المؤشرات مالها وماعليها، نظرة تحليلية مفصلة.

تأليف

هيثم

11 ذو الحجة 1428

مقدمة

يمكن أن توصف المؤشرات بثلاثة كلمات: مميزة، مرنة وخطيرة في ان واحد...
فهي مميزة لأن أغلب ان لم يكن كل لغات البرمجة (ماعدا السي والسي++) لاتعطيك الحرية بفعل ماتريد كيفما تريد كما هو في السي++. فمبرمج السي++ له الحرية المطلقة في التعامل مع المؤشرات أو عدم التعامل على مستوى المؤشرات، وطبعا التعامل يختص بالتعامل مع الذاكرة ليس الا.
نجد ان الفيجوال بيزك لا تعطي المبرمج حرية أبدا كمبرمج السي++ نظرا لخطورة التعامل مع المؤشرات وادارة الذاكرة يدويا. وأيضا لغة الجافا التي اشتقت من السي++ لاتعطي المبرمج الحرية المطلوبة. وللغة الجافا ململم نفايات[1] ويمكن الاختصار له ب GC مهتم بالغاء تخصيص كميات الذاكرة المحجوزة يدويا بشكل الي. ومن الملاحظ أن أكثر أدوات لغة الجافا مبيعا تلك المهتمة باكتشاف الاخطاء وخصوصا (بطء عمل التطبيقات) بسبب ململم النفايات. فهذا الأخير يتسبب بعمليات تنظيف دورية للذاكرة مما يتسبب ببطء البرنامج.
أما بالنسبة للمرونة فهي ماتوصف به أقوى لغة برمجة على الاطلاق وهي السي++ فمن أهم ميزاتها ميزات لتعامل الحر جدا مع الذاكرة. حيث تعطي بعدا جديدا كليا لهندسة البرمجيات ككل. سنتطرق في دروس قادمة عن أمثلة لحالات مرنة جدا كان من الصعوبة وربما إستحالة أن تتم دون المؤشرات.
فبتكامل المؤشرات مع الوراثة مع الفئات مع الوراثة وتعدد الاشكال نصل لمراحل متقدمة جدا في هندسة البرمجيات ككل.
ثالث الخطورة تتمثل في خطورة التعامل المباشر مع الذاكرة، له سيئاته طبعا بالنسبة لعدم المتمرسين والحذرين.
نعلم أنه يجب الغاء تخصيص أي ذاكرة يتم حجزها يدويا بالغاء يدوي أيضا والا سيتم توقف البرنامج أو إنتهاك ذاكرة النظام انتهاكا سيئا يضر بالنظام ككل، وغالبا ماينسى المبرمج تحرير الذاكرة التي أنشئها. واذا لم ينسى فقد يحرر الذاكرة في وقت غير مناسب يمكن أن يتسبب في شلل البرنامج ككل مما يؤدي الى أصعب أنواع الاخطاء على الاطلاق الا وهي أخطاء الإستثناءات وانتهاكات الذاكرة.
يوجد نوع اخر من الاخطاء المنطقية التي لاتتسبب في وقوف البرنامج انما حدوث نتائج غير مرغوب فيها وهي أيضا معقدة وصعبة الاكتشاف، ولهذا قامت مايكروسوفة أخيرا

[1] ململم نفايات: Garbage Collection

بانتاج طقم الدوت نت لأسباب كثيرة منها مشكلة التعامل مع الذاكرة بهذه الطريقة المباشرة خصوصا لمبرمجي السي++، حيث أدخلت مايكروسوفة ململم نفايات خاص بمنصة الدوت نت لكن يتم تفعيله في السي++ دوت نت بشكل إختياري فلم يتم الغاء الحجز اليدوي في نسخة السي++ الخاصة بمنصة الدوت نت والتي تدعى *بشيفرة سي++ /المدارة*[2]. فكل ماحدث ان زادت مرونة السي++ في نواح جديدة ولكن للأسف تم الغاء يعض ميزات المرونة الاخرى لكي تتدنى من كل من لغتي السي شارب و الVisual Basic.Net. قد ندخل في موضوع ال Manged C++ لاحقا.

إن مشكال الادارة اليدوية للذاكرة أو التعامل على المستوى الادنى مع المؤشرات وجدت صدى جيدا في ماكروسوفة ورأت أن تنتهج منهج صانعي لغة الجافا وتدير الذاكرة بشكل آلي بسبب أن محاسن الادارة اليدوية أقل من المساويء؛طبعا للمبتدئين. فأدخلت ململم النفايات ليتم الاختيار بين التحرير اليدوي للذاكرة أو التحرير الآلي أو كليهما معا وهذا طبع بالنسبة للمؤشرات.

والمؤشرات ليست إلا متغيرات تحمل قيم عناوين في الذاكرة وليست قيما حقيقية كالأرقام والحروف وماشابه ذلك:

```
int *q;
```

مؤشر ونحجز له ذاكرة قبل ان نتعامل معه على النحو التالي:

```
q = new int;
```

المتغير q حاليا يحمل قيمة لعنة في الذاكرة، اما اذا كتبنا:

```
int w;
```

فيكون قد حجز لنا الحاسوب بشكل الي مكانا في الذاكرة لمتغير واحد صحيح وال w تحمل قيمة لعدد صحيح وليس مؤشرا.

[2] *بشيفرة سي++ /المدارة*: Manged c ++

الفصل الأول

سنحاول في هذه الجلسة أن نفهم الفرق بين:

1) المتغيرات التي تحمل قيما كالأعداد والحروف وبين المتغيرات التي تحمل عناوين في الذاكرة.

2) ثم اذا كان لدينا متغير يحمل قيمة كيف يمكننا أن نعرف عنوانه في الذاكرة.

واذا كان لدينا متغير يحمل عنوان كيف يمكننا أن نحصل على القيمة التي في ذلك العنوان، اي العملية العكسية لكل منهما. اذا فهمنا هاتين النقطتين سنكون قد خطونا خطوة كبيرة في الفهم الصحيح للمؤشرات.

وأهمس في اذانكم قبل ان نبدأ .. أنه بعد تعلم المؤشرات بشكل جيد يجب علينا أن نحاول استعمالها كل ماأمكن ذلك في برامجنا لكي نعتاد عليها. ثم يجب أيضا أن تتم مراجعة وقراءة المؤشرات برمتها كل 6 شهور على الاقل لكي ترسخ في الذهن بشكل مناسب. الان فالنذهب الى النقطة الاولى.

الفرق بين المتغيرات التي تحمل قيما وتلك التي تحمل عناوين، في الصيغة الكتابية تكون على النحو التالي؛ المتغيرات العادية:

```
int y;
```

المؤشرات:

```
int *w;
```

الفرق كما هو واضح فقط في علامة * التي تعني أن هذا المتغير x سيحمل عنوانا وليس قيمة. الى هنا سنذهب الى النقطة الثانية ونتعرف على باقي الفروق لاحقا. اذا كان لدينا متغير يحمل قيمة كيف يمكننا أن نعرف عنوانه في الذاكرة ؟؟

```
int y=10;
```

مثال 1:

من الملاحظ أن قيمة y هنا هي 10. ولكن ماهو عنوان هذا المتغير في الذاكرة؟

لمعرفة ذلك وطباعة قيمة العنوان على الشاشة:

```
cout<<" y value is : "<<y;
cout<<"address is : "<<&y;
```

لاحظ لل & هنا تعني أوجد عنوان الشيء الذي يلي الحرف & يعني هنا أوجد عنوان y. سيقوم الجهاز بطباعة قيمة العنوان المحجوز للمتغير y. وستكون النتيجة قيمة ستعشرية وقد تكون على الشكل التالي:

```
int y value is :10
int y address is 0x8f51fff3
```

لكن لن تكون القيمة السابقة ثابته لكل جهاز فحتما ستختلف من جهاز لأخر لأن البرنامج يقوم بحجز القيم في أماكن تتعلق بالنظام ككل. أما بالنسبة للمتغيرات التي تحمل عناوين "المؤشرات"، واذا كان لدينا متغير يحمل عنوان كيف يمكننا أن نحصل على القيمة التي في ذلك العنوان ؟؟ الامر كالتالي:

```
int *x;
```

لن نمهد للمتغير x بقيمة حاليا وانما لاحقا سنعرف كيف نمهد للمتغيرات.

```
*x = 50;
cout << " x value is : "<< *x;
cout << " x address is : "<< x;
//x value is 10
//x address is 0x8f51fff3
```

مثال 2

لاحظ هنا أولا أننا أعطينا x قيمة 50 لكن x عنوان وليس قيمة لذا فاننا وضعنا العامل * قبل المتغير x لنرمز لى اننا نريد قيمة المتغير الذي يلي العامل *. وهنا يعني نريد قيمة x لنعين لها المقدار 50.

لاحظ في النتائج على الشاشة أن خرج القيمة سيكون50 وهي القيمة لل x* والعنوان هو x فقط.

وهذا عكس مايحدث للمتغيرات المعرفة بأنها عادية وتحمل قيما. عندها ملخص النتائج

العامل & يستخدم لاستخراج عنوان من متغير في الاصل يحمل قيمة

والعامل * يستخدم لأستخراج القيمة من متغير هو في الاصل يحمل عنوانا.

أخيرا هناك مفاجأة !! المثال 2 لن يعمل وسيعطيك المصرف بأن هنا انتهاك للذاكرة.

مالسبب في ذلك؟ كما يبدو للعيان المثال مباشر تماما ولاتعقيدات حوله

لكن فلنعلم انه عند انشاء متغير عادي يحمل قيمة يقوم الكمبيوتر بشكل الي بحجز مكان في الذاكرة لهذا المتغير الجديد. لهذا السبب المثال الأول سيعمل بشكل جيد لأنه عندما عيينا القيمة 10 للمتغير y كان المتغير y له مكان صالح في الذاكرة. أما المثال الثاني فعندما عينا القيمة 50 لتوضع في المتغير x * كقيمة، لم نكن قد أنشأنا مكانا في الذاكرة للمتغير x بعد لأن الامر ؛ int * x لايفعل شيئا ولايحجز مكانا صالحا في الذاكرة للمتغير x. وانما فقط يخبر المصرف أن هناك متغير اسمه x يمكن أن يحمل عنوانا. أما عملية الحجز فيتركها المصرف لك كليا ولك الحرية في ذلك. لكن تذكر لاتعين

قيمة لمتغير مؤشر قبل أن تحجز له مكانا في الذاكرة. وحجز الذاكرة سيكون يدويا كما ذكرت ونستخدم لذلك أمرا أساسيا جديدا وهو new وسصبح لونه بالازرق في مصرف لغة السي ليشير الى أنه كلمة اساسية والصيغة كالتالي:

```
int *x;
x=new int;
```

وهكذا اذا أكملنا المثال 2 ستعرض النتائج كما هو موضح في الخرج لأننا الان حجزنا مكانا صالحا في الذاكرة للمتغير x.

أخيرا يجب أن نعين متغير عنوان الى عنوان ومتغير قيمة الى قيمة. ولايمكن ان نعين قيمة الى عنوان أو عنوان الى قيمة. يعني بلغة الامثلة:

```
int * x;
x = new int;
int y;
x = y; // ERROR x is address and y is Value
x= y; // Good y is Value and *x is value *
x = &y; // Good x is address and &y is address
```

مثال 3:

الخلاصة

والان وبعد أن عرفنا هذه المعلومات، يأتي السؤال المهم ولماذا نريد أن نعرف عنوان متغير في الذاكرة؟

لن أجيب الان وانما في الجلسة القادمة ولكن يجب لكي تفتح ذهنك أن تعرف أن المؤشرات هي المصفوفات. كيف؟

سنعرف في الفصل القادم.

الفصل الثاني

عرفنا في الفصل السابق كيف يمكننا انشاء مؤشرات وكيف نحجز لها مكان في الذاكرة وكيف نعين قيما لتخزن في داخل تلك العناوين. الان يجب أن نعلم أنه يمكننا حجز أكثر من مكان في الذاكرة، فمثلا:

```
int * x;
x = new int;
```

مثال 1:

حجزنا موقعا واحد فقط في الذاكرة لهذا المتغير x. في المتغيرات العادية يمكننا أن نحجز متغير واحد أو أكثر باستخدام المصفوفات:

```
int z[10]; //Array
```

مثال 2:

المتغير z يمكنه تخزين 20 قيمة من النوع الصحيح. كيف يمكن أن نفعل ذلك في المؤشرات:

```
int *x;
x = new int[10]; // Our Pointer
```

مثال 3:

هنا حجزنا 10 خانات للمتغير x وعملية الحجز هذه تحجز الاماكن في الذاكرة بحيث تكون متتالية في قيم العناوين بفرق 4 بايت بين كل عنوان والاخر " 4 بايت هو حجم ال int ". والمفاجيء أن طريقة استعمال المصفوفات والمؤشرات متشابهه تماما وأقصد متشابهه بالنسبة لطريقة تعين المتغيرات لكل منهما كالتالي:

بالنسبة للمثال 2 الذي يستخدم المصفوفات سنمرر القيمة 10 للثلاثة عناصر الأولى:

```
z[0] = 10;
z[1] = 10;
z[2] = 10;
```

مثال 4:

وبالنسبة للمثال 3 الذي يستخدم المؤشرات:

```
x[0] = 10;
x[1] = 10;
x[2] = 10;
```

مثال 5:

المفاجيء أن لهما نفس طريقة الاستعمال والمفاجيء أكثر أن المصفوفات هي المؤشرات تماما، لكن بفرقين بسيطين.

قبل أن نعرف الفرقين يجب أن نعرف أننا يمكن أن نكتب المثالين السابقين بطريقة أخرى وهي منطقية تماما لمن فهم الجلسة الاولى:

```
*(z+0)=10;
*(z+1)=10;
*(z+2)=10;
```

مثال 6:

وبالنسبة للمؤشرات:

```
*(x+0)=10;
*(x+1)=10;
*(x+2)=10;
```

مثال 7:

مالعلاقة بين المثالين السابقين وبين الذين قبلهما؟؟

التدوين[2]zوالتدوين (z+2)* متشابهين تماما كما هو ملاحظ، لكن كيف حدث ذلك؟

قلنا سابقا أن الغرض من العامل * هو استخراج القيمة التي في العنوان.

يعني x* تعني القيمة التي في العنوان x. "وهي تساوي (x+0)*" وقد قلنا سابقا أن العناوين المحجوزة بالمؤشرات أو المصفوفات تحجز على التوالي بفرق عدد بايتات من نوع المصفوفة أو المؤشر.

هاقد استخرجنا القيمة الموجودة في العنوان الاول x. لكن كيف نستخرج باقي القيم التي في باقي خانات المؤشر الـ10 التي حجزناها في المثال 3؟؟

الطريقة سهلة، وهي بأن نقول استخرج القيمة التي في العنوان التالي للعنوان x. كالتالي:

```
*(x+1);
```

حيث أن العنوان الذي يلي العنوان x هو x+1 باستخدام العامل * نستخرج القيمة التي في هذا العنوان "لكن لاحظ يجب استخدام الاقواس". القيمة التي في العنوان الذي يليه (x+2)* وهكذا ...

يمكنك أن تستخدم هذا النوع من الصيغ أو الصيغة الاساسية للمصفوفات. لكنك في بعض الاماكن الضيقة ستفضل تدوين ال*.

الان من أوجه الشبه أيضا بين المؤشرات والمصفوفات طريقة استعمالهما للتمرير الى الدوال. لنفرض ان لدينا الدالة التالية:

```
int Function(int W[]) // Our Function
{
 cout << W[0];
 cout << W[4];
 cout << W[6];
}
```

مثال 8:

تقوم الدالة بطباعة العنصر الاول والرابع والسادس من المصفوفة. اذا كانت مصفوفتنا عادية كما في المثال 2 سنمررها كالتالي:

```
Function(Z);
```

مثال 9 :

فالمتغير z هنا هو عنوان أول عنصر في المصفوفة z. واذا كنا نستخدم المصفوفات التي تعمل بالمؤشرات كما في المثال 3 سنمررها كالتالي:

```
Function(x);
```

مثال 10:

لاحظ لافرق بين النوعين اطلاقا. فأريد أن أقول أن المؤشرات والمصفوفات متشابهتان في أمور كثيرة لكن يمكن أن نميز الفرق بينهما بالثلاثة فروق التالية.

1)عنوان مصفوفة المؤشرات ديناميكي. أما عنوان المصفوفة العادية ثابت،

2) حجم مصفوفة المؤشرات ديناميكي ولايشترط معرفته قبل وقت التشغيل. أما حجم المصفوفات العادية فهو ثابت ويجب معرفته قبل وقت التشغيل.

3) يمكن الغاء الذاكرة الخاصة بمصفوفات المؤشرات في أي وقت من البرنامج، واذا نسيت الغاءها فلن يلغيها لك البرنامج الا بعد انتهاءه وتوقفه عن العمل. أما الذاكرة الخاصة بالمصفوفات العادية فلا يمكنك الغاءها متى تشاء فالغائها يتم بشكل الي بعد الخروج من مدى الرؤية الخاص[3] بها. للثلاثة أسباب الفائتة اخترعت المؤشرات لتزيد من مرونة البرمجة بشكل فعال جدا. ويمكنكمنالان أن تلغي جميع استخداماتك للمصفوفات العادية وتستعمل بدلا منها تدوين المؤشرات.

الان سنشرح الفرق الأول:

عنوان مصفوفة المؤشرات ديناميكي. أما عنوان المصفوفة العادية ثابت. لنأخذ المثال التالي الذي سيوضح تماما المقصود:

[3] مدى الرؤية: SCOPE

```
int *x;
x= new int [10];
int z[20];
```

مثال 11:

الان يمكننا كتابة:

```
x=z;
```

ولايمكننا كتاية:

```
z=x;
```

لأن عنوان ال z سيكون ثابتا طوال فترة تشغيل البرنامج. وسعطى للمتغير z فقط وقت إنشاء المتغير "أي وقت التعريف". أما ال متغير x فعنوانه يعطى وقت التعريف لكن يمكن تغييره في أي لحظة، كما في المثال أعلاه قمنا بتعيين عنوان جديد للمتغير x كما في x = z ؛ فلن يعترض المصرف لأنه يمكنك تغيير العنوان كما تشاء لأنك تتعامل مع مؤشر.

وأخيرا قبل أن ننهي هذه الجلسة مالسبب الذي يدعوك لتغيير عنوان المصفوفة؟؟

لنفرض مثلا أنك تريد أن تنسخ مصفوفة كاملة الى مصفوفة فارغة أخرى. يمكنك فعل ذلك بحلقة for بحيث تمر على جميع العناصر في المصفوفة وتعينها الى مقابلاتها في المصفوفة الفارغة، لكن التدوين يغنيك عن كل هذا.

```
x=z;
```

فالذي حدث هنا أن المصفوفة x أصبحت محتويات عناصرها هي نفسها التي في المصفوفة z. فالذي حدث هنا أن عنوان المصفوفة x أصبح هو نفسه عنوان المصفوفة z، لذا فان عنوان أول عنصر للمصفوفة x سيكون هو نفسه في ال z والثاني هو الثاني هناك والثالث كذلك لكل عناصر المصفوفة. وهذه الطريقة سريعة جدا في نسخ

الخلاصة

المصفوفات لكن لها استعمالاتها الخاصة ولها عيوبها سنذكرها في الجلسة القادمة.

وسنشرح الفرقين الباقيين بين المؤشرات والمصفوفات لكي نغوص بعدها في الجلسات التي تليها أكثر في عالم المؤشرات.

الفصل الثالث

الفرق الأول

توقفنا في الفصل السابق عند طريقة تعيين عنوان مؤشر الى مصفوفة لنسخ المصفوفة الى مصفوفة المؤشر، ذكرنا أن لها استخدامات في مناطق خاصة و ذكرنا أن الحسنة في ذلك هي السرعة المذهلة ولكن العيب:

```
int *q = new int[12];
int w[12];

w[0] = 1;
w[1] = 1066;
w[2] = 6;
w[3] = 76;
w[4] = 323;

q = w;

void Set_Data(int *z)
{
  z[0] = 10;
  z[1] = 10;
  z[2] = 10;
}
```

مثال 1:

سنستدعي الدالة أعلاه بعد عدة أسطر من الشيفرة ونعلم أن قيم المصفوفة q أصبحت هي قيم المصفوفة w. وقررنا في هذه اللحظة أن نستدعي الدالة Set_Data لنعين قيم جديدة لأول 3 عناصر في المؤشرة q.

```
void Set_Data(q)
```

المفاجيء هنا اننا غيرنا قيم المصفوفة المؤشرة q، لكن الأمر السيء هو أن المصفوفة w ستتغير أيضا مع ان العلاقة بين المصفوفتين q و w كانت في السابق.

الى أن تعين قيمة عنوان لاخر يعني أن لهما نفس الخانة في الذاكر ة مما يعني أن أي تغيير في أي من المتغيرين سيسبب في تغيير الاخر وليس ذلك هو المرغوب دائما. لذا سنستعين في بعض الأحيان بالنسخ العادي اي نسخ كل عنصر الى مقابله في المصفوفة الأخرى."طبعا باستخدام حلقة For" لتسهل العملية.

عموما تمرير المتغيرات بالعنوان له فوائد كبيرة عند تمرير الكائنات الكبيرة والضخمة الى احدى الدوال، حيث ستستهلك عملية النسخ العادية الكثير من الوقت. لذا يلجأ المبرمجون عادة الى التمرير بالعنوان في نسخ الكائنات"Classes "

الفرق الثاني

حجم مصفوفة المؤشرات ديناميكي ولايشترط معرفته قبل وقت التشغيل. أما حجم المصفوفات العادية فهو ثابت ويجب معرفته قبل وقت التشغيل. وهي المشكلة الأزلية التي لايمكن حلها مطلقا الا بالمؤشرات. حيث اذا أردنا من المستخدم ادخال عدد من الأرقام، وسنقوم بترتيبها لاحقا. ولكننا لانعلم عدد الأرقام الذي سيدخلها المستخدم. يجب طبعا أن نعلم عدد الأرقام المراد تخزينها. فعندها سنطلب من المستخدم قبل ان يبدأ في عملية ادخال الارقام أن يدخل عدد ها لنتمكن من أن نحجز لها مكانا لاستقبالها في الذاكرة.

يوجد حل عملي أكثر بحيث لانطلب من المستخدم ان يدخل عدد الأرقام بل يبدأ مباشرة بادخال الأرقام سنتطرق له لاحقا.

نفرض أن المستخدم سيدخل عدد الأرقام في المتغير x. بعد ادخال عدد الأرقام سنحجز بها عدد من الخانات في الذاكرة كالتالي:

```
int x;
cin >> x;
int *q = new int[x];
```

مثال 2:

هكذا تم حل المشكلة فالمصفوفة المؤشرة q حجمها سيناسب عدد الأرقام التي ستدخل فيها. الان يأتي سؤال مهم؛ لماذا لانستخدم المصفوفات العادية في ذلك، أي تكون:

```
int x;
cin >> x;
int q [x];
```

مثال 3:

المثال 3 لن يعمل، وسيقوم المصرف باصدار خطأ بأن حجم المصفوفة q غير معروف لأن المصفوفات العادية يجب معرفة حجمها وقت تنفيذ البرنامج وليس وقت استخدام البرنامج. أما المؤشرات فلا مشكلة لديها في ذلك.

يعود السبب في ذلك الى ان المصفوفات العادية يتم تخزينها بالنسبة للبرنامج في ذاكرة المكدس الخاص بالبرنامج. أما مصفوفات المؤشرات فانها تخزن في الذاكرة التكويمية. الان عرفنا مايجب فعله ومالايجب.

الفرق الثالث

يمكن الغاء الذاكرة الخاصة بمصفوفات المؤشرات في أي وقت من البرنامج، واذا نسيت الغاءها فلن يلغيها لك البرنامج الا بعد انتهاءه وتوقفه عن العمل. أما الذاكرة الخاصة بالمصفوفات العادية فلا يمكنك الغاءها متى تشاء فالغائها يتم بشكل الي بعد الخروج من مدى الرؤية الخاص بها.

بداية مالسبب الذي يدعوك لالغاء ذاكرة قمت بتخصيصها لبيانات معينة ؟؟

السبب دائما مايرجع الى استهلاك موارد الذاكرة مما لاطائل له؛ يعني يمكن في لحظة معينة أن لانحتاج الى 1000000 متغير محجوز في مصفوفة من نوع int أي اننا سنحجز 4 ميجابايت من الذاكرة. فبعد ان لانحتاج لهذه المتغيرات بتاتا يجب أن نحاول أن نقوم بالغاءها لنزيد من فعالية البرنامج ولترك مجال الذاكرة للمتغيرات الجديدة.

في هذا السيناريو لو كنا نستخدم المصفوفات العادية لأنتظرنا القدر ليأتي وينهي الدالة التي تم حجز المصفوفة داخلها ليقوم بتدمير المصفوفة. "بعد الانتهاء من تنفيذ الدالة " قرر مصممو السي++ ايجاد حل لهذه الحماقة فقامو باختراع المؤشرا لتعطي حرية وقت حذف النتغيرات.

```
int *q=new int[10000000];
```

مثال 4:

بعد الفراغ من المصفوفة المؤشره أعلاه سنقوم بحذفها كالتالي:

```
delete q[];
```

لتعلم أن delete هو أمر اساسي جديد سيتغير لونه للأزرق وهو مضاد للأمر الاساسي new الذي نحجز به الذاكرة.

بعد السطر أعلاه "سطر ال delete" سيتم تحرير الذاكرة فورا ولن يعود هناك وجود في الذاكرة للمتغيرات التي كانت موجودة في المصفوفة المؤشرة بتاتا.

لاحظ في صيغة ال delete أن المعقفات [] موجودة قبل المتغير q، وهذا يعني أننا نريد حذف كل المصفوفة وطبعا بعد حذف المصفوفة وقمنا بتعيين قيمة جديدة لأحد العناصر كالتالي سيحدث خطأ انتهاك للذاكرة. لأننا أعدنا المصفوفة لفترة ماقبل التاريخ. فيجب مرة أخرى قبل استخدام المصفوفة أن نقوم بحجز خانات لهذه المصفوفة:

```
q=new int[1233333333];
```

هذا شرح مختصر جدا للفرق بين المصفوفات والمؤشرات ...

الخلاصة

سندخل في الأيام التالية لبعض المواضيع الأكثر حماسا. وسيتعلق الأمر بالمتغير void والكائنات ومصفوفات التي تخزن قيما لمؤشرات ونأخذ نظرة سريعة على اللوائح المربطة[4] ثم سنتناول العلاقة بين الكائنات والوراثة وتعدد الأشكال والمؤشرات. ونعرف كيف يمكننا أن ننشيء مؤشرا لمؤشر. بحيث نقوم بتخزين عنوان العنوان. وكان الله في العون. وسنحاول بقدر الامكان أن نجعل هذه الجلسات ناضجة لتكون مرجعا لابأس به لمن كان لديه لبس في موضوع المؤشرات.

[4] اللوائح المربطة: Link List

الفصل الرابع

سنتحدث في هذا الفصل عن أمرين مهمين في المؤشرات:

الأول: اذا أردنا من المستخدم أن يدخل اسما في متغير طبعا من النوع char. لكننا لانعرف طول الاسم الذي سيقوم المستخدم بادخاله. في هذه الحال يلجأ المستخدم المبتديء لاستعمال المصفوفات وجعل حجم المصفوفة كبير لدرجة حتى تسع لأي اسم مهما كان.

طبعا لهذه الطريقة سيئة أنها تبذر الذاكرة بشكل فعال عند استخدام عدة أسماء.

الحل: الحل في أن نجعل المصفوفة عبارة عن مؤشر ونحجز للمؤشر الحجم المطلوب في وقت التشغيل "هي ميزة للمؤشرات على المصفوفات.

يمكن بأن ننشيء مؤشرين ونحجز للأول حجما كبيرا حيث يتسع لأي اسم مهم كان كبيرا ونجعل المستخدم يدخل فيه الاسم. ثم يحسب حجم الاسم المدخل. ويحجز للمؤشر الثاني بهذا الحجم ويضع في الاسم، ويمسح المؤشر الاول. وكلما أردنا أن ندخل اسما جديدا كررنا هذا السيناريو. وهي طريقة جيدة لكن يوجد ماهو أفضل.

لنفرض اني أريد من المستخدم أن يدخل عدة أسماء وليس واحدا فقط وطبعا لانعرف حجم وطول كل اسم. لو استخدمنا المصفوفات العادية فقط لكونا مصفوفة ثنائية الأبعاد؛ البعد الأول لعدد الاسماء والثاني لعدد الاحرف في كل اسم:

```
char *X[200][3];
```

مثال 1:

هنا حجزنا 200 خانات في الذاكرة لـ200 اسم، وكل اسم طوله 30 خانة.

هذه الطريقة أيضا ليست عملية لأن عدد الاسماء كبير لدرجة قد تبذر الذاكرة بشكل كبير.

إذا الحل مرة أخر باستخدام المؤشرات، لكن لاحظ توجد مشكلتين هنا؛ عدد الاسماء وطول أحرف كل اسم. لو اكتفينا بحل الثانية فقط سنكون شيئا يدعى *مصفوفة المؤشرات*. وفي هذه الحالة لن نستطيع أن نعالج مشكلة عدد الاسماء حيث سيكون عدد الاسماء ثابت سنقوم بادخاله:

```
char *X[200];
```

مثال 2:

لاحظ * قبل اسم المتغير X هذا يعني أننا سنكون مصفوفة حجمها 200 عنصر هذه المصفوفة ستخزن عناوين وليس قيما كما في المعتاد !! هذه هي مصفوفة المؤشرات. بالمقارنة مع مثال المصفوفة ثنائية البعد يمكننا تشبيهها ان البعد الاول هو 200 والثاني ديناميكي في حالتنا. أي سنقوم بادخاله لاحقا "وهي ميوة للمؤشرات".

سنقوم بادخال طول الاسم كل مرة قبل أن نخزن الاسم في المؤشر بهذه الطريقة سنكون قد وفرنا الكثير من الذاكرة بدل أن نفترض ان طول كل اسم 30 خانة كما في مثال المصفوفة.

بعد التعريف أعلاه يكون الاستعمال كالتالي:

```
X[0] = new char[18];
cin >> x[0];
X[1] = new char[22];
cin >> x[1];
```

وكذا حتى ندخل جميع لاسماء، طبعا سنستخدم حلقة لفعل ذلك.

لنفرض أنني أريد أن لأعرف أيضا عدد الاسماء حتى أخصص الذاكرة بأفضل صورة. في هذه الحال يجب علي أن أحل المشكلة الاولى وهذه المشكلة سويا. طبعا عدد الاسماء ديناميكي اذن سأستخدم مؤشرا لأحجز العدد المناسب لعدد الاسماء في وقت التشغيل. لكن كيف أعل ذلك ؟؟ اليك هذا التدوين الذي قد يبدو غريبا للوهلة الاولى:

```
char **X;
```

مثال 3:

وهذا مايدعى بمؤشر المؤشر أو عنوان العنوان. في هذه الحال سننشيء عدد خانات ديناميكي لنخزن فيه عناوين فقط. الان كيف يمكننا أن نحجز لهذا الشيء عدد الخانات؟

```
char **X;
X = new char* [200];
for ( int q=0; q < 200; q++)
X[q] = new char[30];
```

مثال 4:

الان المثال أعلاه أصبح جاهزا ليعامل كما يعامل المثال 1، لكن بالطبع يمكن أن أغير في الحجم كما أشاء بحيث يكون متغير وليس ثابتا. لاأخفيكم سرا أن استعمال تدوين مؤشر المؤشر نادر الاستخدام جدا، لكن هناك حالات عصيبة في البرمجة قد تضطرك الا فعل ذلك. يمكنك الاستغناء عن هذا التدوين لو استعضت عنه بانشاء مشر ذذو بعد واحد وأنشأت اخر في متغير مختلف ليكون مجرد فهرس للأول. بهذ الطريقة ستكون على الطريق الصحيح، قد تتحير فعلا في بعض الحالات النادرة عن كيفية انجاز مهمة بدون مؤشر المؤشر.

كنت في أحد المرات أكتب برنامجا للعبة شطرنج، ووصلت الى مرحلة جعل الكمبيوتر يلعب مع المستخدم. وكانت المشكلة في انشاء شجرة من الاحتمالات لكل دور للاعبين، بحيث يتوقع الكمبيوتر جميع الخطوات المقبلة ويحدد الافضل منها. اضطررت لكتابة البنية التي تخزن الاحتمالات لدراستها بحيث كل احتمال يكون في داخله عدد من الاحتمالات لللدور التالي وهذه لم تكن أبدا بالسهولة التي أقوم فيها بكتابة مؤشرات متداخلة.

بالنسبة للعبة وصلت الى مستوى شبه ناضج وبدأ الكمبيوتر "بقدرة قادر" بالتفكير وتوقع النتائج ل 4 أدوار، وهي لابأس بها. حيث يكون عدد الاحتمالات الكلي كبير جدا قد يصل 10 مليون احتمال "لخمسة أدوار".

كان الجهاز وقتها يحسب 180 ألف احتمال في الثانية "وهو رقم سيء بالنسبة لألعاب الشطرنج الاخرى" واكتشفت السبب بعد فترة لكن بعد فوات الاوان وسأذكر السبب لاحقا وهو يتعلق بالمؤشرات.

فقط أحببت أن أوضح مثال قد يكون واقعي لاستخدام المؤشرات المتداخلة و قبل المواصلة أقول أنه يمكنك أن تنشيء مؤشر لمؤشر المؤشر وزد على ذلك كما تشاء:

```
int ****W;
```

مثال 4:

لكن لن تحتاج لذلك الى اذا أحببت أن تقوم ببعض أعمال فرانكنشتاين !!!

VOID هو نوع المتغيرات العقيم الذي لايصلح لشيء في عالم المتغيرات، لكنه كبير هنا في عالم المؤشرات. إذا أردنا أن نعين عنوان لعنوان أي متغير مؤشر الى متغير مؤشر اخر يجب أن يكون كلا المتغيرين من نفس النوع:

```
int *x;
int *e;
float *w;
float *y;
char *t;
char *u;

*e = 10;
x = e; // OK
x = w; // ERROR Cant covert From float to int
w = y; // OK
t = u; // OK
w = t; // ERROR cannt convert fromchar to float
```

مثال 5:

وهكذا كما في المثال أعلاه يجب ان يكون تعيين المؤشر الى نفس نوعه إلا void.
يمكن لل لمؤشر ال void أن يقبل أي نوع من الانواع الاخرى:

```
void *R;
int *q;
double *u;
char *c;

R = q; // OK
R = u; // OK
R = c; // OK
```

مثال 6:

لاحظ أني قلت مؤشر ال void وليس متغير ال void لأنه لايمكنك انشاء متغير من النوع void أقصد متغير عادي ليس مؤشرا كما يلي void F.
المثال أعلاه سيعطي خطأ من المصرف بأنه لايمكن إنشاء متغير من النوع void. يجب أن يكون فقط مؤشر أو مؤشر لمؤشر أو أكثر من ذلك كما تشاء.
العظمة في استخدام مؤشرات void هي في أنك يمكن أن تنشيء مؤشر وتحجز له مثلا 100 خانة ولاتقتصر بوضع نوع واحد فقط من المتغيرات في المصفوفة !! يمكنك انشاء مصفوفة تضع في متغيرات مخلوطة من جميع الانواع .. يعني العنصر الاول يكون فيه قيمة int والثاني float والثالث int والرابع class قمت أنت بإنشاءه وهكذا .. لكن لايمكنك أبدا فعل ذلك بالمتغيرات العادية؛ كل نوع يخزن نوعه فقط ولامجال للتلاعب. ماعدا الفئات فلها حالة خاصة وهي تعدد الاشكال يمكن أن تقدم هندسة كهذه التي لدينا هنا في void.
كما ذكرت تستخدم مؤشرات void في تكوين مصفوفة لأنواع مخلوطة أو تمرير متغيرات لدوال تتوقع عدة أنواع من البيانات. بالنسبة لمصفوفة المؤشرات تكمن المشكلة عند محاولة استرجاع البيانات التي في المصفوفة، فلا نعلم نوع البيانات الموجود في كل خانة. يمكنك لحل هذه المشكلة أن تضع فهرسا لنوع المتغيرات في كل خانة بحيث تسترجع النوع فورا كما تريد. مثلا لو علمنا أن المتغير الاول في المصفوفة void التالية int والثاني char والثالث double لاسترجاعها سنقوم بالتالي:

```
void ** V;
V = new void* [10];
// Putting Data Here
cout << *( int* ) v[0];
cout << *( char*) v[1];
cout << *( double*) v[2];
```

مثال 7:

طبعا باستخدام الامر typeid(TYPE). name () ستحصل على النوع.

الخلاصة

سأوضح ذلك في الفصل المقبل وسأضع بنية للتعامل مع مصفوفات المتغيرات المختلفة، لكن بشرط كونها من الانواع الاساسية فقط.

قد توجد طريقة أخرى لعملها مع الأنواع الجديدة الفئات والبنى لكن ليست في حدود علمي حتى الان.

في الفصل التالي سنتناول مثلا لمصفوفة void وبنية للتعامل بشكل عملي مع مصفوفات المتغيرات المختلفة، ثم سنتناول موضوع ال مؤشرات الثابتة const ونحاول أن ألقي الضوء على مشكلة واجهتني تتعلق بالاداء عند استعمال المؤشرات.

في الفصل الاخير ذكرت أننا سنتناول موضوع مصفوفة VOID ذات الانواع المختلفة في هذه الفصل، لكن كتابة مصفوفة كتلك ليس فعالا دون استخدام الفئات Class لتغليف بعض المهام الجانبيه الخاصة بالمصفوفة. لذا سنأجل موضوعها قليلا حتى نتناول في جلسات أخرى في موضوع اخر سأطرحه في هذا المنتدى موضوع الفئات Classes ..

ثم نعود لنتابع هنا مرة أخرى ....

الفصل الخامس

تدرجنا حتى الان في مواضيع المؤشرات من الابتدائية وبعض الامور التي تستخدم كثيرا .. وحتى الامور المتوسطة التي لاتكاد تذكر في الكتب كموضوع Void* و موضوع مؤشر المؤشر int** ... ولأن هذه المواضيع شحيحة جدا في الكتب قررت أن أستعرضها هنا عسى أن تنفع أحد السائلين. سنتناول الان عدة أمور؛

أولا : سنكمل موضوع VOID ولكن ليس مصفوفات VOID وانما الدوال التي تستلم متغيرات من النوع VOID. لماذا توجد دوال هكذا ؟؟ مالداعي لتمرير بارمتر من النوع VOID الى الدالة ؟؟ وكيف يمكن للدالة التعامل معه ؟؟؟
لنفرض أننا نريد أن نكتب دالة فلنقل تقوم بضرب رقمين فقط ليس الا بحيث نمرر لها متغيرين وهما الرقمين المراد ضربهما ولكن يجب أن تعيد الدالة متغيرا من نفس نوعي المتغيرين وفي نفس الوقت أريد ان أجعل هذه الدالة تتعامل مع كل الانواع int , float double, وغيرها ... ولا أريد أن أكتب سوى دالة واحدة وتقوم في نفس الوقت بالعمل الصحيح تماما.
ماذا علي أن أفعل ؟؟؟ طبعا من الواضح أن من أراد كتابة كهذه دالة يجب أن يكتب 6 دوال مختلفة بنفس الاسم لكن بأنواع بارمترات مختلف لتغطية كل الانواع الاساسية ... بحيث يتم استدعاء الدالة المناسبة كل مرة .. لكن أنا لا أريد ان أتعب نفسي وأكتب 6 دوال ولاأريد تعقيد الشيفرة وزيادة عدد الاسطر مما لاطائل منه. وأريد أن أكتب فقط دالة واحده، فكل ماسأفعله أن الدالة ستكون كالتالي:

```
void* MULT (void* x, void* y, char* type)
{
  switch(type)
  {
   case 'int':    return &( *(int*)x) * (*(int*)y);       break;
   case 'float': return &( *(float*)x) * (*(float*)y);    break;
   case 'double': return &( *(double*)x) * (*(double*)y); break;
  }
}
```

مثال 1:

وهكذا نكون قد دعمنا 3 أنواع أساسية وتكفي. يمكن أضافة الباقي بوضع case جديدة.
المهم لاحظ هنا أن الدالة ستعيد القيمة المناسبة دائما حسب النوع الممرر لها.
هذا المثال ليس عمليا جدا لأن الكود كثير وكان يمكن بدل كل هذا التعقيد الاستعاضة عنه ب 3 دوال مختلفة كل دالة عبارة عن سطر واحد فقط.

فعلا هذا أسهل، لكن اذا مررنا الى الدالة مصفوفة من النوع ** VOID تحمل عدة أنواع مختلفة int , float , long ونريد أن نوجد مثلا أكبر رقم من الارقام الممررة بحيث نعيده بنوعه "يعني أكبر رقم int نعيده int واذا كان float كذلك وغيره... "، وفي نفس الوقت لانعلم عدد الارقام المدخلة !!! في هذه الحالة يستحيل حل هذه المشكلة الا بمصفوفات ** VOID:

```
void* MULT (void** x, char** type, int* size)
```

مثال 2:

ستكون الدالة كما في السابقة. لن أدخل في تفصيلها، لكن ان رغبتم في ذلك كتبتها لكم. عموما لاداعي بأن تقلق مطلقا بشأن هذه النجوم الكثيرة 99% من استعمالات السي++ وان لم تكن كلها يمكن أن تكتب بستخدام مؤشرات فقط "نجمة واحدة" وليس عدة نجوم. فقط رغبت في أن أوضح كل شيء هنا ولم أقصد فقط تبسيط المنهج. وهو الواضح من عنوان الدروس " نظرة تحليلية مفصلة" لكن سنبعد عن التعقيد في الفصول القادمة حتى نرجع لاحقا الى الفئات.

بداية ماذا يعني const ؟ لمن لايعرف فان const هو أمر أساسي في السي ومعناه ثابت .. ومن اسمه فهو يجعل المتغير الذي بعده ثابت طوال فترة البرنامج ولا يمكن أبدا تغيير قيمته بل يجب تعيين قيمة له وقت التمهيد فقط:

```
const int y = 0;
y = 7; // ERORR y Is Constant Var
```

مثال 3:

لايمكن أن تتغير قيمة y من الصفر أبدا طوال وقت تنفيذ البرنامج. الان مادخل const بالمؤشرات. كما خمنت يمكن أن نجعل المؤشر ثابت.

وموضوع const مع المؤشرات متعب قليلا فيمكن جعل المؤشر المطلوب ثابت أو جعل القيمة التي يؤشر اليها المؤشر ثابته:

```
int w = 33;
int q = 10;
const int * y = &q;
```

مثال 4:

هنا القيمة التي يؤشر اليها المؤشر y ثابته يعني y* ثابته وليس y هي الثابتة:

```
y = 55; // ERROR *y is const *
y = &w; // OK You Can change Th Value
```

مثال 5:

لايمكن تغيرها بعد تمهيدها كما في المثال 3. أما y نفسها فيمكن تغييرها, أما المثال التالي:

```
int w = 33;
int q = 10;
int * const y = &q;
```

مثال 6:

هنا العكس يمكن تغيير القيمة y* ولايمكن تغيير العنوان y. لاحظ لمكان توضع const في المثالين 6 و 4:

```
y = 55; // OK You Can change The Value*
y = &w; //ERROR y is Const
```

مثال 7:

الامر واضح اذا كانت const في البداية كما في المثال 4 فان القيمة التي يحملها المتغير ثابتة. أما اذا كانت بعد نوع المتغير كما في المثال 6 فهذا يعني أن عنوان المتغير هو الثابت. أخيرا يمكن وضع const في كلا الموضعين لكي يصبح كل من المؤشر وقيمة المؤشر ثابتة:

```
int w = 33;
int q = 10;
const int * const y = &q;

y = 55; //ERROR *y is Const*
y = &w; //ERROR y is Const
```

مثال 8:

أخيرا بالنسبة لما يتعلق بالاداء فان سرعة عمل البرنامج وهو يعمل بالمؤشرات أقل من سرعته وهو يعمل بالمصفوفات العادية بمدة زمنية لاتتجاوز 1 من كل 10 مليار لافرق بينهما مطلقا، وهذا ماتأكدت منه وقد كنت أشكك في ذلك حتى جربت بنفسي. عموما المشكلة الوحيدة في الاداء يمكن أن تحدث بسبب المبرمج، حيث لو كتبت كود برنامجك بحيث وضعت الكثير من عمليات حجز للمتغيرات:

```
w = new int[100];
```

الامر السابق يأخذ وقتا طويلا. فيجب محاولة الابتعاد عن تخصيص الكثير من الذاكرة كلما أمكن ذلك، فلو كان الامر السابق داخل حلقة تعمل 40000000 مرة سيأخذ البرنامج وقتا طويلا جدا مما يسبب في فشل أداء البرنامج بشكل كبير. وكل ماحاولت الابتعاد عن حجز كميات الذاكرة كان الاداء أفضل.

عموما حاول بقدر الامكان أن تحجز لمتغيراتك كلها مرة واحدة، أو عدة مرات قليلة اذا استلزم الامر ذلك. لكن اياك أن تضع عمليات حجز داخل حلقات Loop كبيرة لكميات كبيرة. واذا اضطررت لذلك فيجب عليك النظر في هندسة البرنامج ككل من البداية.

وهذا هو الخطأ القاتل الذي وقعت فيه عندما كتبت كود الشطرنج .. كان هناك العديد والعديد من استدعاءات الحجز بشكل ديناميكي مما أثر على الاداء بشكل كبير جدا. وقد عرفت سبب المشكلة لكن بعد فوات الاوان.

عموما هذا مثال مرفق (الفهرس) يعرض الزمن بالملي ثواني لعملية حجز داخل حلقة وعملية حجز لعدد مرات أقل داخل نفس الحلقة ستلاحظ الفرق الكبير بين السرعتين، ولكن اذا حجزت مرات قليلة داخل حلقة و لأحجام ضخمة سيكون الاداء هو الاسوأ على الاطلاق.

الخلاصة

احجز لعدد مرات متوسط لكميات متوسطة حيث المتوسط يساوي 10000.

وبالمناسبة هذه المعلومات قيمة جدا ولم أجد لها مثيلا في الكثير من الكتب، فقط يجب عليك القيام ببعض التجارب لتعرف ماهو أفضل أو محاولة قراءة تعليمات الاسمبلي الخاصة بعمليات الحجز الديناميكي للخروج بأفضل النتائج في سرعة تنفيذ البرنامج.

يوجد أمر واحد لم أذكره حتى الان وهو أحد أوجه الخطورة أيضا ومن الاسباب التي دعت مايكروسوفة لتبني ملمم النفايات ليدير عمليات تنظيف المؤشرات المحجوزة بشكل آلي، حيث يجب دائما في المؤشرات عندما نحجز بالامر new أن نضيف الامر delete لحذف البيانات التي قمنا بحجزها .. والا سيتعطل البرنامج كله اذا كانت كمية البيانات المحجوزة كبير. وهذا هو عمل ملمم النفايات الذي يهتم باستدعاء الامر delete بشكل الي عند الحاجة، ولكن هذ الملمم غير متاح في لغة السي/سي++ وانما يوجد في ال .net C++ , JAVA , c# ...

الفهرس

```
#include <iostream.h>
#include <typeinfo.h>
#include <time.h>

void main()
{
      clock_t start,finish;

      int *y = new int[10];
      int W;
      int e[10],w,q=0;
      for(w=0;w<1000000000;w++)
            W=0;
      cout<<"HERE WE GO :"<<endl;
      /*
      for(int I=0;I<2;I++)
      {
      cout <<" HERE WE GO ...:"<<endl<<endl;
    start = clock();
      for( w=0 ;w <1000000000;w++)
      {
            if(w%10==0)q=0;
            e[q] = 100;
            q++;
      }

      finish = clock();
      cout <<"SECOND NORMAL IS :"<<(double)(finish-
start)/CLOCKS_PER_SEC<<endl;

      start = clock();
      for(w=0 ;w <1000000000;w++)
      {
            if(w%10==0)q=0;
            *(y+q) = 100;
            q++;
      }

      finish = clock();
      cout <<"FIRST POINTER TIME IS : " <<(double)(finish-
start)/CLOCKS_PER_SEC<<endl;

      start = clock();
      for(w=0 ;w <1000000000;w++)
      {
            if(w%10==0)q=0;
            y[q] = 100;
            q++;
      }

    finish = clock();
      cout <<"FIRST POINTER TIME IS : " <<(double)(finish-
start)/CLOCKS_PER_SEC<<endl;
      }
  */
      start = clock();
      for(w=0 ;w <20000000;w++)
      {
            if(w%10==0){q=0;y= new int[10];}
```

```
            //if(w%10==0){q=0;y= new int[10];}
            y[q] = 100;
            q++;
      }

      delete []y;
    finish = clock();
      cout <<"BAD VERSION TIME IS : " <<(double)(finish-
start)/CLOCKS_PER_SEC<<endl;
    ////////////////////////////////////////////////
      start = clock();

      for(w=0 ;w <20000000;w++)
      {
            if(w%10000==0){q=0;y= new int[10000];}
            y[q] = 100;
            q++;
      }

      //delete []y;
    finish = clock();
      cout <<"MORE BETER TIME IS " <<(double)(finish-
start)/CLOCKS_PER_SEC<<endl;
    ////////////////////////////////////////////////////
    start = clock();

      for(w=0 ;w <20000000;w++)
      {
            if(w%10000000==0){q=0;y= new int[10000000];}
            y[q] = 100;
            q++;
      }

      delete []y;
    finish = clock();
      cout <<"THE WORST TIME IS:" <<(double)(finish-
start)/CLOCKS_PER_SEC<<endl;
    ////////////////////////////////////////////////////
      cin.ignore();
}
```