما شرحناة بأعلى لقد المحنا ان علامة & تقرأ على أنها " عنوان الـ " الا فى حالة الاعلان عن المؤشر. في حالة وجودها في الاعلان عن المتغير مثل تلك المبينة أدناه، ينبغي أن تقرأ على أنها "مرجع الى".

```
int& Number = myOtherNumber;
Number = 25;
```

المراجع مثل مؤشر الى *myOtherNumber* ما عدا انة يتم تغيير مرجعيتها اليا (يقصد لست بحاجة الى وضع علامة النجمة امام المؤشر لكى تتمكن من تغير قيمة المتغير) لذا انت تتعامل مع المرجع كما لو كان متغير عادى بدلا من المؤشر . والكود الذى يؤدى نفس وظيفة الكود السابق ولكن بالمؤشرات:

```
int* pNumber = &myOtherNumber;
*pNumber = 25;
```

الفروق الاخرى بين المؤشرات والمراجع هو انة لا يمكنك اعادة ضبط القيمة التى هو مرجع لها. بشكل اخر لايمكنك تغيير ما يشير الية بعد الاعلان عن المرجع. مثلا اخراج الكود التالى هو 20:

```
int myFirstNumber = 25;
int mySecondNumber = 20;
int &myReference = myFirstNumber;

myReference = mySecondNumber;

printf("%d", myFristNumber);
```

وعند يكون فى الفئات . ان قيمة المرجع يجب ان يتم اعدادها فى بانى *constructor* الفئة بالطريقة التالية:

```
CMyClass::CMyClass(int &variable) : m_MyReferenceInCMyClass(variable)
{
```

```
    // constructor code here
}
```

كلمة اخيرة

ملاحظة أخيرة : يجب عدم حذف الذاكرة التي لم يتم تخصيصها بأستخدام *new* مثال :

```
void main()
{
    int number;
    int *pNumber = number;

    delete pNumber; // خطأ - *pNumber لم يتم تخصيص الذاكرة لة بأستخدام new.

}
```

الاسئلة المتكرة والشائعة.

س: لماذا يحدث الخطأ "*Symbol undefined*" عندما استخدم *delete* و *new* ؟

ج : وذلك لان كود الملف مكتوب بلغة السي أى بالامتداد *c* . وبما ان *new* و *delete* من المميزات الجديدة فى لغة سي++ . فلذلك فأننا نؤكد على استخدام الامتداد *cpp* لملف الكود الخاص بك حتى تضمن عدم حدوث الخطأ.

س: ما هو الفرق بين *new* و *malloc* ؟

ج : *new* موجودة فقط بداخل سي++ وتعتبر من الاموار القياسية فى التخصيص الديناميكى للذاكرة. انت لا يجب عليك اطلاقا استخدام malloc بداخل برامج سي++ الا فى حالة الضرورة القصوى. لان- *malloc* ليست مصممة لميزة البرمجة كائنية التوجية *OOP* فى سي++. ان استخدمتها فى تخصيص الذاكرة للفئات هذا سيمنع استدعاء بانى الفئة. هذا مثال واحد على المشاكل التى يمكن ان تحدث. وكنتيجة للمشاكل التى تسببها *malloc* و *free* ولان لهما استخدام محدود. لم يتم مناقشتهم بتفصيل فى هذا المقال. اقترح عليك تجنب استخدامها بقدر المستطاع.

س: هل يمكننى استخدام *free* و *delete* معا ؟

ج : يجب عليك تحرير الذاكرة التى سبق ان قمت بتخصيصها بالطريقة المناسبة. على سبيل المثال: استخدم *free* فقط على الذاكرة المخصصة بواسطة *malloc*. واستخدم *delete* فقط مع الذاكرة المخصصة بواسطة *new* ؟

الملخص:

يجب عليك ان تتذكر ان هذا المقال صعب فهمة بشكل كامل من المرة الاولى. لذا من المناسب قراءة على الاقل مرتين. اغلب الناس لا تفهمة فى الحال. التالى النقاط الرئيسية فى المقال:

1. المؤشرات هى متغيرات تشير الى مكان محدد فى الذاكرة. يمكنك تعريف المؤشر بأضافة النجمة(*) قبل اسم المتغير مثل (*int *number*).
2. يمكنك الحصول على عنوان اى متغير بأضافة علامة & قبل اسم المتغير. مثل

pNumber=&my_number .

3. علامة النجمة * تقرأ بالطريقة " مكان الذاكرة المشار الية بـ " فيما عدا استخدامها فى الاعلان عن المتغير مثل *int *number* .
4. علامة & تقرأ بطريقة " عنوان الـ " فيما عدا استخدامها فى الاعلان مثل *int &number* .
5. يمكنك تخصيص الذاكرة باستخدام الكلمة *new* فى برامج سي++ وليس سي.
6. يجب ان تكون المؤشرات من نفس نوع المتغيرات التى تشير اليها. لذا فأن *int *number* لن يشير الى *MyClass* .
7. يمكنك تمرير المؤشرات الى الدوال.
8. يجب عليك تحرير(حذف) الذاكرة التى قمت بتخصصيها باستخدام الكلمة *delete*.
9. يمكنك الحصول على مؤشر الى مصفوفة باستخدام &*array[0]* .
10. يجب عليك تحرير الذاكرة التى خصصتها للمصفوفة بأستخدام *new* عن طريق *delete*[] وليس *delete*.

هذة المقالة ليست دليل كامل للمؤشرات. هناك القليل من الاشياء التى لم استطع تغطيتها بتفصيل أكثر. مثل: مؤشر يشير الى مؤشر آخر.

دورك انت

بعد ان قرأت المقالة أكيد قد افادك الله منها تذكر ان عليك دور للنهضة وتطوير بلادك اذا كنت تمتك المهارة العلمية فى تخصصك ايا كان فبادرك بتخصيص جزء منة لنقل المعرفة لغيرك وذلك أم بترجمة مقال او تأليفة او الاشتراك فى المنتديات والمساهمة فى تقديم الحلول الى الاخرين حتى تنهض بلادنا بالعلم فلاسبيل لنا الا هو من بعد الله.

رابط مناقشة المقال

لقد نشرت هذة المقالة على موقع الفريق العربى للبرمجة *lhttp://www.arabteam2000-forum.com* وهو موع غنى عن التعريف. واذا كان لديك اى استفسار بخصوص او توضيح يمكنك مشاركتنا على الرابط التالى حيث مكان نشر المقال رابط مناقشة المقال :

http://www.arabteam2000-forum.com/index.php?showtopic=254271

المصادر:

اعداد وترجمة محمد ابوزايد

هذة المقالة ترجمة بتصريف من مقالة منشورة بسنة 2002 على الرابط
http://www.codeproject.com/KB/cpp/pointers.aspx وتاريخ ترجمتها نهاية سنة 2011