مقدمة

السلام عليكم,,,

ستكون هذه سلسلة دروس للتعرف بشكل أفضل باذن الله على البرمجة كائنية المنحى ,, وهي بالأساس الفئات,,,
وهو موضوع جد طويل ويحتاج لبعض المجهود وهو سهل لمن سهله الله عليه,,,
وأقول طويل لأن الفهم الصحيح لمفهوم الفئات يحتاج لبعض الوقت ,,
فعلى الله توكلنا,,,,

الدرس الاول

OOP(Object Oriented Programming) البرمجة كائنية المنحى " البرمجة الموجهه بالكائنات " هي ذلك المفهوم الذي بزغ الى عالم البرمجة ليغير طريقة البرمجة الاجرائية القديمة ذات الدوال والمناهج الضيقة الأفق الى سعة ورحابة الكائنات,,,

منذ بزوغ فجر تاريخ البرمجة بدأ المبرمجون بكتابة برامج باستخدام لغة الالة ,, فوجدو بعد فترة أن هذه اللغة متعبة لحد يجعل تطور البشرية في هذا المجال أمرا صعبا للغاية ,, فقررو تطوير البرمجة لاستخدام دلالات تعبر بلغة أقرب للغة البشر عن برمجة الصفر والواحد,, فاخترعو لغة الاسمبلي " التي ليست الا اختصارا لتعليمات ست عشرية هي في الأصل صفر وواحد " ,, فتطورت البرمجة بشكل كبير وسريع لكن مع ازدياد الحاجة البشرية للسرعة قررو أن هذه اللغة تأخذ وقتا طويلا للغاية ,, فقررو مرة أخرى تبسيطها اكثر فبدأو باختراع لغات البرمجة عالية المستوى أمثال الكوبول والباسكال والبيسك وأمثالها الكثير ,, وكانت كلها مسيطرة في وقتها حتى بزغ فجر لغة السي,,
مرة أخيره " حتى الان " قرر المبرمجون أنهم بحاجة الى التبسيط زيادة فقررو أن يحاولو محاكاة الواقع فلا أسهل من التعامل بشكل طبيعي مع الاشياء ,,وهنا بزغ فجر لغات البرمجة كائنية المنحى,,

لغة السي لغة اجرائية لاتحتوي على مفهوم الكائنات فالسي أفضل مالديها في التعامل البنيوي هو ال Struct ,,,
أما في لغة السي ++ فتوجد الفئات Classes التي هي أساس التعامل الكائني المنحى,,,
قبل أن نستمر انظر الى مكتبة ال ,, Win32 Api مكتبة ضخمة جدا تتحكم بالوندوز كله لكنها مكتوبة بلغة السي ,, ليست كائنية المنحى حيث تتكون من عشرات الالاف من الدوال التي يستحيل تذكرها ,,
وفي نفس الوقت انظر الى مكتبى .NET FramWork. مكتبة كائنية المنحى بشكل كامل تغلف دالات ال Win32 Apiلتسهل الوصول اليها ,, وهي رائعة جدا سيحبها كل من يتعامل معها نظرا لسهولة وسرعة تعلم استخدامها ,,," وهي المصة الخاصة بلغات برمجة الدوت نت الجديدة,,, "

الان عندما أقول كائن هذا المصطلح غامض بعض الشيء ,,, لكن يمكنك تشبيهه فورا بمفهوم الكائنات التي في العالم الحقيقي ,,, يمكن أن يكون الكائن انسانا حيوان جماد مثل المكتب المصعد الكهربائي وحتى كرة القدم,,,
الان مالعلاقة بين كائنات العالم الحقيقي وكائنات البرمجة ؟؟
عندما فكر مخترعو البرمجه الكائنية بهذا المفهوم الجديد كل ماكان لديهم في ذلك الوقت هو تسهيل البرمجة بأكبر فرصة لتصبح مشابهة للتصرفات على الواقع تماما,,,

فكر المخترعون على طريقة لابعاد المبرمج كليا عن طريقة عمل كائن ما في البرمجة ,, بحيث يركز عمله فقط على كيفية استعماله!!
لتركيز هذا المفهوم في الواقع خذ عندك مثالا : لعبة رجل الي,, يلعب بها طفل ويحركها بيديه ويضغط فيها أزرارا لتصدر بعض الاصوات والحركات وتنفذ بطاريتها فتتوقف عن اللعب ويرميها في الأرض فتتحطم!!

الان هذا الطفل لن يعرف مطلقا كيف يعمل هذا الرجل الالي كيف يتحرك اذا ضغطنا هذا الزر كيف يصدر صوتا اذا ضغطنا ذلك الزر!!

هذا مشابه تماما لما يريدنا مخترعو ال OOP الوصول اليه ,,, أن نتحكم بالكائنات بكل سهولة دون الدخول في تفاصيل طريقة عملها,,,
ومن هنا بزغ فجر مفهومين جديدين للبرمجة " صانعو الفئات " "ومستخدمو الفئات,,,, "
صناع الفئات هم كما في لعبة الرجل الالي الشركة المصنعة لهذه اللعبة ,,
والمستخدمون هم الاطفال الذين يلعبون بها ولايعلمون شيئا عن طريقة عملها الداخلية ,,, فقط يصدر الصانعون Manual لطريقة الاستخدام لكي يعرف الطفل كيف يستمتع بها وهو تماما مايحدث في كائنات ال,,, OOP

الان هل يمكن فعلا أن تكون البرمجة بهذه السهولة ,؟ أقول نعم اذا ركز كل على عمله,,,
مصنعو الفئات سيكون بالطبع عليهم العبء الأكبر ,, المستخدمون قد يكون عليهم عبء وقد يكونون في قمة حالات الاستمتاع بهذا الكائن,,
حالات الاستمتاع في الواقع كما لدينا الطفل الذي يلعب بالكائن الالي الكامل وهي اخر مراحل استخدام الكائن,, لأن الطفل لن يستخدم الكائن ليطوره لكائن اخر "الا اذا كنا في عالم ال Matrix ونحن لانعلم !!!",, فقط سيكتفي باللعب به,,,

أما لوكنا في مثال اخر لو كان الكائن الحالي لدينا هو عبارة عن محرك سيكون هناك بعض العبء على مستخدم الكائن الذي سيقوم بتركيبه مع عدة كائنات أخرى ليكون في النهاية كائنا جديدا ... هنا نحن لم ننتهي من سلسلة التطوير لهذا الكائن بعد ,, فيمكن اعتبار المستخدمين مطورين,,,

بهذا المفهوم مطورو المحركات سيبيعونها لمصنعيين اخريين وبهذا التكامل نبني واقعنا في الحياة,,,
نفس المفهوم تماما موجود في عالم البرمجة OOP لكن من يستطيع الوصول لهذه المراحل من التطوير !!
من قال أنه لايوجد ,, لو دخلت ورأيت برمجة الألعاب ستجد العجب العجاب,,,
ولو اضطلعت على نماذج محاكاة الواقع الافتراضي فهي القمة في استخدام الكائنات ,, لأنها تبنى اساسا على محاولة محاكاة كائن في الطبيعة بشكل حقيقي تماما ليعمل على الكمبيوتر بنفس طريقة عمله في الطبيعة,,
مثلا متتباعات الاشعة ,, ومحاكاة حركة الرياح والاعاصير ,, محاكيات أحوال الطقس ,, محاكيات التفاعلات الكيميائية وغيرها,,,

مثلا في محاكيات التفاعلات الكيميائية سيكون المطورون بداية كائن هو عبارة عن ذرة بالكتروناتها ونواتها وبوزوتروناتها وبروتوناتها وكل محتوياتها,,,
هذا الكائن سيدمج في كائن أكبر منه وهو الجزيئ سيتكون من عدة كائنات ذرة ,, ثم نتدرج حتى نصل الى المادة الكيميائية ,, ويكون مبرمجو الكائنات السفلية قد اضطلعو على كيفية تفاعل الجزيئات مع بعضها بشكل تام ثم يبدؤون بكتابة " الدوال (أقول الدوال هنا وأنا لاأمزح) " التي ستقوم بعملية الاتحام الدمج بين الجزيئات ويملؤنها بكل تفاصيل التفاعلات ,, في النهاية فقط ما على مستخدم الكائن النهائي وهو " كائن بيانات المحلول " [ان ندخل له بيانات المحلول الاول والثاني ونطلب منه أن يفاعل بينهما وننتظر نتيجة التفاعل!!!!
هذه الاشياء بالطبع تحتاج لكمبيوترات عملاقة سريعة لتنفذ كل هذا الكم من التعليمات,,
لكن يمكن التدرج وصولا لمستويات مبرمجي الالعاب حيث يقومون ببناء ألعابهم على أساس الكائنات,,,
مثلا خذ عندك لعبة بلياردو وهو مثال أوضح نوعا ما,,,
ماعلى مطوري اللعبة الا استخدام كائنات كرة بلياردو " لأنها الجزء الاصعب " كائن البلياردو هذا سيتعامل كما في الحياة الواقعية تماما ,,سيكون الكائن عبارة عن جسم كروي له كتلة بافتراض ان الجاذبيه الارضية 9.8 سنعطيه أيضا مكان لتخزين معلومات طاقته الحركية وطاقته الكامنة ,,, فكل ماعلينا هو كتابة دالة لتقوم بعملية التصادم بحيث أن كل كرة عندما تصطدم بكرة أخرى ستستمد طاقة حركية وطاقة كامنه داخلها بهذا المبدأ يمكن أن نحرك كراتنا وننسى تماما كيفية تصادمها وانعكاسها!!!

أردت ان أبين هذه الأمور لأنها الاشياء التي أتعبتني في فهم الكائنات بالشكل الصحيح ,,, لم لأجد كتابا يتحدث عنها بالشكل المفروض ,, كل الكتب تعطي أمثلة سطحية سريعة مباشرة لاتعبر عن الاستخدام الامثل للكائنات,,,
فمثلا لو قلت لك مثال مصعد كهربائي هو عبارة عن كائن ,, ستقول لي يمكن أكتب برنامجا كهذا دون الدخول في تفاصيل الكائنات باستخدام لغة اجرائية بسيطة !! فيصبح المبرمج المبتديء الذي سكون ضيق الافق في البداية مشوشا لايعرف الاستخدام الامثل لهذه التقنية,,,
أخيرا قبل أن أنتهي من هذه المقدمة الفلسفية أقول أن المستقبل سيحمل فقط لغات كائنية المنحى من لم يرد الدخول في ذلك سيسقط وما عليه الا بانتظار قدره,,,
فأمر الكائنات ليس معقدا بل مفهومه مختلف فقط,,,

الدرس الثاني

في المرة السابقة تحدثنا عن الكائنات وعرفنا سبب التسمية " أنها تحاول محاكاة الكائنات في الواقع " ,,, وتعرفنا على أمثلة كان من الصعوبة تنفيذها دون استخدام الكائنات وتشبيهها بطريقة عملها في الحياة الواقعية,,,
في هذه الجلسة سنلقي مزيدا من الضوء على ميزات ال OOP في الامثلة الفعلية ,,ولن ندخل في أمور الكود الا الجلسة المقبلة ,, فال OOP مفاهيمية أكثر من كونها كتابة كود,,

لنفرض أن لدينا مبرمج يريد انشاء لعبة تصويب ثلاثية الابعاد ,, مثل Quake3 مثلا,,
بها شخصيات واناس يتحركون ويتصرفون بشكل ذكي وكأن لهم عقول يفكرون بها,,,
الان انظر الى حال أحد المبرمجين القابعين أمام أجهزتهم كل الوقت وهو يكتب كود تكامل اللعبة مع بعضها ,,, لولا الكائنات في البرمجة لظل هذا المبرمج 6 سنوات وهو يحاول أن يكامل بين الاف الاجزاء في مقابل أن يجلس سنتين فقط وهو يستعمل كود للكائنات,,
"الالعاب الكبيرة تستغرق فترة متوسطها سنتين,, "

الان لنفرض مثلا أن هذا المبرمج لايستخدم كود كائني ,, سيضطر في كل frame أن يتكفل بتحريك كل شخصية في اللعبة ويقلق بشأن تصرفها هل هو سليم أم لا ,, هل تعدى الكائن الفلاني حدود المشهد أم لا ,, هل اصطدم شخصين مع بعضهما في المشهد ام لا ,, سيجن جنونه وهو يحاول ملاحقة هذه الاحتمالات ,, وكل تعديل طفيف سيأخذ منه وقتا كبيرا ,, وكل تعديل كبير يمكن أن يؤدي بالمشروع الى الهاوية,,,
هذا بالنسبة لحال مبرمج واحد ,, فما بالك اذا تشارك فريق لتطوير اللعبة ,, يجب عليهم أولا أن يتواجدو في مكان واحد ,,, واحتمال تضارب الاكواد بينهم كبير لدرجة تجعل من المستحيل تنفيذ المشروع,,,

في المقابل افرض أن مبرمجنا يستخدم كود كائني المنحى ,,, سيتم تقسيم أعضاء المشروع الى فرق ,, كل فريق له مهمه واضحة محددة كالشمس ,,, مثلا الفريق الذي سيهتم بكتابة كود الشخصيات ,, سيقوم بكتابة فئة تعرف الشخصية ويضع كل الاحتمالات الممكنة لهذه الشخصية ,, الحركة التخاطب الاصوات ,, حدود المشهد ,, التصادم بين الشخصيات ,, ماذا لو اصطدمت الشخصية بأخرى ,, قد ترتد وتصدر صوتا مثلا ,, أو غيرها من الاستجابات,,
حتى الان هذا الكائن بدأ يتجسد ,, بالطبع بعد مكاملة فريق رسم الشخصية مع المبرمج ,, يبقى أمر مهم ,, بث الحياة في هذه الشخصية ,, !! كيف يمكن بث الحياة فيها,,
بعد تعريف الفئة وتعريف كل المتغيرات الضرورية فيها والدوال التي ستقوم بالامور المهمة ,, تبقى الدالة الام التي هي في الواقع كأنها العقل البشري الذي يحدد مايجب فعله حسب التغيرات الخارجية ,, " لايمكن بالطبع جعلها تتصرف كالعقل البشري " ,,مثلا لنفرض أن الشخصية ستكون حارس لبوابة وكل من يقترب من هذه البوابة سيتم التصدي له,,
سنكتب دالة اسمها UpDate يتم استدعاءها كل Frame مثلا ,, بحيث يتم مسح دائرة نصف قطرها 8 أمتار من الشخصية واذا وجدت شخصية أخرى في هذا المدى تستدعى دالة أخرى لتحفيز القتال !!!
دالة تحفيز القتال ستستدعي دالة لتغير وضعية الشخصية الرسومية ثم تستدعي دالة الهجوم ,,,
وهكذا بسلسلة كهذه من الاحتمالات الاساسية يكون لدينا في النهاية مقاتل صنديد يتصرف بتلقائية وبالشكل المطلوب,,,
الان فلنعد لمبرمجنا الذي كان سيقضي 6 سنوات وكأنه يقضيها في السجن ,, ونعطيه فئة الشخصية ,, وأنواع الشخصيات الاخرى ,, سيكون سعيدا جدا لأنه لن يفعل شيئا في كل Frame الا أنه سيستدعي الدالة UpDate كل مرة وينتهي الامر !!! ,, لأن الدالة هي التي ستجعل الكائن يتصرف ,, هكذا يمكن لكل عضو في الفريق أن يركز فقط على عمله وبشكل مدهش ,, وأن يعملو مع بعضهم بشكل فعال حتى لو كان بينهم الاف الاميال!!

الان هذه الفئة فئة الشخصية حجمها قد يكون كبير ,, لكن مبرمج اللعبة لن يقلق بشأنها فليس له أي علاقة بحجمها فقط كل ماعليه هو أن يضعها ويقرأ طريقة استخدامها وينسى كل شيء ,, ويعتمد على أن مبرمج الفئة قد أتقن عمله فعلا,,
هنا تقريبا يكمن العبء الأكبر على مبرمج الفئة حيث يجب أن يكون حذرا ويتأكد بشكل كبير من عمل الفئة بالشكل الصحيح,,

حتى الان في هذه الجلسة والمقدمة السابقة كنا نتحدث فقط عن الميزة الاولى للبرمجة كائنية المنحى ,, بقي الميزة الثانية والثالثة,,,
الميزة الاولى كما قلت هي الكائنات وتصرفها كالواقع تماما,,
الميزة الثانية الوراثة ,, والثالثة تعدد الأشكال,,,
بالنسبة للوراثة فهي ميزة رائعة جدا ,, وتجعل التطوير يتم بسرعات خرافية ,,وتسمى أيضا اعادة

الاستخدام,,,
في مامعناها المختصر وراثة كل صفات وخصائص كائن اخر ,,,
وبالطبع يمكن أن تزيد على الكائن الاب في الكائن الابن كما تشاء بحيث يكون الاب كما هو دون تغيير,,

فالنعود لمثال الشخصية ,, لنقول أن الشركة المطورة للعبة طورت سابقا لعبة بها شخصيات أيضا ,,, لكن كتاب كائن الشخصية لم يطوروها بالشكل المطلوب ,,فقط اكتفو بجعلها تتصرف التصرفات الطبيعية المشتركة بين الشخصيات الطبيعة الحقيقية ,, مثلا التنفس ,, الحركة الطبيعة ,, وحدود القطع في المشهد والتصادم,,

ثم "اتفركش المشروع " وتم اغلاقه واعلان فشله ,,, وبعد 5 سنوات قام مشروع جديد وهو مشروع مبرمجنا الصنديد ,,, وتم تغيير اصطاف المبرمجين في هذه الفترة ,, لكن الوثائق القديمة والادوات والفئات مازالت موجودة ,, فكل ماسيفعله مدير المشروع هو أن يهرع بجلب الفئة التي عرفت سابق ويضيفها للمشروع بحال كائن أب ,, ويشتق فريق تطوير شخصية المحارب منها ليكتسب كل الصفات الاساسية في لمح البصر ,, ويصبو مجهودهم على الاضافات فقط ,, كالقتال وغيره!!!
يمكن لفرق الشخصيات انتاج مئات الشخصات بسرعة خرافية لأنهم سيركزون على الاضافات فقط ,, مثلا شخصية طباخ وشخصية لاعب كرة وشخصية وحش "اذا كان يتنفس أيضا !! " كل هذه الشخصيات ستتطور بسرعة كبيرة,,,
بالطبع مبرمجنا سيكون سعيدا للغاية وهو يحتسي قدحا من القهوة وهو يراقب هذه الفرق وهي تعمل ,,فهو بلا عمل لأنه أنجز كل عمله!!!
فاعادة الاستخدام ميزة رائعة جدا لكن تتطلب بعض الحذر واتساع الافق وبعد النظر,,,
ومحاولة جعل التطور يكون في أكبر عدد من المستويات لكل مستوى فئة ترث من القبلها ليسهل في أي لحظة الوصول للفئة الاقرب للحاجة,,,

حتى الان مبرمجنا سعيد للغاية فقد قضى 6 أشهر فقط لمكاملة المشروع ككل ,, لكن هل توجد طريقة لاختصار المزيد من الوقت ,,نعم وهي الميزة الثالثة " تعدد الأشكال,, "
تعدد الاشكال كما هو واضح من اسمه شيء واحد لكن له عدة أشكال ,, كيف يكون ذلك ؟
يتم ذلك في السي++ باستخدام الفئات والوراثة وكتابة الكلمة الاساسية Virtual كل هذا بشرط استخدام المؤشرات ,,, بهذه الخلطة السرية يمكن لمثال لعبة مبرمجنا أن تتطور الى التالي,,,

لنفرض أن لدينا في المثال الف شخصية مختلفة ,, ولنفرض مثلا أن الشخصية الرأيسية ستدخل الى سوق فيه شخصيات كثيرة من أنواع مختلفة,,
لنفرض عدد الانواع الكلي مثلا وليكن 100 ,,, سيتضجر مبرمجنا من هذه الكمية الكبيرة من الانواع ليستدعي لها الدالة ,, UpDate والمشكلة الأكبر لو أنه يريد استدعاء الدالة UpDate بترتيب معين لكل الكائنات هذا الترتيب يتغير بشكل ديناميكي ,, حسب أحوال وظروف اللعبة وبعض الامور التي تتعلق بالأداء ,, الان المبرمج في مشكلة حقيقية ليسيطر على ألف شخصية ذات 100 نوع مختلف" ,, لكنه مازال أفضل حالا بكثير من مبرمج اللغة الاجرائية,,,"
لحل هذه المشكلة نستخدم تعدد الأشكال ,, لن أشرح طريقتها الان بالطبع فقط سأشرح الفكرة,,
كل ماسيحدث هو ان المبرمج سيأتي بأول كائن أب لكل الشخصيات " وهي الشخصية التي صدرت من المشروع القديم الفاشل قبل 5 سنوات" ويمرر له فقط اسم نوع الشخصية التي يريد تحديثها ,,
ويستدعي الدالة UpDate بواسطة الكائن الأب ولكن لن تستدعى الدالة التي لصالح الكائن الأب !!!
الغريب في الامر أن الدالة UpDate ستنفذ لصالح الكائن الذي مررنا اسمه للكائن الاب ,, وليس لصالح كائن الاب بعينه ,,
سنعلم أنه من المفترض أن أي دالة تستدعى باسم كائن معين ستكون خاصة به هو فقط,,
بهذه الطريقة يمكن أن يمرر المبرمج أنواع المتغيرات بالطريقة التي يريد ولايقلق بشأن اسم المتغير ولاجمل الاختبار الكثيرة التي ستحلل مايجب استدعاءه وما لايجب,,,

مرة أخرى أقول لايمكن فهم هذه الاساسيات الثلاثة الا بالأمثلة الدسمة ,, المفهوم العام وقد اتضح حاليا للأمثلة الحقيقية,,
بقي أن نفهم تمثيل ذلك في الكود ,,, ويجب دائما أن نتخيل الكائن من بدأ كتابة كوده وحتى ينتهي بأنه كائن حقيقي ,, فقط ينقصه بعض النضج,,,

أخيرا قبل أن أنتهي أريد أن أربط بين مثال هذه الجلسة " كائن الشخصية " ومثال المقدمة الفائتة " كائن الذرة والجزيئ,,, "
في مثال الجلسة السابقة كانت علاقة الكائنين ببعضهما " كائن الذرة وكائن الجزيء " هي أن كائن الجزيء له كائن ذرة,,,
أما في المثال الحالي فان كائن المحارب نوع من الكائن الأب,,

وهما نوعي اعادة الاستخدام في الفئات,,,
هكذا تنتهي هذه المفاهيم الفلسفية ,, لندخل في الجلسات التالية لأمور الشيفرة وكتابة الكود ونتعرف على قوة الفئات في لغة السي++ ,, وسندعم ذلك بأمثلة قوية باذن الله ,, ولن " نشطح " بكم في صعوبة الامثلة حتى يبقى الكل قريب من الموضوع,,

أخرا اعذورو لي كثرة الحديث فقط أردت أن أوضح الميزات بشيء من الأمثلة التي تجعل الميزات ميزات فعلا ,, وتختصر الوقت بشكل جدي,,
بهذه الامثلة الواضحة لن تكون هناك مشكلة في فهم الامثلة الاسهل منها أبدا باذن الله,,

الدرس الثالث

الان كل مايتعلق بالبرمجة كائنية المنحى في سي++ يتكون بشكل كامل من 6 كلمات أساسية ,,,
وهي:
class public private protected virtual friend ,,, فقط!!

هذه كل البرمجة كائنية المنحى,, وبعد أن نعرف استعمالات هذه الكلمات الاساسية نكون قد انتهينا من السي!!! ++

لكن الامر ليس في الكلمات الاساسية فقط ,, بل في كيفية استخدام هذه الكلمات,,,
ومواضع استخدام هذه الكلمات ,, كما سنرى لاحقا يمكن كتابة عدة كلمات في أماكن بوضعيات مختلفة ستعطي معاني مختلفة,,,

أما الان فالنبدأ:
خذ المثال التالي بداية:

مثال:1

CODE

```
class Ball
{

public:
int m_M;
float m_speed;
int m_pos;
int m_direction;

void GetSpeed();
};
```

هذا المثال يستعرض كلمتين اساسيتين من المستخدمة في برمجة الكائنات,,, class public,,
لاحظ للكلمة الاساسية class كما في تعريف البنية struct تماما,,
الان الكلمة Ball ستصبح أحد المتغيرات الاساسية المعرفة لدى المصرف ,, مثلها ومثلا ال int float والاخرين !! فكل مايحدث في الفئات أنك تقوم بتعريف أنواع جديدة من أصل أنواع أساسية معرفة مسبقا ,,
الان من هذا الوصف ,, لاحظ داخل الفئة Ball كل المتغيرات مصرحة في القسم,, public ويكلها من الانواع الاساسية,,,
ويمكن بالطبع انشاء متغيرات من أنواع قام المستخدم بتعريفها أي من فئات أخرى ,, كما سنرى لاحقا ,,,
أخيرا لاحظ أن الفئات تتكون من متغيرت ودوال !! أي أن كل ماتحدثنا عنه من أعمال بطولية لكائنات

وطريقة تصرفها سابقا ,, ليست سوى دوال ومتغيرات !! نعم هو كذلك,,
لكن من كان يتوقع من الصفر والواحد أن تبني مستعمرة كالتي بين أيدينا الان!!!
وكما قيل في المثال الدارجي " يعمل من الحبة قبة " هو مايحدث في الكائنات ,, ستبني مستعمرة بهذه المتغيرات والداول,,,

الان لاحظ للكلمتين الاساسيتين ,, private public هتان الكلمتان يتم استخدمهما فقط في الفئات classesوالبنى,, struct
ال private public اذا كتبا داخل جسم الفئة فلهما معنى وضح ,, هو صلاحيات الوصول,,
يتم تحديد صلاحيات لوصول الكائنات التي من النوع Ball لاحقا بواسطة الكلمتين pirvate public وأيضا potectedالتي سنتطرق لها لاحقا,,
الى ماذا يريد ان يصل الكائن ؟؟
الكائن يريد الوصول للمتغيرات والدوال المعرفة داخل نوع فئته ,, يعني في هذه الحال يريد الوصول لل m_speed m_Mوغيرها,,,
كلمة وصول قد تكون غامضة قليلا ,, لذا فلنوضح معنى كلمة وصول أولا,,,
خذ المثال بعد أن عرفنا الفئة في الاعلى,,,

مثال 2:

CODE

```
void main()
{
Ball myball;

myball.m_speed = 10;
myball.m_pos = 15;

}
```

الان العملية myball.m_speed تسمى الوصول ,,, الوصول الى أحد الاعضاء المعرفة في الفئة,,,
لأنه الان اذا كان لدينا فئة أخرة معرفة تدعى Car كالتالي:

مثال 3:

CODE

```
class Car
{
public:
```

```
int m_speeed;
int m_color;
};
```

وأنشئنا كان منها يدعى mycar سيكون الوصول لأعضاءها البيانية والدالية بنفس الطريقة بالعامل . "
النقطة " ,,, فعندما أقول mycar.m_speed فأنا أقصد m_speed الخاصة بالفئة Car وليس الفئة ,, Ball و
أكثر من ذلك ,, لكل كائن جديد يتم تعريفه من نوع فئة معين متغيرات مختلفون "في القيمة " عن
الكائنات الاخرى,,,
يعني لو أنشئنا كائنات تدعى myball1 و myball2 اضافة ل myball في المثال 2:
فان لكل الثلاثة متغيرات ستكون هناك قيم مختلفة ل m_speed والمتغيرات الاخرى,,,
كما قلت المتغيرات هي التي تصف الكائن فكل كائن يجب أن تكون له قيم مختلفه لهذه المتغيرات,,,,
أما الدوال التي في الفئة فهي خاصة بالفئة يأكملها لذا فهي تنطبق على كل كائنات الفئة بلاتغيير,,,

الان لماذا وضعت محددات الوصول ؟؟ لم أشرح بعد ماهية ال ,, private public لكن سنأخذ هذا المثال
الذي ليس له علاقة بالفئات لنفهم محددات الوصول بشكل أفضل,,,
مثال 4:

CODE

```
void Get()
{
int cat = 10;

}


void main()
{

cat = 7;

}
```

المثال أعلاه لن يعمل ,,, السبب في ذلك أن المتغير cat تم تعريفه في مجال لايسمح بالوصول اليه من
مجال اخر,,
أي تم تصريح المتغير cat في دالة أخرى لايمكن الوصول اليها من ال,,, main
فنفس ماسيحدث سيحدث في لفئات ,, لكن بمفهوم مختلف قليلا ,, خذ المثال الذي سيوضح الامر
بشكل أفضل,,,
مثال 5:

CODE

```
class Ball
{

private:
int m_M;
float m_speed;

public:
int m_pos;
int m_direction;

void GetSpeed();
};


void main()
{

Ball bn;

bn.m_pos =10;    //OK Scope is Public

bn.m_speed =15;  //ERROR Scope is Private Access Must be From Member Functions of class Ball Only
                 // Just Like GetSpeed() Function

}
```

الان في هذا المثال لن يتم الوصول ل m_speed الخاصة بالمتغير bn لكن يمكن الوصول ل ,,, m_pos السبب في ذلك يعود لنوع الوصول حيث هو الان لأول متغيرين m_speed m_M هو private ويعني أنه لايمكن الوصول اليهما الا من أحد الدوال الموجودة داخل الفئة ولايمكن الوصول اليهما مطلقا من أي مكان اخر من البرنامج ,,," فقط يمكن الوصول اليهما من داخل تعريف الدالة GetSpeed() التي داخل الفئة ,, لأنها من نفس فئتها,,"

أما المتغيرين الاخرين m_pos m_direction فيمكن الوصول اليهما من أي مكان في البرنامج سواء من دالة داخل الفئة أو من أي دالة عامة في أي جزء من البرنامج,,,
والسبب في ذلك وجود الكلمة الاساسية public قبلهم ,, يعني أي متغيرات أو دوال قادمة بعد تصريح الكلمة الاساسية ستكون من نوع الكلمة الاساسية لو كانت,,, private or public or protected

,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

الصيغة الكتابية للفئة واضحة ,, اسم الفئة بعد الكلمة الاساسية class ثم فتح قوس بداية تعريف الفئة,,
ثم محدد الوصول وبعده نقطتين:
ثم كتابة المتغيرات او الدوال التي تريد ,, لمحدد الوصول,,
وكتابة محدد وصول اخر ثم كتابة الدوال أيضا والمتغيرات التي تريد لمحدد الوصول هذا,,
ويمكن بالطبع أن تكتفي بمحدد وصول واحد ويمكن أن تجمع كل المحددات الثلاثة,,,
private protected public في فئة واحدة,,,

,,,,,,,,,,,,,,,,,,,,,,,,,,,,

أخيرا أقول لماذا تم اختراع محددات الوصول أصلا ؟؟ لماذا لانجعل كل محددات الوصول public ليستطيع برنامجنا الوصول لما يشاء من متغيرات داخل الفئة وعدم القلق بشأن هل سيتم الوصول لهذا المتغير أم لا,,,

أقول هنا أن محددات الوصول تم اختراعها لزيادة مستوى التكامل بين فرق البرمجة وبالتحديد بين " مصنعي الفئات " و " مستخدمو الفئات,, "

والتكامل من حيث الابتعاد بقدر الامكان عن الوصول للمتغيرات الخطيرة التي لاينبغي لمستخدم الفئة الوصول اليها ,, وفعلا توجد متغيرات خطيرة كهذه لذا من الرائع عدم ترك الحبل على الغارب,,,

,,,,,,,,,,,,,,,,,,,,,,,,

هنا أقول أننا قد قضينا مشوارا لابأس به لنخطو اول خطواتنا للدخول في عام البرمجة الحقيقية ,,,والى الجلسة المقبلة حيث سنغوص أكثر ,, باذن الله,,,

سأطرح أمثلة قوية من المرات المقبلة ليتضح الفهم بشكل كبير,,,

الدرس الرابع

الان بدأنا في أبجد هوز الفئات ,, وذكرت أن الفئات برمتها ليست الا 6 كلمات أساسية,,
وضحنا مفهوم الوصول للمتغيرات والدوال داخل الفئة ,, ولماذا لايجب أن يكون محدد الوصول دائما public ,,
كما ذكرت سابقا ميزات الفئات مترابطة بشكل كبير ,, بداية بمحددات الوصول والمشيدات وتحميل العوامل بشكل زائد والوراثة وتعدد الاشكال وأنواع الوراثة الهرمية الاخرى والدوال الصديقة والفئات الصديقة وقوالب الفئات",,, وهنا تنتهي السي++ ويبقى فقط الاستمتاع بهذه الميزات لكتابة شيفرة مذهلة"!!
الان بسبب هذا الترابط فان أفضل طريقة للتعلم هي طرح مثال يبدأ صغيرا ونقوم كل مانأخذ ميزة جديدة ندخلها على الفئة التي نطورها ,, وننتظر النتيجة وفائدة هذا التطوير,,
بهذه الطريقة سنعرف الاستخدام الامثل لهذه الميزات بمزيد من الفهم,,
نهاية قبل أن نبدأ ,, للميزات السابقة التي ذكرتها الكثير من القوانين ,, قد يتم نسيانها من قبل المبرمج وقد تعرضه لأخطاء يصعب اكتشافها ,, لذا يجب على المبرمج ان يراجع ميزات الفئات برمتها كل 4 شهور تقريبا ,, ويجب استخدامها في كل برامجنا التالية لترسخ في الذهن وكل ماعودت نفسك على الكتابة الكائنية ستجد أنك كل مرة لاتستطيع الاستغناء عن الفئات أكثر من التي قبلها ,," طبعا ماعدا البرامج الصغيرة جدا التي لاتتجاوز بضعة أسطر لاداعي لاستخدام الفئات فيها"
,,,

```
int *m_data;
int m_SIZE;
void InitSize(int size);
void PutData(int data , int index);
int GetData(int index);

};
```

الفئة واضحة ,, المتغير المهم هنا في الفئة والذي يعبر عن البيانات الفعلية هو المتغر ,, m_data وهو مؤشر من النوع int سنطور المثال لاحقا ليخدم عدة أنواع أخرى غير ال ,, int والسبب في جعله مؤشر لكي نأخذ حرية استخدامه بالحجم المطلوب ولن تكون هذه الفئة ذات معنى مطلقا دون استخدام المئشرات للدلالة على المتغير,, m_data

يوجد متغير اخر اسمه m_SIZE وهو يحتوي دائما على حجم المصفوفة الحالي ,, وليس على عدد البيانات بل الحجم الاقصى,,,

الدالة InitSize تأخذ بارمتر واحد وهو الحجم المراد حجزه للمصفوفة ,,كما هو واضح من اسمها init تعني تمهيد ,, وهنا ستستدعى هذه الدالة كلما أردنا تغيير حجم المصفوفة,,, Array

الدالة PutData أيضا واضح أنها لادخال البيانات في المصفوفة ,, Array تأخذ بارمترين الاول ال data التي نريد ادخالها في المصفوفة ,, والبارمتر الثاني هو الخانة التي سنضع فيها الداتا في المصفوفة,,

الدالة الاخيرة GetData من اسمها أيضا واضح أنها للاستعلام عن data في خانة عنصر معينة,, لها بارمتر واحد هو رقم العنصر ,, لكن لها قيمة اعادة return من النوع int نظرا لأن ال data التي ستعود لنا بالتأكيد من النوع,,, int

الان فقط أدرجت تصريح الفئة وليس التعريف ,, الصريح هو أن أقول للمصرف ,,يامصرف ستأتيك لاحقا فئة اسمها Array بها 3 دوال "بمواصفاتها في الفئة " مع منغيرين " كما ي الفئة أيضا " ,,, ولاتشغل نفسك يامصرف بأن المستخدم قام باستخدام أحد الدوال قبل أن تعرف تعريفها وطريقة عملها,,

وبهذا لن يقلق المصرف من كون ان المستخدم استدعى دالة لايعرف المصرف طريقة تنفيذه ,, لأن تعريفها سيكون في مكان ما من البرنامج وما عللى المصرف الا أن يبحث عنها!!

قبل أن ندخل في تعريف الفئة سأتناول طريقة استخدامها بمثال ,, وبعدها نتعرف على طريقة عملها,,

CODE

```
void main()
{
Array a;

a.InitSize(100);
a.PutData(1100 , 3);
cout << a.GetData(3);

}
```

هنا نستخدم المتغير a لنستخدم الوظاءف التي تقدمها لنا الفئة,,
لاحظ للاستدعاء a.InitSize ممرة اليها القيمة 100 ,, هذا يعني أننا سنهيء مصفوفتنا Array لامكاية استقبال 100 عنصر ,,, يجب دائما أن نمهد للحجم المطلوب قبل استخدام المصفوفة,,
ثم استعملنا PutData لاضافة القيمة 1100 للعنصر رقم 3 من أصل ال 100 عنصر في المصفوفة,,
وأخيرا لأن الدالة GetData كما عرفناها سابقا لها قيمة اعادة return اذا فلها قيمة !! نعم تكون للدالة قيمة ويمكن أن نعاملها تماما كأنها متغير عادي اذا كان لها قيمة اعادة" return لكن ببعض السمكرة التي سنتناولها لاحقا,,"
لذا تجد ال cout قبل الدالة وكأنها ستطبع قيمة الدالة ,, ماهي قيمة الدالة ؟ هي قيمة ال return الذي في تعريف الدالة ,, وسنعرف بعد قليل,,

الان اراكم تتضجرون وتقولون صعبت الامر بلا فائدة يمكن استخدام المصفوفات العادية او المؤشرات بشكل مباشر دون كتابة فئة كامله كهذه",, ولسه بقي التعريف " ,, والاعقد من ذلك طريقة الاستخدام الصعبة والغريبة والطويلة!!
باختصار الفئة سيئة لحد مرعب ولايمكن استخدامها عمليا أبدا,,
أقول أني أريد أن أريكم كيف يمكن تحويل هذه الخرابة الى جنة لكن بالتدريج ,, رويدا رويدا ,, عندما ندخل في الميزات اللاحقة,,,
دعونا لانكثر الكلام وننتقل الى التعريف,,

CODE

```
void Array::InitSize(int size)
{
m_data= new int[size];
m_SIZE = size;
}

void Array::PutData(int data , int index)
{
m_data[index] = data;
}

int Array::GetData(int index)
{
return m_data[index];
}
```

هكذا نكون قد انتهينا!!
قبل أن أدخل في الشرح لاحظ ال :: بين اسم الدالة واسم الفئة التي تنتمي اليها الدالة ,, هكذا يمكن أن يفهم المصرف أننا نقصد الدالة التي مثلا اسمها GetData الموجودة في الفئة التي اسمها Array ,, لأنه قد توجد دالة أخرى اسمها GetData لكن في فئة أخرى اسمها Point مثلا , فسيتحير المصرف عندها ,, لكن بهذا التوضيح ذو ال :: لاتوجد تضاربات في الاسماء اطلاقا,,

الان لكي يفهم المصرف ماذا نعني ب ,, a.InitSize(100) سينتقل المصرف فورا الى تنفيذ الدالة InitSizeصاحبة الفئة المعرف منها الكائن " a كما في الاستخدام السابق " ,, وهي الدالة التي في الفئة,, Array

في تنفيذ الدالة كما هو واضح يتم حجز للمتغير m_data بحجم 100 خانة ,, والسبب في ال 100 أنها قيمة ال size البارمتر الذي استلمته الدالة من الاستدعاء الذي في المثال ,, main لاحظ السطر الثاني ,, m_SIZE = sizeوهو حجم الكائن الذي نملكه ,, دائما مايشير المتغير m_SIZE الى حجم الكائن الحالي " هنا هو " a

الدالة PutData لاتفعل شيئا سوى تمرير القيمة data المستلمة كبارمتر الى المصفوفة الفعلية المحركة للكائن برمته وهي m_data في الخانة رقم ,, index

في مثالنا ستمرر القيمة 1100 الى الخانة رقم 3.

أخيرا الدالة PutData أيضا لاتقوم بشيء سوى أنها تعيد return للقيمة التي في مصوفتنا الفعلية للكائن برقم خانة index وهو البارمتر الوحيد للدالة,,,

في مثالنا ستعاد القيمة 1100 وستطبع على الشاشة,,

أخيرا اقول لاحظ لمحدد الوصول الذي للفئة وهو public لجميع الاعضاء " الدوال والمتغيرات " ,,وهو أمر ليس جيدا كما سنرى الجلسات المقبلة,,,

هكذا نكون قد كونا أساس الفئة التي سنبني شروحنا التالية عليها ,, ونستغني بالتدريج عن بعض خدماتها السيئة !! لاستبدالها بما هو أكثر رحابة منها,,

هكذا نكون قد كونا فهما أفضل لطريقة استخدام الفئات ,, بمثال واضح يكفي تنفيذه ليعمل ,, "الان يمكن لأي شخص أن يكتب فئة كاملة بدون مجهود,,"

سنتحرك في الجلسات بشكل أسرع باذن الله لأنه توجد الاطنان من المعلومات التي يجب علينا أن نعرفها لنرتقي بأمتنا بشكل أفضل ,,

وبالتوفيق,,

```
#include <iostream.h>

class Array
{
public:

        int *m_data;
        int m_SIZE;
        void InitSize(int size);
        void PutData(int data , int index);
        int GetData(int index);

};

void Array::InitSize(int size)
```

```
{
        m_data= new int[size];
        m_SIZE = size;
}

void Array::PutData(int data , int index)
{
        m_data[index] = data;
}

int Array::GetData(int index)
{
        return m_data[index];
}

void main()
{
        Array a;

        a.InitSize(100);
        a.PutData(1100 , 3);
        cout << a.GetData(3);

}
```

الدرس الخامس

بعد أن انتهينا من تشكيل الهيكل الرئيسي لفئة المصفوفة ,, ندخل الان في ادخال التحسينات شيئا فشيئا ,,

في هذا الدرس:

سنتحدث عن المشيدات والمهدمات,, Constructors And Destructors,,

,,,

مثال 1:

CODE

```
() Array
{
}
```

لايحتوي على أي أوامر ,, فهو فقط ليسكت نفسه من اعتراض خطأ,,
الان اذا قام المستخدم بكتابة مشيد بنفسه ,, فالمصرف سيقوم بالغاء المشيد الافتراضي تماما ,, انتبه الى هذا الكلام ,, أي مشيد يكتبه المستخدم سواء له بارمترات أو يشبه المشيد الافتراضي "يعني بدون بارمترات ",, فان المشيد الافتراضي سيتم الغاءه ,,وعمليات الحجز وتمهيد المتغيرات تقع في هذه اللحظة على عاتق المستخدم كاتب الفئة الذي كتب المشيد بنفسه ,, فكل مايحدث أنك تعفي المصرف من مسؤليته!!

الان كما قلنا المشيدات دوال عادية ,, اذن يمكن استدعاءها ,, ولكن كيف ؟
الاستدعاء يتم بشكل الي دون تدخل المستخدم ,, ويكون ذلك عند تمهيد كائن من نوع الفئة ,, مثلا,,
Array X

هذا الامر يقوم بتمهيد كائن اسمه X من النوع Array ويستدعي المشيد الخاص بالفئة الذي ليس له وسيطات ,, سواء كان الافتراضي اذا لم يعرف المستخدم واحد أو الذي عرفه المستخدم ,, ولكن ليس له بارمترات انتبه لذلك,,
حسنا ,, يمكنك بالطبع كما ذكرت كتابة مشيد له بارمترات ,, يأتي السؤال .. لماذا أنشيء واحد له بارمترات في الاصل ؟ واذا أنشأته كيف يمكن استدعاءه ؟

السبب الذي يدعوك لكتابة مشيد له بارمترات ,, هو أن الكائن في بعض الاحيان يحتاج لأن يتزود بمعلومات ابتدائية مهمة ,, يحتاجها وقت انشائه مباشرة ,, مثلا في حالتنا ,,

في مثال المصفوفة نريد أن نمر حجم المصفوفة للمشيد الذي يأخذ بارمتر لكي يقوم بنفس العملية التي كانت تقوم بها الدالة InitSiae ولكن بشكل الي!!
خذ تصريح المشيد الذي سنعرفه لاحقا,,

مثال:2

CODE

```
( Array(int size;
```

لاحظ هنا لتصريح المشيد يوجد بارمتر من النوع ,, int ولايوجد للمشيدات نوع اعادة ,, return فلا يوجد voidأو غيره خلف اسم المشيد,,

الان كيف يمكن استدعاء هذا المشيد من الكود ؟,, على النحو التالي,,

(Array X(10

لاحظ الفرق ,, يوجد بارمتر ممرر للمتغير X لكن بصيغة غريبة نوعا ما حيث نمرر القيمة وقت التمهيد ,, يمكن هكذا أن تنشيء مشيد له 100 بارمتر وكما تشاء من أي نوع

من الانواع ,, مثلا لو أنشأت مشيد له 4 بارمتارت من نوعين,, int char

(Array X(10,'d',656,'y'

مثلا يمكن أن يكون تصريحه يقبل بارمترات بهذا الترتيب!!

حسنا الان عرفنا هذه المعلومات ,, ماذا سنفعل بها في برنامجنا ؟

سنقوم أولا بالتحسين رقم 1 في هذه الجلسة وهو أن نستغني عن خدمة الدالة InitSize بحذفها تماما ,, وننشيء المشيد الذي سيتم استدعاءه تلقائيا ,, التصريح كما في المثال 2 ,, والتعريف كالتالي ,, ومن المستحسن دائما وضعه خارج الفئة مثله مثل الدوال العاديه تماما ,, كما في الجلسة السابقة ,, كالتالي,,

CODE

```
Array::Array(int size)
{

m_data= new int[size];
m_SIZE = size;

}
```

الان يمكن كتابة البرنامج الذي يستخدم الدالة على النحو التالي,,

CODE

```
void main()
{

Array a(100);
a.PutData(1100 , 3);
cout << a.GetData(3);
```

```
}
```

هذه النسخة عملية أكثر من ناحية عدد السطور المفروض كتابتها ليتم تمهيد الكائن بالشكل المطلوب ,,
هنا سطر واحد مقابل سطرين في النسخة الاولى ,,,, وتوجد مزايا أخرى لايمكن استخدامها بطريقة
الدوال العادية ,, سنتطرق اليها الجلسة القادمة,,

الان حللنا مشكلة السطر الاضافي ,, لكن لم نحل مشكلة نسيان تحديد الحجم,,
لحل هذه المشكلة ,, علينا بانشاء مشيد ليس له بارمترات بحيث لو أنشأ المستخدم كائنا كالتالي,,
Array X

سيستدعي المشيد الذي ليس له بارمترات فورا بعد تمهيد المتغير ,, X الان كل مانريده من هذا
المشيد الذي ليس له بارمترات أن يجعل حجم المصفوفة الداخلية m_data بحجم افتراضي ,, ولنقل مثلا
50 ,, بحيث لو قام المستخدم بكتابة:

CODE

```
Array a;
a.PutData(1100 , 3);
```

لن تكون هناك أية مشاكل ,, لكن لو فعلنا هذا دون انشاء المشيد الذي ليس له بارمترات سيحدث
انتهاك يتسبب في فشل البرنامج ,, تصريح المشيد الذي ليس له بارمترات كالتالي:

CODE

```
Array();
```

وتعريفه كالتالي,,

CODE

```
Array::Array()
{
m_data= new int[50];
}
```

اليك النسخة 2 من المصفوفة Array في نهاية الدرس
حتى الان المصفوفة ليست عملية ,, أوامر PutData و GetData سيئة وغير عملية في استخدامها ,,

حيث يتم استدعاء دالة كاملة ,, فقط لادخال بيانات ! و GetData لاسترجاعها ! ,, سنحذفهما أيضا قريبا ,,لكن الجلسة المقبلة سنركز فهم المشيدات والمهدمات بشكل أكبر باذن الله قبل أن ننطلق ,, فربما كانت الامور غامضة حتى الان ,,

```
#include <iostream.h>

class Array
{
public:


        int *m_data;
        int m_SIZE;
//New Changes From Here ,,,
        //void InitSize(int size);
        Array();
        Array(int size);
////////////////////////////
        void PutData(int data , int index);
        int GetData(int index);

};

Array::Array()
{
        m_data = new int[50];
}

Array::Array(int size)
{
        m_data= new int[size];
        m_SIZE = size;
}

/*
void Array::InitSize(int size)
{
        m_data= new int[size];
        m_SIZE = size;
}
*/

void Array::PutData(int data , int index)
{
        m_data[index] = data;
}

int Array::GetData(int index)
{
        return m_data[index];
}

void main()
{
//Using Constroctor With Out Param ..
        Array a;
        a.PutData(1100 , 3);

//Using Constroctor With Parm ...
        Array z(300);
```

```
        z.PutData(777,280);
        cout << a.GetData(3);
}
```

الدرس السادس

في هذا الدرس

تمهيد المتغييرات في المشيدات,,

مشيد النسخ الافتراضي,, Copy Constructor ,,

المهدمات,, Destructors ,,

,,,,,,,,,,,,,,,,,,,,,,,,,,,,

الان مالفرق بين تمهيد المتغيرات وتعيين قيمة للمتغيرات ؟

التعيين كالتالي ,,

```
int x
x= 10
```

هنا تم تعيين القيمة 10 للمتغير x ولكن بعد انتهاء مرحلة التصريح عن المتغير ,,

انما التمهيد ,, فهو تعيين لقيمة لكن وقت التصريح ,,

هذا بالنسبة للمتغيرات العادية في المجال الخارجي أو مجال الدالة ,, وبالطبع البارمترات الممررة للدالة تعمل فقط على مبدأ التمهيد,, حيث سيتم تعيين قيمتها فور انشاءها ,, والقيمة هي البارمتر الممرر,,

لكن الامر مختلف قليلا بالنسبة للفئات ,, فلو صرحنا عن متغير داخل فئة ,, فهل يمكن تمهيد قيمته ؟ ,,

كما نعرف اذا لم يتم التمهيد للمتغير بقيمة وقت تصريحه ,, فانه سيأخذ قيمة عشوائية لو كان في مجال غير المجال العام " الخارجي",,أما لو كان في المجال الخارجي سيأخذ القيمة صفر,,

لكن هل يوجد فرق فعلا بين التمهيد والتعيين ؟

نعم يوجد فرق في الامور الحساسة مثل قيمة المتغير الثابت ,, كيف يمكننا أن نعين قيمة لمتغير const ؟ لاتوجد طريقة مباشرة مطلقا ,, يعني لايمكن كتابة:

```
const int x
x=10
```

سيعترض المصرف فورا أن x ثابت لايمكن تغيير قيمته ,, وقبل ذلك سيعترض لأنك لم تمهد ل x أصلا,,

اذن الحل أن تعين القيمة للمتغير x وقت التصريح ,, ليصبح تمهيدا كالتالي,,

```
const int x =10
```

هكذا تم حل المشكلة ,, لكن ماذا نفعل لو أردنا تعيين قيمة لمتغير عادي أولا داخل فئة ,؟

بداية لايمكننا أبدا أن نقوم بتعيين قيمة أو حتى تمهيد قيمة لمتغير داخل فئة في جسم الفئة نفسه ,,

وانما يجب أن يكون ذلك داخل دالة ,, خذ المثال:

مثال 1:

CODE

```
class Array
{
public :
int m_SIZE = 10;
.
.
.
.
};
```

سيعترض المصرف فورا ,, لأنه لايمكن التمهييد لقيمة داخل جسم الفئة ,, عليك بأن تقوم بذلك داخل جسم دالة ,, والافضل أن تكون داخل المشيد ,,كالتالي مثلا:

مثال 2:

CODE

```
class Array
{
public :
int m_SIZE;

Array()
{
 m_SIZE = 10;
}

.
.
.
.
};
```

هذا المثال سيقوم بالمطلوب تماما ,, لكن تصبح المشكلة في أمر اخر,,,

ماذا لو كانت القيمة المراد التمهيد لها لمتغير في الاصل,, const ؟؟

لو كتبنا المثال كالتالي لن يعمل:

مثال 3:

CODE

```
class Array
{
public :
const int m_SIZE;

Array()
{
 m_SIZE = 10;
}
```

```
};
```

سيعترض المصرف !! وسيرسل رسالة مفادها التالي,,

'm_SIZE' : must be initialized in constructor base/member initializer list

وهو أن قيمة المتغير الثابت يجب تمهيدها ,, وليس تعيينها,,

مافعلناه في المشيد هو تعيين وليس تمهيد ,, يجب أن يكون التمهيد كما ذكرت في وقت الانشاء ,, أو وقت التصريح ,, اذن كيف يمكن الخروج من هذه الازمة ؟

اليك الحل:

مثال 4:

CODE

```
class Array
{
public :
const int m_SIZE;

Array() :m_SIZE(10)
{
}


};
```

هذا المثال سيعمل بدون مشاكل ,, لماذا ؟ ,,, لاحظ للصيغة

CODE

```
Array() :m_SIZE(10)
```

بهذه الطريقة بكتابة اسم المتغير ووضع القيمة بين الاقواس ,, بعد اسم المشيد مباشرة (" فقط هذه الطريقة تعمل مع المشيدات") يتم التمهيد للمتغيرات التي داخل الفئة,,

فبهذه الطريقة لن يعترض المصرف لأن القيمة m_SIZE فعلا يتم التمهيد لها وليس تعيينها,,

,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

أمر اخر هو مشيد النسخ الافتراضي ,, The Copy Constructor وهو نوع المشيدات الثاني الذي يقوم المصرف بانشاءه لك تلقائيا ,, قلنا الجلسة الفائتة أن المشيد الذي ينشئه المصرف بشكل الي ليس له بارمترات ,, اذا لم تنشيء أنت واحدا سيقوم المصرف بانشاءه عوضا عنك,,,

مشيد النسخ الافتراضي كذلك ,, وصيغة تمهيده على الشكل التالي,,

مثال:5

CODE

```
Array(const Array& z);
```

لاحظ هنا للبارمتر ,,, من النوع ,, &Array وهو عنوان من نفس نوع الفئة الخاص بها ,,والبارمتر الممرر سيصبح من النوع const لضمان ثبات قيمته,,

الان بعد كل هذا الكلام ,, مافائدة مشيد النسخ هذا ؟؟

مشيد النسخ يعطي المصرف تعليمات عن كيفية نسخ كائن لكائن اخر ,, في وقت التمهيد ,, خذ المثال عندك,,

مثال:6

CODE

```
void main()
{

Array x;
x.m_SIZE =19;
Array y  =x;

}
```

لاحظ ل

Array y=x

يمكن أن تكتب أيضا

Array y(x)

الذي سيحدث هنا أن الكائن y ستتحول قيمته تماما كما هو الكائن,, x

قيمة الكائن ,, هي قيمة جميع المتغيرات المعرفة داخل هذا الكائن ,, حيث لكل كائن قيم مختلفة للمتغيرات المعرفة داخله,,

كما ذكرت يمكنك أن تعرف طرق أخرى لتمهيد المتغيرات ,, مشيد النسخ الافتراضي ينسخ البيانات المعرفة في الفئة بنفس ترتيبها الموجود ,, يهني m_SIZE لm_SIZE و m_data ل ,, m_data وهكذا,,

لكن قد تأتي عليك أفكار وتحبذ تغيير طريق النسخ التي يقوم بها المصرف ,, عندها يجب عليك كتابة مشيد النسخ بنفسك وتكون صيغة تصريحة كما ذكرت سابقا,,

,,,

اذا كان مؤشرا,,"

الان السؤال ,, ماذا تريد أن تفعل في المهدم ؟ مالذي تريد فعله بالضبط عند تدمر كائن معين وخروجه من نطاق الذاكرة ؟

بالطبع قد يخمن بعضكم الاجابة .. أنه عند تدمر الكائن فسيتم حذف المتغيرات الخاص به التي في المكدس فقط ,, وليست المؤشرات,,

في مثالنا لدينا m_data عند انتهاء أي كائن Array ننشئه لخروجه من مجال تصريح الدالة مثلا ,, لايتم الغاء ذاكرة m_data ولكن يتم حذف كل المتغيرات التي سواها!!!,,

لماذا m_data بالذات ,, كما قلت لأنها مؤشر ,, يجب حذف متغيرات المؤشرات يدويا باستخدام الامر delete والا لن يكترث بك المصرف ,, وسيحدث عند الاستعمال السيء لذلك انتهكات في الذاكرة بشكل كبير ,, مما يضعف أداء البرنامج,,

طيب الحل ,, .؟

الحل في أن يتم تنفيذ أمر الحذف اليدوي ,, كل ما يتم الاستعداد لتدمير كائن معين ,,

ومتى نعرف أن الكائن سيتم تدميره ؟ ماعلينا الا أن نكتب تعليماتنا في المهدم ,, وسيتم استدعاءه اليا عند بداية عملية الهدم !! وبكل سهولة,,

قبل أن أستعرض الامر ,, كما قلت المشيدات ,, اسمها يجب أن يكون باسم الفئة التابع لها ,, أيضا كذلك المهدمات ,, لكن الفرق بينهما ,,

أنه لايوجد الا مهدم نوع واحد وليس له بارمترات مطلقا ,,

حيث المشيدات يمكن أن يكون لها أنواع ذات بارمترات مختلف ,, وأيضا كلا النوعين ليس له نوع اعاده Return مطلقا,,

أخيرا وهو الفرق المهم ,, في الصيغة الكتابية ,, حيث يسبق اسم المهدم العلامة ~ لتفريقه من المشيد ,, ويكون دائما تصريحه على الصيغة:

CODE

```
~Array();
```

الان سنحسن فئتنا Array مرة أخرى بان نستفيد من الميزة الاولى والثالثة التي تعلمناهم في هذه الجلسة ,, وهما تمهيد المتغيرات في المشيد ,, و المهدمات,,

تمهيد المتغيرات بالنسبة للمشيد الافتراضي الذي لايأخذ بارمترات ,, فعندما ذكرت المرة السابقة أن المستخدم قد ينسى اعطاء مصفوفته حجم ابتدائي ,, فان المشيد الافتراضي سيقوم بحجز حجم تلقائي يكون ثابتا ,, حتى يخرج المستخدم من مشكلة انتهاك الوصول لذاكرة غير محجوزة ,, سنمهد لقيمة m_SIZE هذه المرة بدل من تعيينها داخل المشيد الافتراضي ,, النسخة السابقة كانت:

مثال 7:

CODE

```
Array::Array()
{
m_SIZE = 50;
m_data = new int[m_SIZE];
}
```

النسخة الحالية ستصبح:

مثال 8:

CODE

```
Array::Array(): m_SIZE(50)
{
m_data = new int[m_SIZE];
}
```

لم يحدث فرق بالطبع بين المثالين ,, لكن من الافضل التعود على تمهيد المتغيرات وليس تعيينها ,, لأنها قد تحدث بعض المشاكل عند التعامل على مستوى,, const

هذا الامر الاول , ,الامر الثاني ,, كتابة المهدم الذي سيقوم بتحرير الذاكرة المحجوزة للمتغير m_data مباشرة عندما يتم تدمير كائن Array الذي نستخدمه ,, ولأي كائن يتم انشاءه,,
تصريح المهدم كما ذكرت في الاعلى ,, وتعريفه كالتالي,,

مثال 9:

CODE

```
Array::~Array()
{
delete []m_data;
}
```

هكذا سنكون على ضمان أن مصفوفتنا أصبحت امنه أكثر من المصفوفات العادية ,, حيث لاداعي للقلق أبدا بشأن البيانات التالفة والتي تملأ الذاكرة ,, لكي نخرج من كوابيس مشاكل المؤشرات,,
تطورت المصفوفة قليلا ,, لكنها لم ترتقي بعد لمستوى يأهلها للتعامل بها بشكل فعلي ,, سنزيد الامكانيات في الجلسات المقبلة باذن الله,,

خذ النسخة 3 من المصفوفة :

```
#include <iostream.h>

class Array
{
public:

        int *m_data;
```

```
        int m_SIZE;
//New Changes From Here ,,,
        //void InitSize(int size);
        Array();
        Array(int size);
        ~Array();

//////////////////////////////
        void PutData(int data , int index);
        int GetData(int index);

};

Array::Array(): m_SIZE(50)
{
        m_data = new int[m_SIZE];
}

Array::Array(int size)
{
        m_data= new int[size];
        m_SIZE = size;
}

Array::~Array()
{
        delete []m_data;
}


/*
void Array::InitSize(int size)
{
        m_data= new int[size];
        m_SIZE = size;
}
*/

void Array::PutData(int data , int index)
{
        m_data[index] = data;
}

int Array::GetData(int index)
{
        return m_data[index];
}

void main()
{
//Using Constroctor With Out Param ..
        Array a;
        a.PutData(1100 , 3);

//Using Constroctor With Parm ...
        Array z(300);
        z.PutData(777,280);

        cout << a.GetData(3);

}
```

الدرس السابع

وصلنا للتطوير الثالث للمصفوفة Array ورأينا الان كيف انها أصبحت آمنة بالنسبة لحذف المتغيرات من الذاكرة بعد موتها ,, وذلك بواسطة المهدم الذي يتم تنفيذه بشكل الي,,

المهدم والحجم التلقائي هما الميزتان الحقيقيتان حتى الان ,,

في هذا الدرس

تطوير اضافي على الفئة,,

محددات الوصول,, private public

,,,,,,,,,,,,,,,,,,

تطوير اضافي على الفئة,,

التطوير الذي سيطرأ هنا بداية يتعلق بتغيير حجم المصفوفة في وقت معين,,

لنفرض أن المستخدم قد حدد الحجم 100 للمصفوفة وقت التمهيد ,, لكنه قرر لاحقا أن يزيد حجمها أو ينقصه مالذي سيحدث ؟ وكيف يمكن معالجة المشاكل المحتملة ,,؟

لزيادة حجم المصفوفة يجب أن نزيد حجم المتغير m_data في مصفوفتنا ,, لكن تكمن المشكلة أنه اذا حجزنا لعدد خانات للمؤشر m_data أو أي مؤشر اخر "قاعدة عامة " ,, ثم قررنا في لحظة معينة تغيير هذا الحجم مرة أخرى بالزيادة أو النقصان ,, فيجب علينا أن نقوم باستعمال new مرة أخرى , أليس كذلك ؟ وهو الامر الواضح ,, فعلا سنقوم باستعمال ne لكن المشكلة ستكون بعد ذلك ,, كل البيانات التي تم حفظها في مصفوفة المؤشر m_data يتضيع بعد استعمال new مرة أخرى ,, وستكون مشكلة فعلا,,

خذ المثال,,

مثال:1

CODE

```
m_data= new int[size];
m_data[0] = 11;
m_data[1] = 777;
.
.
```

هنا حجزنا 100 خانة للمصفوفة ,, m_data وأضفنا بعض المتغيرات للمصفوفة ,, ثم قرر المستخدم أنه يريد زيادة الحجم في لحظة معينة لتتسع لبيانات جديدة أكثر من 100 عنصر ,, كل ماعليه فعله هو:

مثال 2:

CODE

```
m_data= new int[200];
```

هكذا ستتسع المصفوفة m_data لتستحمل 200 عنصر وليس 100 كما في السابق ,, لكن المشكلة كما ذكرت ستكون كل البيانات التي أضيفت سابقا قد تم محيها !! فما الحل ؟ الحل في أن نضع متغير مؤشر Temp مؤقت سيحفظ عنوان المصفوفة قبل اعادة استدعاء new لزيادة الحجم ,, ثم نمرره مرة أخرى ل m_data بعد زيادة الحجم ,, خذ المثال الكامل الذي سيعمل,,

مثال 3:

CODE

```
m_data= new int[200];
m_data[0] = 11;
m_data[1] = 777;
.
.
int * Temp = m_data;
m_data = new int [200];
m_data = Temp;
```

لاحظ لعملية التبديل مع Temp ستحل كل المشكلة,,
بالطبع سنطور مصفوفتنا لنجعلها تقبل تغيير الحجم في أي لحظة بالزيادة أو النقصان دون خسارة البيانات المسجلة,,

التطوير سيكون على مرحلتين ,, الاولى كتابة دالة جديدة اسمها SetSize تقوم باستلام الحجم الجديد لتتبناه,,
الاخرى .. سنطور الدالة PutData و GetData لتستعلم عن الخانة التي سنضيف فيها البيانات هل هي صالحة أم لا ,, يعني اذا كان حجم المصفوفة الاقصى هو 100 ,, وأراد المستخدم أن يغير بيانات الخانة 101 أو أكثر فلا يمكنه ذلك ,, لأنه خرج من الحدود المسموح بها ,,

في الحقيقة هذا الامر ممكن أن يكون في المؤشرات ,, يمكنك أن تكتب فوق بيانات أي خانة في الذاكرة ("ليس كلها حقيقة ") بدون أن تكون قد حجزت لها !!! وهي تسبب مشاكل كثيرة المبرمج في غنى عنها ,, لذا من الافضل عدم السماح للمستخدم بأن يقوم بذلك ,,
للقيام بهذا التغيير سنضيف أمر تحقق يدعى assert مهمته التحقق من الخانة index التي ستوضع القيمة data فيها ,, اذا كانت أكبر من الحجم الاقصى m_SIZE للمصفوفة m_data سيتم عرض رسالة خطأ للمستخدم بأن الحجم أكبر من الحد الاقصى ,, خذ التغيير الذي سيطرأ على الدالتين GetData و SetData ,,
مثال 4:

CODE

```
void Array::PutData(int data , int index)
{
assert(index<m_SIZE);
m_data[index] = data;
}

int Array::GetData(int index)
{
assert(index<m_SIZE);
return m_data[index];
}
```

لاحظ السطر:

CODE

```
assert(index<m_SIZE);
```

في كلا الدالتين هو التغيير الوحيد الذي سيتحقق من الحجم ,, هذا أمر ,,

الامر الاخر كما ذكرت كتابة الدالة SetSize لتغيير الحجم ديناميكيا ,, دون مشاكل سيكون تصريح الدالة الجديدة كالتالي ,,

CODE

```
void SetSize(int size);
```

والتعريف كالتالي:

مثال 5:

CODE

```
void Array::SetSize(int size)
{
int* Temp = m_data;
m_data = new int [size];
m_data = Temp;
}
```

الامر بسيط للغاية ,, الان لدينا دالة جديدة لنغير الحجم كما نشاء ,, دون مشاكل ,, يمكن أن أقول في هذه اللحظة أن المصفوفة أصبحت قوية فعلا ,, وهي أأمن من المصفوفات الفطرية العادية في السي++

حيث مصفوفتنا الان يمكنها التعامل اذا:

نسي المستخدم تعيين حجم ,, أراد المستخدم تغيير الحجم بعد وقت الحجز للمرة الاولى ,, أخيرا سيتم حذف المتغيرات الميتة من الذاكرة بشكل الي دون أية مشاكل,,

محددات الوصول,, private public

تحدثنا في الجلسة الثانية بشكل مختصر عن محددات الوصول private و,, public
وعرفنا أن الكلمة private و public عندما يكتبان داخل الفئة ليس لهما الا معنى واحد فقط ,, وهو مدى رؤية المتغيرات,,
ماذا أعني بمدى الرؤية ؟
اذا كانت المتغيرات المصرحة في فئة موجودة في جزء ,, public فاننا نستطيع الوصول لهذه المتغيرات من أي مكان من البرنامج مهما كان ,, من داخل فئة أخرى من داخل دالة من المجال الخارجي ...الخ,
أما اذا كان محدد الوصول private فلا يمكن الوصول للمتغيرات الا من الدوال التابعة لهذه الفئة فقط ,,
)أعتقد ان الامر واضح(

الحالة الثالثة هي protected وهي كما نلاحظ واحدة من الكلمات الاساسية الستة المتعلقة بالفئات ,,
هذه ال protected تقع في المنتصف بين مدى تساهل ال public و تشدد ال,, private
لكن ليس لها ميزة الا في الوراثة ,, عندما نصل للوراثة سنتعرف عليه أكثر,,

الان ماالمشكلة في أن تكون كل المتغيرات المعرفة في فئة من النوع? public
من الغريب أن نعرف أن المشاكل تنجم من private لأنك ستتعرض الى أخطاء من المصرف كل مرة تحاول فيها الوصول لمتغير في مجال غير مرئي لموقع الاستدعاء,,

لكن مامشكلة ال public ؟ ال public تعطيك الحرية أكثر من اللازم للوصول لكل المتغيرات الامر الذي سيكون خطرا للغاية في بعض الاحيان ,, خصوصا اذا تعاملنا بطريقة "مصنع الفئة " و "مستخدم الفئة " في هذه الحال ,, سيكون مصنع الفئة قد كتب الفئة وهو يعلم أنهاتعتمد بشكل أساسي مثلا على عدة متغيرات حساسة لاينبغي أن يتم التغيير فيها ,, وكتب ذلك التحذير في ارشادات استعمال الفئة ,, لكن المستخدم لم يعبء بالتحذيرات التي ستذهب به الى الهاوية ,, وفي مشروع كبير غير أحد هذه المتغيرات ,, وحلت الكارثة بعد فترة وليس مباشرة ,, سيدخل المستخدم في دوامة اصلاح هذا الخطأ العسير حقا,,
الحل:
أن يقوم كاتب الفئة بمنع المستخدم من تغيير هذه المتغيرات بشكل مباشر ,, وانما تعريف الية لتغييرها بشكل امن !! كيف ؟
المنع في أن تعرف هذه المتغيرات في الجزء ال ,, private ولايصدر الكود الخاص بالفئة ,, لكي لايستطيع المستخدم مطلقا التغيير فيه,,
تعريف الية التغيير بشكل امن في أن تكتب دالة اسمها set مثلا ويمرر لها بارمتر للقيمة الجديدة للمتغير المطلوب ,, داخل تعريف هذه الدالة عمليات تحقق من صحة هذه القيمة الجديدة ,, فاذا كاتنت ملائمة يتم تعيينها للمتغير ,, والا سيتم رفضها,,
هكذا لن تكون هناك مشاكل لتغيير القيم,,

كيف يمكن أن نرى ذلك بشكل عملي ؟ فلنطور مثال Array مرة أخرى,,

لاحظ للمتغير m_SIZE هذا المتغير الان خطير نوعا ما ,, لايجب تغييره بعشوائية انما يجب أن يتغير فقط باستخدام الدالة SetSize التي طورناها قبل قليل ,, السبب لأنه اذا تم تغيير m_SIZE الى حجم أكبر من حجم المصفوفة ,, فبذلك أمر assert لن يعمل بالشكل الصحيح ولن يمنع المستخدم بأن يضيف في أي مكان من الذاكرة ,, !! اذا مالحل ,؟

كا قلت أن نمنع المستخدم منعا باتا من الوصول لهذا المتغير مباشرة بأن نجعله ,, private لذا لايمكن الا الوصول اليه من الدوال التي عرفت في الفئة ,, ولايمكن أن نعدلها مثلا من ال ,, main هذا أمر ,,

الامر الاخر ,, لنفرض أنن أريد أن أزيد حجم المصفوفة مثلا ,, بمقدار ضعف حجمها الحالي ,, باستخدام الدالة SetSize لسبب ما ,, كيف يمكني أن أعرف الحجم الحالي مع أنني لاألأستطيع الوصول الى المتغير m_SIZE لأنه أصبح private ؟؟

هكذا سأكتب دالة جديدة أسميها GetData وظيفتها أن تعمل return لقيمة m_SIZE الحالية ,,

وتصريحها كالتالي:

مثال 6:

CODE

```
int GetSize();
```

لاحظ الدالة بسيطة لها قيمة اعادة بدون بارمترات ,, التعريف كالتالي:

CODE

```
int Array::GetSize()
{
return m_SIZE;
}
```

هكذا نستعلم عن الحجم الحالي للمصفوفة دون أن نملك الفرصة للتغير المباشر ل m_SIZEفقط نعرف حجمه ,, لكن لايمكن أن نغير فيه لنفسه ,, حيث أن قيمة الاعادة return تمثل نسخة من m_SIZE وليس m_SIZE نفسها",, أعتقد أن الامر واضح.."

بالنسبة لجعل ال m_SIZE في المجال private فموجودة في المثال المرفق ,, فقط لاحظ لجسم الفئة والكلمة الاساسية private وبعدها m_SIZE تابعة لل,, private

,,,

```
#include <iostream.h>
#include <assert.h>

class Array
{
private:

int m_SIZE;

public:

        int *m_data;

        //void InitSize(int size);
        Array();
        Array(int size);
        ~Array();

////////////////////////////
        void PutData(int data , int index);
        int GetData(int index);
// New Changes Here ,,
        void SetSize(int size);
        int GetSize();

};

Array::Array(): m_SIZE(50)
{
        m_data = new int[m_SIZE];
}

Array::Array(int size)
{
        m_data= new int[size];
        m_SIZE = size;
}

Array::~Array()
{
        delete []m_data;
}


/*
void Array::InitSize(int size)
{
        m_data= new int[size];
        m_SIZE = size;
}
*/

void Array::PutData(int data , int index)
{
        assert(index<m_SIZE);
        m_data[index] = data;
}

int Array::GetData(int index)
{
        assert(index<m_SIZE);
```

```
        return m_data[index];
}

void Array::SetSize(int size)
{
        int* Temp = m_data;
        m_data = new int [size];
        m_data = Temp;
}

int Array::GetSize()
{
        return m_SIZE;
}

void main()
{
//Using Constroctor With Out Param ..
        Array a;
        a.PutData(1100 , 3);

//Using Constroctor With Parm ...
        Array z(300);
        z.PutData(777,280);

        cout << a.GetData(3) << endl;
//////////////////////////////////
        z.SetSize(500);
        cout << a.GetData(3);

        cin.get();
}
```