



كلية الهندسة – قسم المعلوماتية

مقرر برمجة 2

ا. د. علي سليمان

محاضرات الأسبوع الثاني والثالث

classes

الفصل الثاني 2025-2026

## محتوى الفصل

1. إنشاء الأصناف واستخدامها.
  2. التمييز بين أعضاء الصنف العامة والخاصة.
  3. تحديد العلاقة بين العناصر والسجلات.
  4. تحديد العلاقة بين العناصر والوحدات.
  5. تحديد واستخدام التوابع الصديقه.
  6. تعريف واستخدام التوابع الصديقه والأصناف الصديقه.
  7. تعريف واستخدام التوابع الأعضاء الثابته والأغراض الثابته.
  8. تعريف توابع البناء وتوابع الهدم للصنف واستخداماتها والبنائي الناسخ.
  9. استخدام الأعضاء الساكنة والأصناف الساكنة.
  10. استخدام الأغراض والتعامل معها ضمن التوابع ومع المؤشرات والمراجع.
1. Create and use classes.
  2. Distinguish between public and private class members.
  3. Define the relationship between items and records.
  4. Determine the relationship between items and units.
  5. Defining and using friendly methods.
  6. using friendly methods and friendly classes.
  7. Definition and use of const member methods and const objects.
  8. Defining the constructor and deconstructor functions and the copyconstructor .
  9. Use static member and static classes.
  10. Using objects and dealing with them within the functions.

المحاضرة من المراجع :

[1]- Deitel & Deitel, C++ How to Program, Pearson; 10th Edition (February 29, 2016)

[2]- د.علي سليمان, البرمجة غرضية التوجه في لغة C++ 2009-2010

# Introduction

## مقدمة

- primitive types or built-in types: هي `int` ، `float` ، `double` ، `char` ، `Bool` .
- compound type: تمثل مجموعة من العناصر أو القيم من نوع بيانات بسيط درسنا منها `strings` ، `arrays` .
- درسنا `procedural programming paradigm` ( `procedures` ، `functions` ، `subroutines` ) لإنجاز المهام المختلفة. حيث اعتمدنا على المنهجية التالية في حل المسائل:
- ✓ تحديد المهام الأساسية الواجب إنجازها لحل المسألة، والتي قد تتضمن تقسيم المسألة إلى مسائل فرعية أبسط وتحديد مهام فرعية لهذه المسائل الفرعية.
- ✓ إنشاء الأفعال اللازمة لأداء هذه المهام والمهام الفرعية كبرامج فرعية، تمرير عناصر البيانات المراد معالجتها من برنامج فرعي إلى آخر.
- ✓ تجميع هذه البرامج الفرعية مع بعضها لتشكيل برنامج واحد أو ربما مجموعة من البرامج والوحدات النمطية `modules` والمكتبات `libraries` والتي تشكل مجتمعة نظاماً لحل المسألة.

- تمت تسمية هذا النموذج الإجرائي للبرمجة نموذج موجه بالأحداث action-oriented وتم اتباع المنهجية التالية في تصميم البرنامج:
  1. تحديد الأغراض objects في المسألة.
    1. ....
  2. تحديد العمليات operations في المسألة .
    1. إذا لم تكن العمليات مسبقاً التعريف predefined، يجب كتابة توابع لأدائها.
    2. إذا كانت هذه التوابع مفيدة من أجل مسائل أخرى يجب تخزينها في مكتبة library.
  3. تنظم هذه الأغراض والعمليات ضمن خوارزمية.
  4. ترميز هذه الخوارزمية على شكل برنامج.
  5. اختبار، تنفيذ البرنامج.
  6. إجراء عمليات الصيانة والتعديل على البرنامج.
- جاءت البرمجة غرضية التوجه Object Oriented Programming لتقدم قراءة جديدة ومطورة للنموذج بالأحداث Action Oriented وللمنهجية المعتمدة في تصميم البرنامج.
- جاءت هذه القراءة انطلاقاً من محاولة الإجابة على السؤالين التاليين:

- ماذا لو كانت أنماط البيانات Pre-defined data types غير ملائمة لطبيعة الأنماط في المسألة؟.
- الحاجة لأنماط البيانات بحيث تملك خاصية إخفاء المعلومات Information hiding، أي إمكانية قيام الأغراض بالاتصال ببعضها البعض من خلال واجهات معرفة ومحددة بدون إمكانية معرفة كيف تم تطويرها (بمعنى آخر إخفاء التفاصيل المتعلقة بها عن الأغراض الأخرى)؟.
- ✓ الحل يكون بإنشاء نمط بيانات جديد لنمذجتها.
- ✓ الاعتماد على مفهوم البرمجة غرضية التوجه والذي يقوم على أساس الاهتمام بالأغراض Objects المأخوذة من السجلات struct أو الأصناف Classes .
- ✓ إذا كان النمط المعرف متعدد الصفات ويملك عدة أفعال خاصة بالتعامل معه، يجب استخدام الأصناف في نمذجته والمطوره عن السجل struct في لغة C والنمط الأكثر استخداماً حلياً الصنف class.

## الأصناف والأغراض 1

الصنف class : أسلوباً لتعريف بنى معطيات مركبة من قبل المستخدم، ذلك لتحقيق هدفين رئيسيين:

- ✓ الأول بناء أنماط بيانات مجردة ADT's تكون فيها قيم عناصر البيانات محتملة وتوابع العمليات المسموح تطبيقها على القيم مغلقة encapsulated في بنية واحدة.
- ✓ الثاني هو الانتقال إلى الأسلوب الغرضي التوجه object-oriented بدلاً من الأسلوب الإجرائي procedural.

تعريف الصنف: يتم إنشاء الأصناف باستخدام الكلمة المفتاحية class وذلك وفق أحد الشكلين الشائعين التاليين:  
الشكل الأول:

```
class ClassName
{
    public:    declarations of public members
    private:  declarations of private members
};
```

```
class ClassNam
{
    private:    //this keyword is optional
                declarations of private members
    public:    declarations of public members};
```

الشكل الثاني:

بالنظر إلى الشكلين السابقين يمكن أن نسجل بعض الملاحظات:

1. تستخدم الكلمتان المفتاحيتان `public` و `private` للتحكم بالوصول إلى أعضاء الصنف وتسميان محددات الوصول `access specifiers` (يمكن أن يتضمن تعريف الصنف محدد وصول إضافي `protected`). الأعضاء المعرفة ضمن القسم `private` مخفية، ويمكن الوصول إليها من قبل أعضاء الصنف و(التوابع، الأصناف) الأصدقاء (`friend (functions, class)`، في حين أن الأعضاء المعرفة ضمن القسم `public` مرئية ويمكن الوصول إليها من قبل جميع أجزاء البرنامج.
2. تكون أعضاء الصنف خاصة `private` بشكل افتراضي، لذلك نرى في الشكل الثاني أن ذكر المحدد `private` اختياري يمكن حذفه، إلا أنه يذكر عادة من أجل زيادة الوضوح.
3. يمكن أن يحتوي تعريف الصنف على عدة أجزاء خاصة وعدة أجزاء عامة (على الرغم من أن ذلك ليس شائع الاستخدام)، في هذه الحالة تحدد الكلمة `private` بداية الجزء الخاص وينتهي هذا الجزء بنهاية الصنف أو ظهور بداية لمحدد آخر وينطبق هذا التعريف على المحددات الأخرى مثل `public`, `protected`, ويولي المحدد (:).
4. إن ترتيب الأجزاء العامة والخاصة ليس مهماً، فبعض المبرمجين يفضلون وضع الجزء العام أولاً والبعض الآخر يفضلون الجزء الخاص أولاً.
5. ينصح عموماً بتعريف البيانات الأعضاء ضمن القسم `private` وتعريف توابع وصول إلى هذه البيانات ضمن القسم `public` لأن ذلك يساعد في حصر التعامل مع القسم الخاص بالتوابع الأعضاء أو الأصدقاء (توابع، أو اصناف) المرتبطة بالصنف.

### الأصناف والأغراض 3

• يمكن أن يتم التصريح عن أغراض باستخدام الصيغة العامة التالية:

`ClassName object_name;`

• يمكن النظر إلى الصنف على أنه تجريد منطقي `logical abstraction` بينما يمتلك الغرض وجوداً فيزيائياً `physical existence`، بكلمات أخرى الغرض هو عبارة عن نسخة أو حالة `instance` من الصنف.

• أوجه التشابه بين الأصناف والسجلات، يمكن أن نلخصها بما يلي:

- ✓ كلاهما يمكن أن يستخدم لنمذجة الأغراض ذات الخصائص المختلفة والمثلة كعناصر بيانات. وبالتالي فهما يستخدمان لتنظيم مجموعات البيانات غير المتجانسة.
- ✓ كلاهما يملكان عموماً نفس الصيغة `syntax`.

• الاختلاف والذي نلخصه بالقول:

- ✓ أعضاء الصنف تكون خاصة `private` (افتراضياً) وأعضاء السجل تكون عامة (افتراضياً). وفي `C++` يمكن التصريح عنها بشكل صريح لتكون خاصة.

تعريف الصنف Time المستخدم مع بيانات الوقت:

```
class Time {  
public: Time(); // constructor  
    void setTime(int,int,int,char); //set hour,minute,second,sep  
    void printStandard(); // print Standard-time format  
    void printMilitary(); // print Military-time format  
private:  
    int hour; // 0 - 23 (24-hour clock format)  
    int minute; // 0 - 59  
    int second; // 0 - 59  
    char sep; // : separator character  
}; // end class Time
```

حيث تم تحديد قيمة الوقت من خلال تحديد الساعة، الدقيقة، الثانية والرمز الفاصل.

- ✓ يبدأ تعريف الصنف Time بالكلمة المفتاحية class.
- ✓ ويتم تحديد جسم الصنف من خلال القوسين { و }.
- ✓ وينتهي التعريف دوماً بفاصلة منقوطة.
- ✓ يتضمن التعريف الحقول الاربع الضرورية له hour ، minute ، second ، sep .
- يتضمن تعريف الصنف Time نماذج لتعريف التوابع الأعضاء للصنف ضمن الجزء public وهي setTime ، printMilitary و printStandard. تشكل التوابع السابقة التوابع الأعضاء العامة public member functions أو الخدمات العامة public services أو واجهة الصنف class interface.
- ✓ نسمي التابع العضو الذي يحمل نفس اسم الصنف بتابع البناء constructor المرتبط بذلك الصنف.
- ✓ الباني يمكن أن يقوم بإعطاء قيم ابتدائية إلى المعطيات الأعضاء لكل غرض من أغراض الصنف أو مهام أخرى تكلفه بها.
- ✓ يتم استدعاؤه تلقائياً عند إنشاء الغرض.
- تعريف الصنف لا يقوم بحجز أي حيز ضمن الذاكرة، فهو ليس سوى تعريف لنمط بيانات جديد يمكن أن يستخدم للتصريح عن كائنات بنفس الطريقة المستخدمة للتصريح عن متحولات من أنماط اولية، مثلاً عند السجلات والاصناف بالكائن، بالمصفوفه، بالمؤشر وبالمرجع.

```
Date d;
Date dA[10];
Date *dPtr=&d;
Date &dRef=d;
```

تعريف كائن d من النمط Date

تعريف مصفوفة اسمها dA من 10 كائنات من النمط Date

تعريف مؤشر الى كائن dptr يحوي عنوان الكائن d من النمط Date

تعريف مرجع اسمه dref للكائن d من النمط Date

تعريف النمط Date: بشكل مشابه، يمكن تعريف النمط Date كما يلي:

```
class Date {
public:    Date();           //constructor
         void print();
private: int month;    int day;    int year;
}; // end class Date
```

- يمكن التصريح عن متحولات من النمط Date كما يلي:
- بالمثل يمكن تعريف Employee (يترك للطالب) أما الصنف Point سنضيف عليه توابع الادخال والاستعادته.

```
class Point {  
public:  
    Point(); // constructor  
    void setX( int ) ; // set x in coordinate pair  
    int getX(); // return x from coordinate pair  
    void setY( int ) ; // set y in coordinate pair  
    int getY(); // return y from coordinate pair  
    void print(); // output Point object  
private:  
    int x; // x part of coordinate pair  
    int y; // y part of coordinate pair  
}; // end class Point
```

- **access specifiers**: تقوم هذا المحددات بتحديد مجال رؤية **scope** مكونات الصنف.
- البيانات الأعضاء والتوابع الأعضاء لصنف تقع ضمن مجال رؤية الصنف **class scope**.
- التوابع غير الأعضاء ضمن مجال رؤية الملف **file scope**.
- التوابع الأعضاء المكتوبه ضمن الصنف يمكنها الوصول إلى أعضائه بعد ذكر اسمها فقط.
- التوابع الأعضاء المكتوبه خارج الصنف يمكن الوصول إليها من خلال اسم غرض أو عنوان غرض أو مؤشر على غرض مرتبط به.
- يمكن إجراء عملية تحميل زائد على التوابع الأعضاء لصنف وذلك بواسطة التوابع الأعضاء الأخرى التابعة للصنف نفسه، للقيام بذلك، يكفي ذكر تعريف نموذج كل نسخة من نسخ التابع الذي نرغب تنفيذ عملية التحميل الزائد عليه ضمن جسم تعريف الصنف ثم نذكر تعريفاً مفصلاً لكل نسخة من هذه النسخ خارج الصنف.

- إذا تم تعريف متحول ضمن جسم تعريف تابع عضو باستخدام نفس الاسم لمتحول ضمن مجال رؤية الصنف، فإن مجال رؤية الصنف يقوم بإخفاء المتحول الموجود ضمنه ويعزله عن متحول مجال رؤية التابع، ويمكن عندها الوصول إلى المتحول المخفي باستخدام عملية تحديد مجال الرؤية :: على اسم الصنف.
- يمكن الوصول إلى أعضاء صنف من خارج مجال رؤية الصنف باستخدام معامل النقطة (.) dot operator مع اسم غرض أو أي عنوان له، ويمكن استخدام معامل السهم (->) arrow operator بدل (.) إذا كان الغرض معرف بمؤشر.
- يمكن أن يتم تعريف التابع العضو ضمن جسم الصنف بشكل مباشر من خلال ذكر نموذج التصريح وعندها يجب أن يعرف خارج جسم الصنف وعندها تستخدم العملية الثنائية لتحديد مجال الرؤية (::) pinary scope resolution operator حيث يتم ذكر اسم الصنف متبوعاً بهذا المعامل ومن ثم تعريف التابع.
- إذا تم تعريف التابع العضو ضمن تعريف الصنف، فإنه يأخذ تلقائياً صفة التابع السطري inline function، ويمكن للتابع العضو المعرف خارج الصنف أن يأخذها من خلال الذكر الصريح للكلمة المفتاحية inline.

- التوابع الأعضاء المعرفة ضمن القسم public للصنف باسم توابع الوصول access functions وذلك لأنه يمكن بواسطتها الوصول إلى البيانات الأعضاء المعرفة ضمن القسم private من خارج الصنف.
- ليس بالضرورة أن يتم تعريف كافة التوابع الأعضاء ضمن القسم public، إذ يمكن أن تعرف ضمن القسم private وعندها تدعى بتوابع الأداء Performance functions لأنه يتم استخدامها لأداء مهمة من قبل التوابع الأخرى في الصنف.
- يمكن تبيان كل ما سبق من خلال إتمام تعريف الصنف Time وتوظيفه ضمن برنامج كامل كما يلي:

```
#include <iostream>
#include <iomanip>
using namespace std;

// defining Time as abstract data type
class Time
{
```

تعريف الصنف Time

```

public:  Time();           // constructor
        void setTime(int,int,int,char); //set hour,minute,second,sep
        void printUniversal()           // print universal-time format
        {   cout << setfill( '0' ) << setw( 2) << hour << sep
            << setw( 2) << minute << sep<< setw( 2) << second;
        } // end function printUniversal ← تعريف التوابع وكتابة printUniversal ضمن public
        void printStandard();           // print standard-time format
private:
        int hour;           // 0 - 23 (24-hour clock format)
        int minute;        // 0 - 59
        int second;        // 0 - 59
        char sep;          // :
};                               // end class Time ← نهاية تعريف الصنف Time
  
```

تعريف التوابع وكتابة printUniversal ضمن public

القسم private للصنف Time

نهاية تعريف الصنف Time

```

Time::Time() ← استخدام :: مع Constructor كونه كتب خارج الصنف
{ hour = minute = second = 0; sep=': '; } // end constructor Time
void Time::setTime( int h, int m, int s, char sp) ← Function setTime in class Time
{
    hour = (h >= 0 && h < 24) ? h: 0;
    minute = (m >= 0 && m < 60) ? m: 0;
    second = (s >= 0 && s < 60) ? s: 0; sep =(sp == ':')? sp : ':';
} // end function setTime
void Time::printStandard() ← Function printStandard in class Time
{
    cout <<((hour==0 || hour==12) ? 12:hour%12)
        <<sep<<setfill('0')<<setw(2)<<minute
        <<sep<<setw(2)<<second<<(hour<12 ? "AM" : "PM");
} // end function printStandard Breakfast, Lunch,teat, Dinner
  
```

```

int main()
{
    Time t;
    cout << "The initial universal time is ";
    t.printUniversal();
    cout << "\nThe initial standard time is ";
    t.printStandard();
    t.setTime( 13, 27, 6, '!' ) ;
    cout << "\n\nUniversal time after setTime is ";
    t.printUniversal();
    cout << "\nStandard time after setTime is ";
    t.printStandard();
}

```

تعريف كائن اسمه t

The initial universal time is 00:00:00 لطباعة القيم الافتراضية printUniversal

The initial standard time is 12:00:00AM لطباعة القيم الافتراضية printStandard

استخدام تابع الاسناد لقيم صحيحة للزمن

Universal time after setTime is 13:27:06 لطباعة القيم المسنده printUniversal

Standard time after setTime is 1:27:06PM لطباعة القيم المسنده printStandard

```

t.setTime( 99, 99, 99, '!' ) ; ← استخدام تابع الاسناد لقيم خاطئة للزمن
cout << "\n\nAfter attempting invalid settings:"
    << "\nUniversal time: "; ← طباعة القيم المصححة 00:00:00
t.printUniversal(); cout << "\nStandard time: ";
t.printStandard(); cout << endl; ← طباعة القيم المصححة 12:00:00AM
system ("pause"); return 0;
} // end main

```

بالنظر إلى البرنامج السابق يمكن أن نسجل الملاحظات التالية:

- ✓ تم تعريف التابع العضو printUniversal داخل الصنف، فيما تم تعريف باقي التوابع الأعضاء خارج تعريف الصنف (سبب استخدام المعامل ::)، سنعتمد فيما تبقى من هذا المساق أسلوب تعريف التوابع الأعضاء خارج تعريف الصنف.
- ✓ يقوم التابع الباني Time() بتهيئة جميع البيانات الأعضاء إلى القيمة صفر وبذلك يتم ضمان أن جميع الأغراض من النمط Time تنشأ بحالة محددة.

## مثال الصنف 5 Time

- ✓ يقوم التابع setTime() بإعطاء قيمة جديدة للغرض من النمط Time مع اختبار صحة القيمة المعطاة، وفي حال عدم صحتها يتم إعطاؤها القيمة صفر للقيم العددية و ( : ) للمحرف الفاصل.
- ✓ يقوم التابعان printStandard و printUniversal بطباعة قيم الغرض من النوع Time بالتنسيق القياسي والتنسيق العسكري.
- ✓ تقوم التعليمة Time t بإنشاء غرض من النمط Time وإعطاء أعضائه القيمة الابتدائية صفر (لاحظ ذلك من خلال طباعة قيم هذا الغرض بالتنسيق القياسي).
- ✓ لاحظ سلوك التابع setTime(99,99,99, '!') عند تمرير قيم غير صالحة للزمن إذ قام بإسناد القيمة صفر لعضو البيانات الذي تم تمرير قيمة غير صالحة له و : بدل الرمز !.
- ✓ لاحظ استخدام المعامل نقطة (.) للوصول للتوابع الأعضاء العامة للصنف من خلال توسطها لاسم العرض واسم التابع العضو، في حين أنه تم الوصول إلى أعضاء الصنف ضمن تعريف الصنف من خلال ذكر اسمها فقط.
- ✓ في حال قمنا بإضافة التعليمة التالية في أي مكان من التابع الرئيسي; t.hour=10 فإن المترجم سوف يعطي رسالة خطأ تفيد بعدم إمكانية الوصول إلى عضو خاص للصنف.

## مثال الصنف 6 Time

✓ سمحت لنا التوابع `setTime()`، `printStandard()`، `printUniversal()` بالوصول إلى البيانات الأعضاء الخاصة من قبل باقي أجزاء البرنامج لإجراء عمليات معينة عليها.

✓ يمكن تعميم هذه الاستراتيجية لدى تصميم الأصناف إذ يتم عادة التخطيط لتصميم الصنف من خلال جعل البيانات الأعضاء ضمن القسم `private` للصنف بحيث يكون الوصول إليها من قبل باقي أجزاء البرنامج مقيداً بالتوابع الأعضاء المعرفة ضمن القسم `public`. ويزود حينها الصنف بمجموعة من التوابع الأعضاء التي تضمن تحقيق كافة عمليات التعامل مع البيانات الأعضاء.

✓ أشهر العمليات، التعديل (كتابة) أو الحصول على (قراءة) لقيم البيانات الأعضاء. تسمى هذه التوابع setter للتعين أو التعديل و getter للقراءة. بالطبع ليس بالضرورة تسميتها بالأسماء `set` و `get` وإنما جرت العادة على تسميتها بما يدل على عملها.

يبين الشكل المعدل التالي للبرنامج السابق استخدام توابع أعضاء لإتاحة الوصول والتعامل مع البيانات الأعضاء الخاصة للصنف حيث تم توزيع تابع الإدخال إلى أربع توابع منفردة تابع لكل حقل من الحقول وكذلك تابع قراءة القيم إلى أربع.

# توزيع توابع الصنف 1 Time

```
// prog2Manara.cpp : main project file.
#include <iostream>
#include <iomanip>
using namespace std;
class Time {
public:
    Time(); //constructor
    void setTime(int,int,int,char); // set hour,minute,second,sp
    void setHour( int) ; // set hour
    void setMinute( int) ; // set minute
    void setSecond( int) ; // set second
    void setSep( char) ; // set separator
    int getHour(); // return hour
```

تم توزيع تعريف التابع set إلى اربع توابع set ←

## توزيع توابع الصنف 2 Time

```

int getMinute(); // return minute ← تم توزيع تعريف التابع get إلى اربع توابع
int getSecond(); // return second
int getSep(); // return separator
void printUniversal(); // output universal-time format
void printStandard(); // output standard-time format
private:
int hour; // 0 - 23 (24-hour clock format)
int minute; // 0 - 59
int second; // 0 - 59
char sep; // :
}; // end clas Time
Time::Time()
{ hour = minute = second = 0; sep=': '; } // end Time constructor

```

## توزيع توابع الصنف 3 Time

```
// set hour, minute and second values
void Time::setTime( int h, int m, int s, char sp)
{   setHour( h) ;   setMinute( m) ;
    setSecond( s) ; setSep( sp) ;
} // end function setTime
// set hour value
void Time::setHour( int h)
{   hour = (h >= 0 && h < 24) ? h: 0; } // end function setHour
// set minute value
void Time::setMinute( int m)
{minute = (m >= 0 && m < 60) ? m: 0; } // end function setMinute
// set second value
void Time::setSecond( int s)
{ second = (s >= 0 && s < 60) ? s: 0;} // end function setSecond
```

تم توزيع كتابة التابع set إلى اربع توابع set

```
// set sep value
void Time::setSep( char sp)
{ sep = (sp == ':' ) ? sp: ':'; } // end function setSep
// return char value
int Time::getHour(){ return hour;} // end function getHour
// return minute value
int Time::getMinute(){return minute;} // end function getMinute
// return second value
int Time::getSecond(){return second;} // end function getSecond
// return sep value
int Time::getSep() { return sep;} // end function getSep
```

```
// print Time in universal format
void Time::printUniversal()
{
    cout<<setfill('0')<<setw(2)<<hour<<sep
        <<setw(2)<<minute<< sep<<setw(2)<<second;
} // end function printUniversal
// print Time in standard format
void Time::printStandard()
{
    cout<<((hour==0 || hour==12)?12:hour%12)
        << sep <<setfill('0')<<setw(2)<<minute
        << sep <<setw(2)<<second<<(hour<12 ? " AM" : " PM") ;
} // end function printStandard
```

```
int main()
{
    Time t;           // create Time object
    // set time using individual set functions
    t.setHour( 13) ; // set hour to valid value
    t.setMinute( 55) ; // set minute to valid value
    t.setSecond( 32) ; // set second to valid value
    t.setSep( ':' ) ; // set sep to valid value
    // use get functions to obtain hour, miunute and second
    cout<<"Result of setting all valid values:\n"
        << " Hour: " << t.getHour()
        << " Minute: " << t.getMinute()
        << " Second: " << t.getSecond()
        << " sep : " <<(char) t.getSep());
```

الاستفادة من توابع set و get

```
// set time using individual set functions
t.setHour( 8765) ; // invalid hour set to 0
t.setMinute( 22) ; // set minute to valid value
t.setSecond( 9870) ; // invalid second set to 0
t.setSep( '!' ) ; // invalid separator set to ':'
// display hour, minute and second sp after setting
// invalid hour and second values
cout<<"\n\nResult of attempting to set invalid hour and"
    << " second:\n Hour: " << t.getHour()
    <<" Minute: " <<t.getMinute()<<" Second: " <<t.getSecond()
    << " sep : " << t.getSep() << "\n\n";
system ("pause");return 0;
} // end main
```

لقد تم إخراج قيمة المحرف بطريقتين الأولى بعد قسرها كمحرف : والثانية القيمة المكافئة لهذا المحرف 58.

## الفصل بين الواجهات والنصوص البرمجية 1

- إن تطوير البرمجيات وتعديلها وكيفية معالجة التوابع للمعطيات أمر من خصوصية المبرمج، ضرورة اخفاء التحقيق لوجود معلومات سرية مثل توليد كلمات سر، طريقة مبتكرة لكتابة خوارزميه وتنفيذها، طريقة تشفير، مع مراعات عدم التعديل في واجهات الربط مع المستخدمين. حيث تعطى التوابع ويشرح ما يحتاج المستثمر لكي ينفذها وما هو الخرج ولا يتم عادةً نشر كيف تعمل التوابع.
- إن الفصل يساعد ذلك المبرمجين المستقلين لبيع مكتبات أصنافهم مع إعطاء النصوص البرمجية المحققه لهذه الأصناف مترجمة.
- إن كل ما يحتاجه المستثمر هو القدرة للربط مع النصوص البرمجية الخاصة بغرض مشتق من صنف، حيث يساعد ذلك المبرمجين المستقلين لبيع مكتبات أصنافهم مع إعطاء النصوص البرمجية المنفذه لهذه الأصناف مترجمة، وهذا ما يجعل المبرمجين محميين ومستقلين والبرامج متنوعة وتنافسية وهذا ما يحقق توصيات هندسة البرمجيات.
- لتحقيق ذلك نضع التصريح عن بنية المعطيات ونماذج التوابع في ملف رأسي بامتداد (.h) ويتم إدراج هذا الملف ضمن أي ملف خاص بمستخدم يريد التعامل مع هذا النمط من البيانات كما ونضع تعريفات كافة التوابع التي تنجز عمليات على هذا النمط في ملف آخر ومن ثم يصبح هذا النمط وتعريفات العمليات عليه متاحة في أي برنامج بمجرد إجراء عملية تضمين للملف الرأسي الحاوي على تعريف نمط البيانات في البرنامج.

```
#ifndef TIME1_H
#define TIME1_H
// defining Time as abstract data type
class Time {
public:
    Time(); // constructor
    void setTime(int,int,int,char); // set hour,minute,second,sep
    void printUniversal(); // print universal-time format
    void printStandard(); // print standard-time format
private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
    char sep; //:
}; // end class Time
#endif
```

إعادة صياغة المثال السابق من خلال تقسيمه إلى ثلاث ملفات:

بداية اختبار إن كان الصنف `Time` غير معرف سيتم تعريفه وإلا لن يعرف مرة ثانية

نهاية تعريف الصنف `Time`

الملف الثاني: يتضمن التوابع التي تطبق على المعطيات.

```
#include <iostream>
#include <iomanip>
#include "time1.h"
using namespace std;
Time::Time()
{
    hour = minute = second = 0; sep=': ';
}
// end Time constructor
void Time::setTime( int h, int m, int s, char sp)
{
    hour = (h >= 0 && h < 24) ? h: 0;
    minute = (m >= 0 && m < 60) ? m: 0;
    second = (s >= 0 && s < 60) ? s: 0;
    sep =(sp == ':')? sp : ': ';
}
// end function setTime
```

الملف الثاني: يتضمن التوابع التي تطبق على المعطيات.

```
void Time::printUniversal()
{
    cout << setfill( '0' ) << setw( 2) << hour << sep
         << setw( 2) << minute << sep
         << setw( 2) << second;
}
    // end function printUniversal
void Time::printStandard()
{
    cout << ((hour==0 || hour==12)?12:hour%12)
         << sep << setfill( '0' ) << setw(2) << minute
         << sep << setw(2) << second
         << (hour<12?"AM":"PM");
}
    // end function printStandard
```

الملف الثالث: يتضمن التابع الرئيس main.

// prog2Ch1Manara.cpp : main project file.

```
#include <iostream>
#include "time1.h"
using namespace std;
int main()
{
    Time t;
    cout << "The initial universal time is ";    t.printUniversal();
    cout << "\nThe initial standard time is ";  t.printStandard();
    t.setTime( 13, 27, 6, ':' ) ;
    cout<<"\n\nUniversal time after setTime is ";t.printUniversal();
    cout << "\nStandard time after setTime is "; t.printStandard();
}
```

الملف الثالث: يتضمن التابع الرئيس main.

```
t.setTime( 99, 99, 99, '!' ) ;
cout<<"\nAfter attempting invalid settings:"<<"\nUniversal time:";
t.printUniversal();
cout << "\nStandard time: "; t.printStandard(); cout << endl;
system("pause"); return 0;
} // end main
```

بالنظر إلى الملفات الثلاثة نجد بعض الملاحظات التوضيحية:

- يساعد التوجيهان #ifndef و #endif المستخدمان ضمن الملف الرأسي والخاصان بمرحلة ما قبل الترجمة بإدراج أو عدم إدراج النص المحصور ضمنها:
- إذا كان الصنف بالاسم TIME1\_H غير معرف مسبقاً أو غير مدرج من قبل، فسيتم تعريف الصنف TIME1\_H باستخدام التوجيه #define وستدرج محتويات الملف الرأسي. وإلا لا داعي لإدراج محتوياته مرة أخرى.
- لاحظ في الملفين الثاني والثالث كيف تم إدراج الصنف (الملف الرأسي TIME1\_H) ضمن قسم التضمينات للوصول إلى محتوياته واستخدامها في أي مكان من البرنامج.

• التابع البنائي **construction** هو تابع خاص للأسباب التاليه:

- ✓ له نفس اسم الصنف ويتم تضمينه تلقائياً ضمن الصنف.
- ✓ ليس له أي قيمة معادة وحتى عدم وجود **void** قبله.
- ✓ يتم استدعاؤه بشكل تلقائي **automatically** عند إنشاء أو اشتقاق أي غرض من الصنف وينفذ جسمه.
- ✓ يقوم بتخصيص الذاكرة **allocated** اللازمة لتخزين قيم متحول الصنف.

• التابع الهادم **destructor** هو تابع خاص كذلك:

- ✓ له نفس اسم الصنف مسبقاً بالحرف **(~) tilde char** ، ويتم تضمينه تلقائياً ضمن الصنف.
- ✓ ليس له أي قيمة معادة و لايملك بارامترات.
- ✓ يتم استدعاؤه بشكل تلقائي لدى مغادرة مجال رؤية الكائن.
- ✓ مهمته تحرير الذاكرة المشغولة من قبل الغرض (الغاء الحجز) **deallocated** .

- نظراً لاستدعاء الباني تلقائياً عند إنشاء كائن من الصنف وبدء حياته وهو الباني بدون بارامترات.
  - يمكن الاستفادة من تجهيز initialize الكائن وأي شيء يرغب بتنفيذه مثل رساله ترحيبية تكتب في الباني.
  - تعريف عدد من التوابع البنائية للصنف Time ويعرف بالتحميل الزائد للتوابع البنائية نذكر منها.
- ✓ 1- بإسناد قيم ابتدائية قد تكون صفر للبيانات الأعضاء (العددية) للأغراض التي يتم إنشاؤها يعرف بالباني الافتراضي:

```
Time::Time()
{
    hour = minute = second = 0; sep='!';
} // end Time constructor
```

✓ كما ويمكن استخدام لائحة التجهيز Initialization list بالشكل:

```
Time::Time() : hour (0), minute (0), second (0), sep('!')
{
} // end constructor Time
```

هنا تم كتابة اسم التابع وقوسيه، يليه '!' ثم كل خاصية يليها قوسان وضمنهما القيمة وتفصل الخواص عن بعضها بعضاً بفواصل، وهذه الطريقة اسرع من الطريقة السابقة، ويمكن استخدام الجسم لتنفيذ أعمال أخرى.

## Constructors and destructors

### التوابع البنائية والتوابع الهادمة (الدمرة) 3

✓ 2- بتمرير بارامترات للتابع الباني (الباني مع بارامترات) ويؤدي وظيفة مزدوجة، من جهة إنشاء الغرض وإسناد القيم الممرره وهنا نموذج التابع ضمن الصنف يجب أن يملك بارامترات:

```
Time::Time(int hr,int mn,int sc, char sp)
{ hour = hr; minute =mn; second = sc; sep= sp; } //end constructor
```

ويمكن أن يكون باستخدام لائحة التجحيز:

```
Time::Time(int h, int m, int s, char c): hour (h), minute (m), second (s), sep(c)
{
} // end constructor Time
```

```
Time t(15,25,54,':');
Time t1=Time(20,15,43,':');
```

- يمكن استخدام أسلوبين مختلفين لإنشاء الأغراض:
  - ✓ الأول باستخدام تعليمة من الشكل التالي:
  - ✓ الثاني باستخدام تعليمة من الشكل التالي:
- إن كلا الأسلوبين يقوم بإنشاء الغرض وتمرير قيم الوسطاء إلى بياناته الأعضاء.

- توجد بعض الفوارق التقنية البسيطة بين التصريحين السابقين وهذه الفوارق متعلقة بالتابع الباني الناسخ `.copy constructor`.
- عموماً فإن الأسلوب الأول هو الأكثر شيوعاً وسنقوم باستخدامه في أغلب أمثلة هذا المساق.
- تكمن أهمية استخدام توابع بناء ذات بارامترات في أنها تجنب المبرمج استدعاء توابع إضافية لتهيئة متحول أو أكثر ضمن الغرض.
- يمكن الدمج بين التابعين البنائين السابقين المستخدمين لبناء الأغراض من الصنف `Time` باستخدام الوسيط الافتراضية ضمن لائحة بارامترات التابع:

// Assigning and passing parameters to the constructor function

```
Time(int hr=0,int mn=0,int sc=0, char sp=':')  
{  
    hour = hr;  
    minute =mn;  
    second = sc;  
    sep=sp;  
}
```

- إن استخدام الوسطاء الافتراضية ضمن التابع البنائي، تتيح إنشاء أغراض بدون تمرير قيم لجميع بياناتها الأعضاء، يتم تهيئة البيانات الأعضاء التي لم تمرر إليها قيم، بقيم ابتدائية محددة ضمن التصريح عن التابع البنائي.

```
int main()
{
    Time t1; //hour=0,minute=0,second=0,sep=':'
    Time t2(10); //hour=10,minute=0,second=0, sep=':'
    Time t3(2,55); //hour=2,minute=55,second=0, sep=':'
    Time t4(13,22,54,':'); //hour=13,minute=22,second=54, sep=':'
    t1.printUniversal(); cout<<endl;
    t2.printUniversal(); cout<<endl;
    t3.printUniversal(); cout<<endl;
    t4.printUniversal(); cout<<endl;
    system("pause"); return 0;
} // end main
```

### ملاحظات:

- في حال لم يحتو الصنف على تابع باني أو تابع هادم معرفين من قبل المستخدم فإن المترجم يقوم باستخدام توابع بناء وهدم افتراضية يقوم بإنشائها إلا أن ذلك يؤثر على قيم البيانات الأعضاء وقد يؤدي إلى حالات غير مرغوبة.
- يمكن أن يحتوي الصنف على أكثر من تابع باني أي التحميل الزائد للتابع الباني، في حين لا يمكن أن يحتوي الصنف سوى على تابع هادم وحيد.
- كحالة خاصة لاستخدام التوابع البنائية، يمكن في حال كون التابع الباني ذو بارمتر وحيد استخدام الأسلوب التالي في إنشاء الأغراض:

```
class Test { public:      Test(int x) { var1 = x; }
    .....
private:      int var1;};
int main()
{      Test obj1 = 987; // passes 987 to var1
    .....
    system("pause");      return 0; } // end main
```

- ترتيب استدعاءات التوابع البنائية والهادمة.
  - ✓ يتم تنفيذ تابع البناء للأغراض عندما يتم التصريح عن الغرض مهما كانت دورة حياته.
  - ✓ توابع الهدم يتم تنفيذها بعكس ترتيب تنفيذ توابع البناء وتهدم الأغراض المحلية (auto)locally أولاً.
  - ✓ في حال كان الغرض ساكن static سيتم حذفه بعد حذف الأغراض المحلية وبنفس المنطق (الغرض الذي يتم إنشاؤه أولاً يتم تدميره أخيراً)، وأخيراً الأغراض العامة globally.
  - ✓ الأغراض الشمولية global objects يتم تنفيذ توابع البناء قبل البدء بتنفيذ التابع main()، وبنفس ترتيب التصريح عنها، ولا يمكن التكهن بترتيب تنفيذ التوابع البنائية الشاملة الموزعة ضمن ملفات عدة.
  - ✓ يتم تنفيذ التوابع الهادمة للأغراض الشاملة بعد انتهاء التابع main() وقد لا تظهر على شاشة الخرج في معظم البيئات.
  - ✓ نعيد تعريف تابعي البناء والهدم بحيث يقومان بإرسال رسائل إلى الخرج لنتمكن من معاينة توقيت تنفيذها.

// constrDeconP47.cpp : main project file.

```
#include <iostream>
using namespace std;
```

```
class Time {
public:
```

```
Time(int hr,int mn,int sc, char sp)
```

```
{ cout<<"construction object by values: ";
```

رسالة تظهر عند تنفيذ التابع البنائي

```
cout<<hr<<"\t"<<mn<<"\t"<<sc<<endl;
```

إخراج قيم بيانات الكائن الذي ينفذ له التابع البنائي

```
hour = hr; minute =mn; second = sc; sep= sp;
```

```
}
```

```
~Time()
```

```
{ cout<<"deconstruction object having values: ";
```

رسالة تظهر عند تنفيذ التابع الهادم

```
cout<<hour<<"\t"<<minute<<"\t"<<second<<endl;
```

إخراج قيم بيانات الكائن الذي ينفذ له التابع الهادم

```
}
```

private:

```
int hour;           // 0 - 23 (24-hour clock format)
int minute;        // 0 - 59
int second;        // 0 - 59
char sep;          //:
}; // end class Time
```

```
Time t1(10,20,30,':');
Time t2(30,40,50,':');
```

void main()

```
{ cout<<"MAIN START HERE "<<endl;
  {Time t3(15,25,35,':');
  static Time t4(40,45,50,':');
  Time t5(11,22,33,':');}
  cout<<"MAIN STOP HERE "<<endl;
  system("pause");
} // end main
```

تنفيذ التابع الباتي للكائن الشامل t1

تنفيذ التابع الباتي للكائن الشامل t2

تنفيذ التابع الباتي للكائن المحلي t3 نوع auto

تنفيذ التابع الباتي للكائن المحلي t4 نوع static

تنفيذ التابع الباتي للكائن المحلي t5 نوع auto

تنفيذ تابع الهدم عند القوس النهائي للتابع الرئيس main وبالتالي لن نرى الخرج في بعض البيئات عند وضع اقواس إضافية قبل اشتقاق الكائنات { ومغلق قيل نهاية التابع } سيظهر الهدام لـ t5 ثم t3 فقط

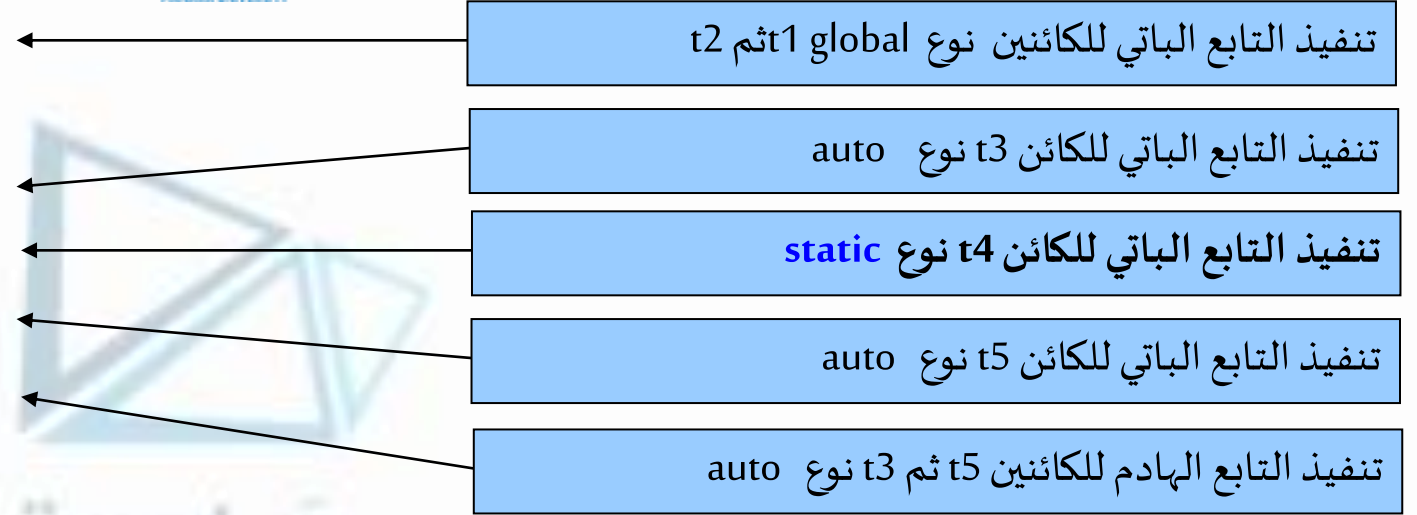
## Constructors and destructors

## التوابع البانية والتوابع الهادمة (المدمرة) 10

```

construction object by values: 10  20  30
construction object by values: 30  40  50
MAIN START HERE
construction object by values: 15  25  35
construction object by values: 40  45  50
construction object by values: 11  22  33
deconstruction object having values: 11 22  33
deconstruction object having values: 15 25  35
MAIN STOP HERE
Press any key to continue . . .

```



- بناء الأغراض t1 و t2 قبل البدء بتنفيذ التابع main والهدم بعد الانتهاء من هدم جميع الأغراض التي عرفت ضمن التابع main.
- بناء الأغراض t3، t4 و t5 بنفس ترتيب التصريح عنها إنما هدم الأغراض t5 ثم t3 (أغراض محلية أتوماتيكية) تم أولاً ومن ثم تم هدم الغرض t4 (غرض محلي ساكن) ويكون بنهاية التابع الرئيس.
- تلخيصاً إن علاقة تنفيذ توابع الهدم بصفوف التخزين كما يلي: ينفذ الهدم بترتيب معاكس للأنشاء (مبدأ الكدسة LIFO كل نوع في مكده) مع مراعاة مدة حياة الغرض لدى هدمه، حيث يتم أولاً هدم جميع الأغراض من النوع المحلي الأتوماتيكية auto، ثم الساكنة static ثم الشاملة global، حيث تهدم الأغراض ذات صف التخزين auto التي تملك حياة على مستوى الكتلة التي تعرف ضمنها وبالتالي تهدم عند الخروج من مجال رؤيتها (كتلتها) ثم الساكنة التي تملك حياة على كامل التابع الرئيس main وبالتالي تدمر عند الخروج من main أما الشاملة فتعرف قبل main وحياتها على كامل الملف وتدمر بعد الخروج منه.

- التحميل الزائد للتوابع هو من أحد أهم المفاهيم المعرفة للبرمجة بلغة ++C، ليس فقط لأنها تدعم تعدد الأشكال وقت الترجمة compile-time polymorphism بل لأنها أيضاً تضيف مرونة وسهولة في الاستخدام.
- تذكره: التحميل الزائد للتوابع – كما سبق وعرفناه لدى دراستنا للتوابع – هو عملية استخدام نفس الاسم لتابعين أو أكثر. يكمن سر التحميل الزائد في أن كل إعادة تعريف للتابع تملك بصمة مختلفة signature أي يجب أن تستخدم أعداداً مختلفة منها أو أنماطاً مختلفة من البارامترات أو ترتيباً مختلفاً لها أو أكثر من حالة (No. of Args, Types of Args, Order of Args)، وبهذه الطريقة يعرف المترجم أي تابع عليه أن يقوم باستدعائه لتنفيذه في أي حالة.
- يمكن أن يتم التحميل الزائد لأي من التوابع الأعضاء في الصنف، إلا أن المثال الأكثر شيوعاً هو التحميل الزائد للتوابع البانية.
- التحميل الزائد للتوابع البانية يتيح إمكانية احتواء الصنف على أكثر من تابع باني واحد وهو أمر نلجأ إليه لأحد الأسباب التالية:
  - ✓ زيادة المرونة إذ أن وجود أكثر من تابع باني يترك للمستخدم حرية اختيار الطريقة الأفضل لبناء غرض بحسب ظروف المسألة.
  - ✓ إتاحة إمكانية إنشاء أغراض مهيئة بقيم ابتدائية أو غير مهيئة.
  - ✓ تعريف التوابع البانية الناسخة.

## Default copying for objects

## النسخ الافتراضي للأغراض

- تحوي جميع الأصناف في C++ طريقتين افتراضيتين للنسخ وهما:
  - ✓ النسخ بالتهيئة.
  - ✓ النسخ بالإسناد.
- في كلا الطريقتين يتم إنشاء نسخة من غرض ما من خلال إنشاء نسخ من أعضاء البيانات للغرض بايت بايت، في حال كون هذه الطريقة الافتراضية غير ملائمة لاحتياجاتنا يمكن لنا أن نعرف تابع باني ناسخ copy constructor من أجل الطريقة الأولى.
- إن معامل الإسناد محمل تحملاً زائداً لتحقيق الطريقة الثانية، ويمكن تحميل المعامل وفق ذوق المبرمج كذلك.
- للتوضيح: افترض لدينا الأغراض bedTime, mealTime من الصنف Time عندئذ فإن التصريح التالي: `Time t=bedTime;` يعتبر تصريحاً مسموحاً عن غرض من الصنف Time وهو شبيه بالتصريح: `Time t(bedTime);` وكلاهما يقوم بتخصيص الذاكرة للغرض t ومن ثم نسخ أعضاء الغرض bedTime إلى مواقع الذاكرة تلك، أي تهيئة الغرض t بنسخة عن bedTime وبالمثل تحصل نفس التهيئة في حال كتبنا:
 

```
Time t=Time(11,30,55,':');
```
- والتي تستخدم الباني ذو القيم الصريحة لبناء غرض Time مؤقت ومن ثم نسخه إلى t.
- الطرق السابقة تعبر عن عملية النسخ بالتهيئة، بالإضافة إلى ذلك تتيح لغة C++ إجراء عملية النسخ بالإسناد كما يلي: `t=mealTime;` هذه العملية ستقوم بنسخ أعضاء mealTime إلى t لتحل محل القيم السابقة.

- إن احترام مبدأ الميزات الأقل principle of least privilege يعتبر من المبادئ الأساسية للبرمجة الجيدة.
- معظم الأغراض تملك طبيعة متغيرة.
- بعضها لا تملك طبيعة متغيرة يمكن التصريح على أنها ثابتة باستخدام الواصف const كما في المثال التالي:

```
const Time noon(12,0,0,':');
```

- تم تعريف الغرض الثابت noon من النمط Time وإعطاءه القيمة الممررة من خلال التابع الباني، ويتم توليد خطأ عند محاولة تغيير قيمة هذا الكائن، ويضبط ذلك أثناء الترجمة وهذا ما يتيح للمترجم العمل بسرعة أكبر مع الكائنات الثابتة مقارنة بالمتغيرة.
  - تمنع المترجمات وصول التوابع غير الثابتة لغرض ثابت، ولهذا يتم بناء توابع ثابتة، هذه التوابع يمكن لها أن تتعامل مع الغرض ولا يمكنها أن تقوم بتعديله.
  - تسمح المترجمات وصول التوابع الثابتة لغرض غير الثابتة.
  - تكتفي بعض المترجمات بإعطاء تحذير فقط warning عندما تتم المحاولة لتغيير عضو ثابت.
  - لا يتم استخدام الواصف const مع توابع البناء أو الهدم للأغراض الثابتة.
  - يسمح لتوابع البناء أن تغير الغرض حتى ولو كان ثابتاً لكي يأخذ قيمة مستقرة ولا يمنع ذلك التوابع المدمرة من القيام بعملها.
- نبين كل ما تقدم من خلال إجراء التعديلات التالية على الصنف Time:

الملف الأول: الرأسى time.h وفيه يتم وضع تعريف الصنف.

```
#ifndef TIME_H
#define TIME_H
class Time { public:      Time(int=0,int=0,int=0,char=':');// default constructor
    void setTime(int,int,int,char); // set hour,minute,second,sep
    void setHour( int) ;           // set hour
    void setMinute( int) ;         // set minute
    void setSecond( int) ;         // set second
    void setSep( char) ;           // set separator// Constant functions
    int getHour() const;           // return hour
    int getMinute() const;         // return minute
    int getSecond() const;         // return second
    int getSep() const;            // return separator
    void printUniversal() const;   // print universal time
    void printStandard();          // print standard time
private:   int hour;              // 0 - 23 (24-hour clock format)
           int minute;            /* 0 - 59 */
           int second;           // 0 - 59
           char sep;             /* : */ };
// end class Time
#endif
```

الملف الثاني: يتضمن التوابع التي تطبق على المعطيات.

```
#include <iostream>
#include <iomanip>
#include "time.h"
using namespace std;

Time::Time( int hour, int minute, int second, char sep)
{ setTime( hour, minute, second, sep); } // end constucter Time

void Time::setTime(int hour, int minute, int second, char sep)
{ setHour( hour) ; setMinute( minute) ;
  setSecond( second) ; setSep(sep) ; } // end setTime

void Time::setHour( int h)
{ hour = (h >= 0 && h < 24) ? h: 0; } // end setHour

void Time::setMinute( int m)
{ minute = (m >= 0 && m < 60) ? m: 0; } // end setMinute

void Time::setSecond( int s)
{ second = (s >= 0 && s < 60) ? s: 0; } // end setSecond
```

الملف الثاني: يتضمن التوابع التي تطبق على المعطيات.

```

void Time::setSep(char sp)
{   sep = (sp == ':') ? sp: ':'; } // end setSep

int Time::getHour() const
{   return hour; } // end getHour

int Time::getMinute() const
{   return minute; } // end getMinute

int Time::getSecond() const
{   return second; } // end getSecond

int Time::getSep() const
{   return sep; } // end getSep

void Time::printUniversal() const
{   cout << setfill( '0' ) << setw( 2 ) << hour << sep << setw( 2 ) << minute << sep
    << setw( 2 ) << second; } // end printUniversal

void Time::printStandard()
{   cout << ((hour==0 || hour==12)?12:hour%12) << sep << setfill( '0' ) << setw(2) << minute
    << sep << setw(2) << second << (hour<12? " AM" : " PM" ) ; } // end printStandard

```

الملف الثالث: يتضمن التابع الرئيس main.

// constFunConstObj.cpp : main project file.

```
#include <iostream>
#include "time.h"
using namespace std;
int main()
{
Time wakeUp( 6, 45, 0, ':' ) ;    // non-constant object
    const Time noon( 12, 0, 0, ':' ) ;    // constant object
    wakeUp.setHour(18); //non-const object non-const member function
//1    noon.setHour( 12 ) ; //const object non-const member function
    wakeUp.getHour(); // non-const object const member function
    noon.getMinute(); //const object const member function
    noon.printUniversal(); // const object const member function
//2    noon.printStandard(); // const object non_const member function

    system("pause");    return 0;
} // end main
```

تم تعليق السطر noon.setHour( 12 ); بسبب محاولة تغيير قيمة الساعات لكائن ثابت

تم تعليق السطر noon.printStandard() بسبب محاولة تابع غير ثابت الوصول لكائن ثابت

بعض الملاحظات على البرنامج السابق

- تم تعريف التوابع `getHour()`، `getMinute()`، `getSecond()` و `printUniversal()` على أنها توابع **ثابتة**. إن هذه التوابع تستطيع الوصول إلى أعضاء الصنف ولكنها لا تستطيع تعديلها، لذا نلاحظ أن الشيفرة الخاصة بهذه التوابع لا تتضمن أي عمليات تعديل على البيانات الأعضاء. ولهذا السبب أيضاً لم يكن من الممكن تعريف التوابع `setHour()`، `setMinute()` و `setSecond()` على أنها توابع ثابتة لأن شيفرتها تتطلب إمكانية تعديل قيم البيانات الأعضاء.
- **تم التعبير عن أن التابع الثابت** من خلال وضع الواصف `const` في **نهاية** سطر التصريح عن التابع، في حين **تم التعبير عن الغرض أنه غرض ثابت** من خلال وضع الواصف `const` في **بداية** سطر التصريح عن الغرض.
- بالنظر إلى المقطع البرمجي المضمن ضمن ملف الاختبار نجد أن هناك **استخدامان خطأ** تم تعليقهما وهما:  

```
noon.setHour( 12);           // const object non-const member function  
noon.printStandard();       // const object non_const member function
```

 وذلك لأن الغرض `noon` هو غرض ثابت، ويحاول التابعان غير الثابتين `setHour` و `printStandard` الوصول إلى قيم البيانات أو تغييرها لهذا الغرض الثابت وبالتالي يعطي المترجم رسائل خطأ تشير إلى هذا الأمر.
- إن باقي الاستخدامات شرعية بما في ذلك قيام تابع البناء وهو تابع غير ثابت بإعطاء قيم ابتدائية للغرض الثابت `noon`.

- تذكرة: بما أن البيانات الأعضاء الخاصة تكون مرئية على مستوى الصنف وبالتالي لا يمكن الوصول إليها إلا من داخل الصنف والتوابع الأعضاء للصنف، مما تطلب إيجاد مفهوم الصداقة، إذ يمكن منح صنف و/أو تابع غير عضو إمكانية الوصول إلى البيانات الأعضاء الخاصة (private) ضمن الصنف بأحد طريقتين:

1. اعتراف الصنف بصداقة تابع له وذلك باستخدام المعرف friend حيث يتم تضمين نموذج prototype ضمن الصنف مسبقاً بالمعرف friend ومن ثم تعريفه في أي مكان من البرنامج وكعادة برمجية جيدة توضع التعاريف مابعد رأس الصنف قبل أية محدد.

2. تعريف هذا التابع كتابع عضو ضمن صنف معترف له بالصداقة من قبل الصنف المراد الوصول إلى بياناته الأعضاء الخاصة. ويكون ذلك من خلال إضافة اسم الصنف الصديق مسبقاً بالمعرف friend ضمن تعريف الصنف المراد الوصول إلى بياناته الأعضاء.

- يمكن إيضاح هاتين الطريقتين من خلال الأمثلة التالية:

## Friend functions, friend classes



## التوابع الصديقة، الأصناف الصديقة 2

### الطريقة الأولى: تعريف التابع على أنه صديق

```
#include <iostream.h>
class MyClass {
    friend int sum(MyClass x);
int a, b;
public:    void set_ab(int i, int j);
};

void MyClass ::set_ab(int i, int j) { a = i;    b = j;}
int sum(MyClass x){    return x.a + x.b;}
int main(){    MyClass n;        n.set_ab(3, 4);        cout << sum(n);    return 0;}
```

- التابع sum() ليس تابعاً عضواً في الصنف MyClass إلا أنه يستطيع الوصول بشكل كامل للأعضاء الخاصة.
- التابع sum () تم تعريفه بدون معامل الارتباط :: واستدعاؤه دون استخدام معامل النقطة مما يدل أنه ليس تابعاً عضواً في الصنف.

الطريقة الثانية: تعريف التابع كتابع عضو ضمن صنف صديق

```
#include <iostream.h>
class TwoValues { friend class Min;          int a;          int b;
public:          TwoValues(int i, int j) { a = i; b = j; } };

class Min {public:          int min(TwoValues x) ;};

int Min::min(TwoValues x) {          return x.a < x.b ? x.a: x.b;}
int main() { TwoValues ob(10, 20); Min m; cout << m.min(ob); return 0; }
```

- يستطيع الصنف Min الوصول إلى الحقول الخاصة a و b المصرح عنها ضمن الصنف Twovalues.
- الصداقة صفة ممنوحة ولا يمكن طلبها ولكن يمكن الانسحاب عنها، أي لكي يكون صنف A صديق للصنف B يجب على الصنف B أن يتضمن تصريحاً بأن الصنف A هو صديق له.
- صفة غير انعكاسية، أي إذا كان الصنف A صديق للصنف B فإن هذا لا يعني أن الصنف B صديق للصنف A.
- صفة غير متعدية، أي إذا كان الصنف A صديق للصنف B، و B صديق للصنف C فهذا لا يعني أن A صديق C.
- الصداقة غير موروثة، أي الأصناف المشتقة من صنف ما لا تكون صديقة للأصناف الصديقة للصنف الأصل.

members Using object from defined class as data  
(Composition)



استخدام أغراض من صنف معرف كبيانات أعضاء  
لصنف آخر (Composition) 1

- يمكن للبيانات الأعضاء في الصنف أن تكون من أي نوع سواء كان مسبق التعريف أو كان نوعاً معرفاً من قبل المستخدم باستخدام مفهوم الأصناف، حيث أن استخدام صنف ضمن صنف يدعى **Composition or Aggregation** وفق العلاقة.
- إذا كان أحد البيانات الأعضاء في الصنف من نوع معرف من قبل المستخدم فإن نقطتين أساسيتين يجب معرفتهما من قبل المبرمج من أجل الاستخدام الأمثل لهذا الصنف وهما:
  - ✓ كيفية التعامل مع التابع الباني للصنف المستخدمة أغراضه كأعضاء في الصنف.
  - ✓ كيفية الوصول إلى الأعضاء العامة لذلك الصنف.
- نوضح هذين الأمرين مباشرة من خلال العمل على مثال الصنف Employee للتعامل مع بيانات الموظف وذلك بإضافة صفتين جديدتين للموظف هما تاريخ الولادة وتاريخ التوظيف.
- إن هاتين الصفتين هما من النوع Date الذي قمنا بتعريفه أيضاً في وقت سابق.
- المقاطع البرمجية الكاملة لبناء هذين الصنفين، ونظراً لطول هذه المقاطع فإننا سنقوم بتسطير مقاطع الشيفرة التي تهتمنا في هذه الفقرة للتركيز عليها.

الملف الرأسي Date.h

```
// header Date.h
#ifndef DATE_H
#define DATE_H
    class Date {
public:
    Date(int=1,int=1,int=1900); // default constructor
    void print() const;        // print date in month/day/year format
    ~Date();                   // provided to confirm destruction order
private:    int month;        // 1-12 (January-December)
            int day;          // 1-31 based on month
            int year;         // any year
            int checkDay( int) const; //function to test valid date //
};
#endif
```

توابع الملف الرأسي Date.cpp

```
#include "date.h"
#include <iostream>
using namespace std;
Date::Date( int mn, int dy, int yr)
{   if (mn > 0 && mn <= 12)           // validate the month
        month = mn;
    else {month = 1;                   // invalid month set to 1
        cout<<"Month " <<mn<<"invalid.Set to month 1.\n";   }
    year = yr;                         // should validate yr
    day = checkDay( dy) ;              // validate the day //
    cout << "Date object constructor for date ";
    print();        cout << endl;
}                                       // end Date constructor
```

توابع الملف الرأسي Date.cpp

```

void Date::print() const
{ cout<<month << '/' << day << '/' << year; } // end function print
Date::~Date(){cout << "Date object destructor for date ";
    print();    cout << endl;} // end ~Date destructor
int Date::checkDay( int testDay) const
{ static const int daysPerMonth[ 13 ] =
    {0,31,28,31,30,31,30,31,31,30,31,30,31};
    if (testDay>0 && testDay<=daysPerMonth[ month ])return testDay;
    if (month==2 && testDay==29 &&(year%400==0 || (year%4==0 && year%100!=0)))
        return testDay;//400, 1600, 2000 leap year and 100,1900, 2100 not leap year
        /* calendar year > solar year by 11 minute and 14 second → A new extra day
        after 128 calendar years in addition to February 29, which is why the
        leap year is canceled once every 400 years. */
    cout<<"Day " <<testDay<<" invalid. Set to day 1.\n";
    return 1; // leave object in consistent state if bad value
} // end function checkDay ret

```

توابع الملف الرأسي Employee.h

```
// header Date.h
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include "date.h"
class Employee{
public:
    Employee(const char *,const char *,const Date &,const Date &) ;
    void print() const;
    ~Employee(); // provided to confirm destruction order
private:
    char firstName[ 25 ];      char lastName[ 25 ];
    const Date birthDate; // composition: member object
    const Date hireDate; // composition: member object
}; // end class Employee
#endif
```

توابع الملف الرأسي Employee.cpp

```
// employemain1F.cpp
```

```
#include "employee.h" // Employee class definition
#include <iostream>
#include <cstring>
#include "date.h" // Date class definition
using namespace std;
Employee::Employee(const char *first, const char *last, const Date
&dateOfBirth, const Date &dateOfHire)
: birthDate( dateOfBirth), hireDate( dateOfHire) //
{ int length = strlen( first) ;
length = (length < 25 ? length: 24) ;
strncpy( firstName, first, length) ;
firstName[ length ] = '\0' ;
length = strlen( last) ;
```

## توابع الملف الرأسي Employee.cpp

```
length = (length < 25 ? length: 24) ;
    strncpy( lastName, last, length) ;
    lastName[ length ] = '\0';
    cout << "Employee object constructor: "
         << firstName << ' ' << lastName << endl;
} // end Employee constructor
void Employee::print() const
{   cout <<lastName<< ", " <<firstName<<"\nHired: ";
    hireDate.print(); //
    cout << " Birth date: "; birthDate.print(); cout << endl;
} // end function print
Employee::~Employee()
{   cout <<"Employee object destructor: "<<lastName<<", " <<firstName<<endl;
} // end ~Employee destructor
```

الملف يتضمن التابع الرئيس `compositEmployeeMain1.cpp`.

```
// compositEmployeeMain1.cpp : main project file.
```

```
#include <iostream>
#include "employee.h" // Employee class definition
using namespace std;
int main()
{
    Date birth( 7, 24, 1989) ;
    Date hire( 3, 12, 2009) ;
    Employee manager( "Salim", "Salam", birth, hire) ;
    cout << '\n';
    manager.print();
    cout<<"\nTest Date constructor with invalid values:\n";
    Date lastDayOff( 14, 35, 1994) ; // invalid month and day
    cout << endl;
    system ("pause");return 0;
} // end main
```

شرح الملف التابع الرئيس `compositEmployeeMain1.cpp`.

- تضمن تعريف الصنف `Date` تابعاً يدعى `checkDay` ويهدف إلى اختبار عدد أيام الشهر إن كانت سليمة بالإضافة إلى مناقشة حالة السنة الكبيسة `leap year`، وتم استدعاء هذا التابع ضمن باني الصنف بحيث تصبح عملية البناء متضمنة لعملية اختبار القيم الابتدائية المرسله إن كانت تعبر عن قيمة صالحة للتاريخ أم لا.
- تضمن تعريف الصنف تابعاً هادماً يهدف اختبار ترتيب هدم غرض من الصنف `Employee` (بمعنى آخر هل يتم هدم الأغراض المستخدمة لتعريف البيانات الأعضاء أولاً أم يتم هدم الغرض من الصنف `Employee` أولاً؟).  
ملاحظات حول المثال السابق:
- ✓ التعريف المبين في المثال السابق للأصناف `Date`، `Employee` والتوابع المضمنة هو صيغة مقترحة، إلا أن ذلك لا يمنع أن يتم تعريفهما بطرق أخرى مرغوب وتضمنيهما التوابع المرغوبة.
- ✓ بملاحظة تعريف التابع الباني للصنف `Employee` نجد أن إعطاء قيم للعضوين `firstName` و `lastName` يكون من خلال تمرير القيم، أما العضوين `birthDate` و `hireDate` فيكون إعطاؤهما القيم من خلال التابع الباني للصنف `Date`. يتم هذا الأمر من خلال التصريح عن التابع الباني للصنف `Employee` وتستخدم النقطتان (: ) للفصل بين قائمة وسطاء التابع الباني للصنف `Employee` واستدعاء التابع الباني للصنف `Date`.
- ✓ نلاحظ أنه ضمن تعريف التابع `print()` للصنف `Employee` كيفية استدعاء التابع `print()` للصنف `Date`.
- ✓ يبين خرج البرنامج أن بناء الأغراض يجري من الداخل إلى الخارج بينما يجري هدمها من الخارج إلى الداخل (أي يجري هدم الغرض من النمط `Employee` ومن ثم يجري الأغراض من النمط `Date` المعرفة ضمنه).

- يمكن استخدام الأغراض مع التوابع سواء كبارامترات للتوابع أو كقيمة معادة منها. كما يمكن أن يكون هذا الاستخدام ضمن التوابع الأعضاء للصنف، أو ضمن توابع أخرى خارج نطاق الصنف (لكن في هذا الحالة يجب مراعاة محددات الوصول ومجالات الرؤية).
- يتم تمرير الأغراض إلى التوابع بنفس الطريقة التي يتم فيها تمرير أي متحول من نوع بيانات آخر. حيث يتم تمرير الأغراض إلى التوابع من خلال أسلوب الاستدعاء بالقيمة call by value وعندئذ يتم إنشاء نسخة من الغرض عند تمريرها للتابع (بمعنى آخر يتم إنشاء غرض آخر)، وهذا يطرح السؤال التالي: هل يتم تنفيذ التابع الباني عند إنشاء النسخة والتابع الهادم عند تدميرها؟
- نختبر تمرير الأغراض إلى التوابع من خلال المثال التالي الذي يستخدم نسخة معدلة من الصنف Time كما يلي:

// UsingObjectsWithinFunctions62.cpp : main project file.

```
#include <iostream>
```

```
using namespace std;
```

```
class Time {public:    Time(int, int, int, char ) ;           //constructor
```

```
    ~Time();           //destructor
```

```
    inline int get_second() { return second;}
```

```
    inline int get_minute() { return minute;}
```

```
    inline int get_hour()   { return hour;}
```

```
    inline char get_sep()   { return sep;}
```

Using objects within functions

```
private:
    int hour;           // 0 - 23 (24-hour clock format)
    int minute;        // 0 - 59
    int second;        // 0 - 59
    char sep;          // :
}; // end clas Time
Time::Time(int hr,int mn,int sc, char sp)
{ cout<<"constructing object "<<hr<<":"<<mn<<":"<<sc<<endl;   hour=hr;
minute=mn;   second=sc;           sep= sp;
} // end Time constructor
Time::~~Time(){cout<<"destructing object "<<hour<<sep <<minute<<sep
<<second <<endl;} // end Time destructor
int time_to_seconds(Time T) //Independent function
{ return T.get_second()+(T.get_minute()*60)+(T.get_hour()*3600);}
int main(){ {Time timeObj(6,8,24,':'); cout<<time_to_seconds(timeObj)<<endl;}
system("pause"); return 0;} // end main
```

### Using objects within functions

بتنفيذ هذا البرنامج يعطي على خرجه:

```
constructing object 6:8:24
destructing object 6:8:24
22104
destructing object 6:8:24
Press any key to continue . . .
```

بملاحظة الخرج وتحليل البرنامج السابق نجد:

- تم تمرير غرض من النمط Time إلى التابع المعرف خارج الصنف time\_to\_seconds() الذي يقوم بتحويل الزمن إلى ثواني، وتم استخدام ثلاثة توابع سطرية معرفة ضمن الصنف للوصول إلى البيانات الأعضاء الخاصة.
- لاحظ أنه قد تم استدعاء التابع الهادم للصنف مرتين، بينما تم استدعاء التابع الباني مرة واحدة. مما يعني أن التابع الباني لا يتم استدعاؤه عند إنشاء نسخة من الغرض لتميرها كبارامتر للتابع والسبب في ذلك أنه لو تم استدعاء التابع الباني فإن الغرض ستم تهيئته بقيم ابتدائية غير قيمته الحالية وبالتالي سيتم تغييره. بينما يتم استدعاء التابع الهادم عند هدم النسخة الممررة.
- كان بالإمكان تعريف هذا التابع كتابع عضو ضمن الصنف وعندها كان بالإمكان الاستغناء عن تعريف توابع وصول إلى البيانات الأعضاء الخاصة لأنها تقم ضمن مجال رؤية هذا التابع. يكون شكل التابع وعملية الاستدعاء في هذه الحالة:

```
int Time::time_to_seconds(Time T)
{   return T.second+(T.minute*60) + (T.hour*3600) ; }
cout<<timeObj.time_to_seconds(timeObj)<<endl ;
```

تختلف عملية إعادة غرض من تابع قليلاً بين حالة كون التابع عضواً في الصنف أو لا، إذ يمكن في حال كونه عضواً أن يتم استخدام المؤشر `*this` لإعادة الغرض، بينما سيتم استخدام أسلوب النسخ بالإسناد في حال كون التابع غير عضو في الصنف (طبعاً هناك بعض الطرق الأخرى التي يمكن أن تكون أكثر أماناً كاستخدام الباني النسخ الذي تعرفنا عليه سابقاً أو استخدام التحميل الزائد لمعامل الإسناد. نختبر هذا الأمر من خلال برنامج يتضمن تابعين يقومان بجمع قيم غرضين من النمط `Time` كلاهما عضوي في الصنف.

**// UsingObjectsWithinFunctionsThis64.cpp : main project file.**

```
#include <iostream>
#include <iomanip>
using namespace std;
class Time {
public:
    Time(int, int, int, char) ;           //constructor
    Time add_times1(Time) ;
    Time add_times2(Time, Time);
    void printUniversal();
```

```

private:
    int hour;           // 0 - 23 (24-hour clock format)
    int minute;        // 0 - 59
    int second;        // 0 - 59
    char sep;          // :
}; // end clas Time
Time::Time(int hr,int mn,int sc, char sp)
{    hour=hr;    minute=mn;    second=sc;    sep=sp;} // end Time constructor
Time Time::add_times1(Time T) {int t=0,m=0;second=second+T.second;
if (second>59){t=second/60; second=second%60;}minute=minute+T.minute+t;
if (minute>59){m=minute/60; minute=minute%60;}
hour=(m+hour+T.hour)>23?(m+hour+T.hour)%24:(m+hour+T.hour);return *this;}
Time Time::add_times2(Time T1,Time T2)
{int t=0; t=T1.second+T2.second; second=t>59?t%60:t;
t=(t/60)+T1.minute+T2.minute; minute=t>59?t%60:t;t=(t/60)+T1.hour+T2.hour;
hour=t>23?t%24:t;return *this;}

```

- المؤشر `this`: هو مؤشر ذاتي يؤشر على نفس الغرض المشتق، ويؤمن عملية الرجوع السليمة عندما يقوم تابع عضو بالرجوع إلى عضو آخر ضمن الصنف.
- يعتبر المؤشر `this` ضمناً كافة الأعضاء ضمن الغرض على أنها عناصر تابع له، إلا أنه يمكن استخدامه بشكل صريح.
- يستطيع كل غرض أن يحدد عنوانه الذاتي باستخدام المؤشر `this` كما يمكن أن يستخدم من أجل الرجوع إلى المعطيات والتوابع الأعضاء، ويتعلق نمطه بنمط الغرض المرتبط به.
- يمكن أن يكون التابع العضو الذي نستخدم ضمنه المؤشر `this` من النوع `const` أولاً، وعند استخدام تابع عضو غير ثابت `A` يأخذ المؤشر `this` الصيغة:  
`A * const`
- أما عندما يكون التابع العضو ثابتاً يأخذ المؤشر `this` الصيغة:  
`const A * const`
- لدى استدعاء تابع عضو، يتم تلقائياً تمرير وسيط ضماني هو عبارة عن مؤشر إلى الغرض المستدعي، هذا المؤشر يدعى المؤشر `this`.  
لكي نفهم الاستخدام الضمني والصريح لهذا المؤشر، لناخذ الصيغة المعدلة التالية للصنف `Time`:

```
// ImplicitExplicitThisPointer67.cpp : main project file.
```

```
#include <iostream>  
using namespace std;
```

```

class Time {
public:
    Time(int, int, int, char) ;    //constructor
    void printUniversal();    // output universal-time format
private:
    int hour;    // 0 - 23 (24-hour clock format)
    int minute;    // 0 - 59
    int second;    // 0 - 59
    char sep;}; // end clas Time
Time::Time(int hr,int mn,int sc,char sp)
{ hour=hr; minute=mn; second=sc; sep=sp; } // end Time constructor
void Time::printUniversal()
{ /*implicte use of this pointer*/ cout<<hour<<sep<<minute<<sep<< second<<endl;
  /*explicte use of this pointer*/ cout<<this->hour<<this->sep<<this->minute
  <<this->sep<<this->second<<endl; cout<<(*this).hour<<(*this).sep<<(*this).minute
  <<(*this).sep <<(*this).second<<endl;} // end function printUniversal

```

Use the reference as a parameter as a return value

```
int main()
{ Time timeObj(10,14,24, ':');
  timeObj.printUniversal();
system("pause");    return 0;
} // end main
```

10:14:24

10:14:24

10:14:24

Press any key to continue . . .



## استخدام المؤشر كبارامتر ومرجع وقيمه معادة 1

يعطي المثال السابق على خرجه:

أما فيما يتعلق بالمراجع references، فهناك ثلاث طرق لاستخدامها:

1- كبارامتر للتابع.

2- كقيمة معادة من التابع.

3- كمرجع بحد ذاته.

تحقيق هذه الاستخدامات من أجل أنماط معرفة من قبل المستخدم. لتوضيح ذلك نبين الاستخدامات الثلاثة في مثال واحد يستخدم نسخة معدلة من الصنف Time كما يلي:

Use the reference as a parameter as a return value

// thisAsReferenceAsParameterReturnValue.cpp : main project file.

```
#include <iostream>
using namespace std;
class Time {
public:
    Time(int, int, int, char) ; //constructor
    void print(); // print time object
    void time_modify(Time &); // using reference as parameter
    Time &time_modify1(int) ; // returning reference
private:
    int hour; // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
    char sep;
}; // end clas Time
```

Use the reference as a parameter as a return value



استخدام المؤشر كبارامتر ومرجع وقيمه معادة 3

```
Time::Time(int hr,int mn,int sc,char sp)
{ hour=hr; minute=mn; second=sc;          sep= sp; } // end Time constructor

void Time::print()
{cout<<hour<<sep<<minute<<sep<<second<<endl;}//end fun print

void Time::time_modify(Time &T)
{   hour=hour-T.hour; minute=minute-T.minute; second=second-T.second; }
// end function time_modify All fields timeObj1 must be greater than timeObj2

Time &Time::time_modify1(int x)
{ hour=hour-x;
minute=minute+x; second=second/x; return *this; }// end time_modify1

int main() {Time timeObj1(6,8,24,':'),timeObj2(4,2,20,':');
cout<<"NORMAL USE OF OBJECTS "<<endl;

timeObj1.print();    timeObj2.print();
cout<<"USING REFERENCES AS PARAMETER "<<endl;
```

Use the reference as a parameter as a  
return value

استخدام المؤشر كبارامتر ومرجع وقيمة معادة 4

```
timeObj1.time_modify(timeObj2);  
timeObj1.print();    timeObj2.print();  
cout<<"USING REFERENCES AS RETURNED VALUE "<<endl;  
  
timeObj1=timeObj1.time_modify1(2);  
timeObj1.print();    timeObj2.print();    system("pause"); return 0;  
} // end main
```

NORMAL USE OF OBJECTS

6:8:24

4:2:20

USING REFERENCES AS PARAMETER

2:6:4

4:2:20

USING REFERENCES AS RETURNED VALUE

0:8:2

4:2:20

Press any key to continue . . .

- تحتوي لغة C++ على معاملي تخصيص ديناميكي هما: `new` و `delete`. تستخدم هذه المعاملات لتخصيص وتحرير الذاكرة وقت التنفيذ (يمكن استخدام هذين المعاملين مع الأنماط مسبقة التعريف والأنماط المعرفة من قبل المستخدم).
- يقوم المعامل `new` بتخصيص الذاكرة وإعادة مؤشر إلى بداية موقع الذاكرة المحجوز، ويقوم المعامل `delete` بتحرير الذاكرة المخصصة باستخدام `new`. الشكل العام لاستخدام هذه المعاملات:

```
p_var = new type; delete p_var;
```

- التصريح، `p_var` هو متحول مؤشريتلقى مؤشراً إلى الذاكرة المحجوزة لاحتواء قيمة من النمط `type`.  
كمثال:

```
Time *timePtr=new Time;
```

- تقوم التعليمة السابقة بإنشاء غرض ذو حجم مناسب وتقوم أيضاً باستدعاء تابع البناء الافتراضي الخاص بالغرض وتعيد بعدها مؤشراً له من نفس نمط الغرض. لتحرير الذاكرة المحجوزة من قبل الغرض، يمكن استخدام المعامل `delete`:

```
delete timePtr;
```

- تقوم هذه التعليمة باستدعاء التابع الهادم للغرض المؤشر إليها باستخدام الغرض `timePtr` ومن ثم يتم تحرير الذاكرة المرتبطة به.
- بما أن ذاكرة الكومة `heap` محدودة الحجم، فإنها قد تستنفذ. إذا لم يكن هناك ذاكرة حرة كافية لتحقيق التخصيص المطلوب فإن المعامل `new` سيفشل، وسيتم توليد الاستثناء `bad alloc` وهذا الاستثناء معرف في الملف الرأسي `<new>` يجب أن يقوم البرنامج بمعالجة هذا الاستثناء والقيام بالاستجابة الملائمة في حال حصوله (ندرس في فصل لاحق معالجة الاستثناءات)، فإن لم تتم معالجة الاستثناء في البرنامج فإن البرنامج سيتم إنهاؤه. المثال التالي يستخدم المعامل `new` لتخصيص الذاكرة لاحتواء غرض من النمط `Time`:

```
// DynamicMemoryAllocation71.cpp : main project file.
```

```
#include <iostream>
#include <iomanip>
using namespace std;
class Time {
public:    Time();                //constructor
        void setTime(int hr,int mn,int sc,char sp);
        void printUniversal(); // output universal-time format
        void printStandard(); // output standard-time format
        ~Time();
private: int hour;              // 0 - 23 (24-hour clock format)
        int minute;            // 0 - 59
        int second;           // 0 - 59
        char sep;             // :
}; // end clas Time
```

Dynamic memory allocation

```
Time::Time() { cout<<"CLASS TIME CONSTRUCTOR STARTD "<<endl;
    hour = minute = second = 0; sep= ':'; } // end Time constructor

void Time::setTime(int hr,int mn,int sc,char sp)
{hour=hr; minute=mn; second=sc;      sep=sp; }

void Time::printUniversal()
{  cout << setfill( '0' ) << setw( 2) << hour << sep << setw( 2) << minute <<sep
    << setw( 2) << second;} // end function printUniversal
void Time::printStandard()
{  cout<<((hour==0||hour==12)?12:hour%12)<<sep<<setfill('0') <<setw( 2) <<minute
    << sep <<setw( 2) <<second<<(hour<12 ? " AM": " PM" ) ;
} // end function printStandard

Time::~~Time()
{  cout<<"CLASS TIME DESTRUCTOR STARTD "<<endl; } // end Time destructor
```

## Dynamic memory allocation



## التخصيص الديناميكي للذاكرة 4

```
int main()
{   Time *timePtr=new Time;
    timePtr->printStandard();
    cout<<endl;
timePtr->printUniversal();
    cout<<endl;
    delete timePtr;
    system ("pause"); return 0;
} // end main
```

```
// allocate Time object
```

```
//deallocate time object
```

```
CLASS TIME CONSTRUCTOR STARTD
12:00:00 AM
00:00:00
CLASS TIME DESTRUCTOR STARTD
Press any key to continue . . .
```

يعطي هذا البرنامج على خرجه:

ملاحظات:

- نقوم في كل مرة نستخدم فيها الصنف Time بإجراء بعض التعديلات بما يخدم الفكرة التي نقوم بمناقشتها. واختصار بعض التوابع تجنباً لطول مقاطع الشيفرة.
  - بما أن الغرض timePtr هو مؤشر لذا كان من الواجب استخدام معامل السهم (->) arrow operator للوصول إلى أعضائه.
  - يجب أن يتم استخدام المعامل delete فقط مع المؤشرات الصالحة التي تم تخصيصها باستخدام new.
- يمكن تهيئة الذاكرة المخصصة ببعض القيم ابتدائية initializer بعد اسم النمط في العبارة new. الشكل العام لهذا الاستخدام كما يلي:
- ```
p_var = new var_type (initializer);
```

بالطبع نوع القيمة الابتدائية يجب أن يكون متوافقاً مع نوع البيانات الذي خصصت الذاكرة من أجله.

يمكن تعديل المقطع البرمجي السابق وكتابة تعليمة التصريح كما يلي: `Time *timePtr=new Time(14,33,35,');`

ثم إعادة تنفيذ البرنامج، حيث يظهر الفرق؟.

يمكن تخصيص المصفوفات بواسطة المعامل new باستخدام الصيغة العامة التالية: `p_var = new array_type [size];`

حيث size يحدد عدد العناصر في المصفوفة.

لتحرير مصفوفة يمكن استخدام delete على النحو: `delete [] p_var;`

حيث إن استخدام [] تخبر delete أن المراد تحريره هو مصفوفة وعند غياب [] سيتم حذف عنوان بداية المصفوفة وليس المصفوفة.

هناك أمر وحيد يجب أن تتم مراعاته لدى تخصيص المصفوفات: وهو أنها لا يمكن أن تتم تهيئتها. أي لا يمكن تحديد لائحة تهيئة عند تخصيص مصفوفة.

يمكن تخصيص مصفوفات الأغراض، إلا أن هناك معضلة، بما أنه لا يمكن لمصفوفة مخصصة ديناميكياً بالمعامل `new` أن تهيأ بالقيم الابتدائية، فإنه يجب التحقق من كون الصنف يحوي تابع باني، وأنه بلا بارامترات. فإن لم يكن ذلك محققاً فإن مترجم لغة C++ لن يجد بانياً مناسباً عندما تحاول تخصيص المصفوفة ولن يقوم بترجمة البرنامج. المثال التالي يوضح هذا الاستخدام:

```
int main(){    Time *timePtr=new Time[3];    // create array of 3 Time object
    timePtr[0].setTime(10,15,20, ':');    timePtr[1].setTime(22,7,55, ':');
    timePtr[2].setTime(2,54,34, ':');
    for (int i=0;i<3;i++){timePtr[i].printStandard();        cout<<endl;
        timePtr[i].printUniversal();        cout<<endl;}
    delete [] timePtr;    system ("pause");    return 0;} // end main
```

```
CLASS TIME CONSTRUCTOR STARTD
CLASS TIME CONSTRUCTOR STARTD
CLASS TIME CONSTRUCTOR STARTD
10:15:20 AM
10:15:20
10:07:55 PM
22:07:55
```

```
2:54:34 AM
02:54:34
CLASS TIME DESTRUCTOR STARTD
CLASS TIME DESTRUCTOR STARTD
CLASS TIME DESTRUCTOR STARTD
Press any key to continue . . .
```

الصنف الخازن Container class : هو عبارة عن أحد أنماط الأصناف ، وهو مصمم لاستيعاب مجموعة من الأغراض، وتقديم الخدمات الملائمة لمثل هذه العملية كالإدراج، الحذف، البحث، ... وغيرها.

من أشهر أشكال الأصناف الخازنة المصفوفات arrays، المكدسات stacks، الأرتال queues، واللوائح المترابطة linked lists. غالباً ما يرتبط الصنف الخازن بالأغراض التكرارية ( المكررات )، حيث أن المكرر هو عبارة عن غرض يعيد العنصر التالي ضمن مجموعة. ويعرف عادة على شكل تابع صديق للصنف الخازن. ويمكن لصنف خازن أن يكون له أكثر من مكرر في نفس الوقت.

الصنف الوكيل Proxy class : هو عبارة عن أحد الوسائل المستخدمة في إخفاء تفاصيل بناء صنف لمنع الوصول إلى المعلومات المرتبطة به ومنع الوصول إلى طريقة برمجة الصنف.

إن التعمق في مفهومي الأصناف الخازنة والأصناف الوكييلة خارج نطاق اهتمامنا في هذا المساق،

انتهت تمارين الأسبوع الثاني

### Utility classes and static function

```
static int count;
int Student::count=0;
```

عند الحاجة لمتغير على مستوى الصنف يتم تعريف هذا المتغير على أنه static ويكون التعريف ضمن الصنف مثل ضمن ملف التحقيق يتم اسناد قيمة ابتدائية بدون ذكر static مع الربط باسم الصنف وليس ضمن الصنف.

```
int student::getcounter() {return counter}
```

يتم زيادة العداد ضمن كل البواني count++;

طبعاً لأبد من كتابع التابع الذي سيعيد قيمة العداد

```
Student s1,s2,s3; cout<<s1.getcount()<<endl;
```

ضمن ملف الاختبار main نكتب

سنحصل على الرقم 3 السبب مكان وجود أمر الطباعة.

عند الحاجة لتنفيذ تابع بدون اشتقاق كائن من الصنف نعرق التوابع على أنها static ويتم نداء هذه التوابع من خلال اسم الصنف معامل ارتباط :: واسم التابع، أي ان التابع يتبع للصنف وليس للكائنات، لن يتم تخزين المعطيات.

```
static float add(float x, float y);
```

ضمن ملف header نعرف التوابع static مثل:

```
float Calc::add(float x, float y) {return x+y ;}
```

يتم كتابة التابع وجسمه ضمن ملف التوابع .

وببساطة نكتب ضمن التابع الرئيس تابع اختبار الصنف:

```
cout<<Calc::add(4,5)<<endl;
```

سيكون الخرج 9, ملاحظة: الأعضاء هي معرفة طالما أن البرنامج يعمل أي على مستوى الملف حتي لو انتهت حياة الكائن لأنها تتبع الصنف وليس الكائنات.

## Structures

## السجلات

- السجل: مجموعة مرتبة من العناصر، قد تكون من أنواع مختلفة، وتعرف بعناصر السجل واحياناً أعضاء السجل members أو حقول السجل fields. ويمكن الوصول إلى كل عنصر من السجل بشكل مباشر وبالتالي يمكن التعامل مع القيم أو تخزينها في هذا العنصر.
- يختلف السجل عن المصفوفة كون عناصر المصفوفة يجب أن تكون من النوع نفسه.
- تعريف السجل: يمكن أن يتم تعريف السجل بالصيغة العامة المبسطة التالية:

```
struct structName  
{   Declarations of members   };
```

- يبتدئ تعريف السجل بالكلمة المفتاحية struct متبوعة باسم السجل (والذي يمكن أن يكون أي اسم يختاره المبرمج وفقاً لقواعد تسمية المعارف التي تعرفنا عليها في مساقات سابقة).
- تعريف عناصر السجل بين قوسي بداية ونهاية كتلة برمجية حيث يمكن للعناصر أن تكون من أي نوع.
- ينتهي تعريف السجل بفاصلة منقوطة.
- حيث يتم وضع هذا التعريف ضمن قسم التضمينات للبرنامج. كما يمكن تعريف السجل بصيغة أكثر تعقيداً وتركيباً كما يلي:

```
struct structName  
{ private:   Declarations of private members  
  public:   Declarations of public members   };
```

## Structures

- الكلمتان المفتاحيتان `public` و `private` محددتا وصول `access specifiers` يعلنان أن ما هو معرف بعدهما يمكن الوصول إليه من قبل أجزاء الغرض نفسه (`private`) أو من قبل جميع أجزاء البرنامج (`public`) على الترتيب.
- بعد تعريف السجل فإنه يصبح نوع بيانات ومن الممكن البدء باستخدامه من خلال التصريح عن متحولات من هذا النوع، حيث يأخذ التصريح الشكل العام التالي:

1. `structName objStruct_name;`
2. `structName objStruct_name={initializer_list};`

في الشكل الأول، تم التصريح عن متحول يدعى `objStruct_name` من النمط الذي قمنا بتعريفه حيث يطلب إلى المنقح أن يقوم بحجز قطاع من مواقع الذاكرة لتخزين الأغراض من الأنماط المحددة وإطلاق الاسم `objStruct_name` على هذا القطاع.

في الشكل الثاني، تكون أنماط القيم المحددة في لائحة القيم الابتدائية `initializer_list` متوافقة مع أنماط الأعضاء المقابلة لها في التصريح عن السجل وتستخدم لإعطاء قيم ابتدائية لعناصر البيانات للمتحول من نوع السجل.

ملاحظة هامة: يمكن لأعضاء السجل أن تكون عناصر بيانات أو توابع خاصة بالتعامل مع عناصر البيانات تلك، إلا أننا سنقتصر في عرضنا المقتضب هذا لموضوع السجلات على تعريف سجلات أعضاؤها هي عناصر بيانات فقط، كما أننا سنقتصر على تعريف سجلات بدون استخدام محددات الوصول حيث يكون محدد الوصول الافتراضي في هذه الحالة هو `public` أي يمكن الوصول لأعضاء السجل من قبل كامل أجزاء البرنامج، وسيصار عند الانتقال لموضوع الأصناف التوسع أكثر في هذا الأمر.

## أمثلة عن السجلات 1

### Examples of struct

يمكن كما سبق وذكرنا أن يتم إنشاء متحول من النوع Time مع إسناد قيم ابتدائية لعناصره باستخدام الشكل التالي:

```
struct Time
{   int Hour;           //0-23
    int Minute;        //0-59
    int second;        //0-59
};
Time lunch={14,30,00,' : '};
```

|       | Hour | Minute | second | Sep |
|-------|------|--------|--------|-----|
| Lunch | 14   | 30     | 00     | :   |

حيث تعطى القيم الابتدائية للعناصر كما سبق:

تعريف النمط Date باستخدام السجلات:

بشكل مشابه، يمكن تعريف النمط Date الذي يستخدم للتعامل مع بيانات التاريخ (اليوم، الشهر، السنة) كما يلي:

```
struct Date
{   int day;           //1-31
    int month;        //1-12
    int year;         //any valid year
};
Date d;
Date birthDate = {17,4,2005};
```

كما في الأمثلة: Date وبالمثل، يمكن التصريح عن متحويلات من النمط

## أمثلة عن السجلات 2

### Examples of struct

تعريف النمط Employee باستخدام السجلات: النمط Employee الذي يستطيع التعامل مع بيانات موظف في شركة أو مؤسسة، ويمتلك الموظف الصفات التالية: الرقم الذاتي، الاسم الأول، الاسم الأخير، القسم. يمكن استخدام السجلات لتحقيق هذا التعريف:

```
struct Employee
{
    int ID;
    char firstName[25];
    char lastName[25];
    char department[100];
};
```

وبالمثل، يمكن التصريح عن متحولات (كائنات) من النمط Employee كما يلي:

```
Employee manager1;
Employee manager={2518, "Ahmad", "Ali", "Programming"};
```

يمكن تمثيل الموظف manager في التصريح الثاني كما يلي:

|         | ID   | firstName | lastName | department  |
|---------|------|-----------|----------|-------------|
| Manager | 2518 | Ahmad     | Ali      | Programming |

## Examples of struct

## أمثلة عن السجلات 3

تعريف النمط Point باستخدام السجلات:

نختتم أمثلتنا بتعريف النمط Point الذي يمكن أن يستخدم للتعامل مع نقطة في جملة الإحداثيات الديكارتية، حيث تعرف النقطة من خلال إحداثياتها على محوري السينات والعيونات.

```
struct Point
{   double x;   double y;   };
```

وكأمثلة للتصريح عن متحولات من النمط Point:

```
Point c; Point center={0,0};
```

لتعريف السجل لا بحجز أي حيز ضمن الذاكرة، فهو ليس سوى تعريف لنمط بيانات جديد يمكن أن يستخدم للتصريح عن كائنات بنفس الطريقة المستخدمة للتصريح عن متحولات من أنماط أخرى، فمثلاً:

```
Point p;
Point pA[10];
Point *pptr=&p;
Point &pref=p;
```

تعريف كائن p من النمط Point

تعريف مصفوفة اسمها pA من 10 كائنات من النمط Point

تعريف مؤشر الى كائن pptr يحوي عنوان الكائن p من النمط Point

تعريف مرجع اسمه pref للكائن p من النمط Point

## Accessing Struct fields

- يتم الوصول إلى حقول سجل لمعرفة قيمها أو تعديلها ويكون باستخدام معاملات الوصول إلى الحقول Access operators وهي المعامل نقطة (.).
- dot operator الذي يستخدم مع التعريف العادي للمتحويلات أو التعريف بالمرجع،
- ويستخدم معامل السهم arrow operator (->) في حال استخدام المؤشرات في التعريف، ويمكن الاستغناء عن السهم واستخدام النقطة مع مؤشر إلى حقل السجل حيث يوضع بين قوسين كما يلي (\*fildName).
- تفيد هذه المعاملات في الوصول إلى حقول السجل باستخدام اسم الكائن الذي يعبر عن ذلك السجل من خلال الصيغة العامة التالية:

```
Struct_object.member_name;
```

```
Struct_object->member_name; //or Struct_object.(*member_name);
```

فمثلاً: لإدخال قيمة للعنصر hour للكائن lunch من النوع Time يمكن أن نكتب:

لعرض قيمة العنصر lastName لكائن manager من النمط Employee ونكتب:

- لاختبار قيمة العنصر month للكائن d من النمط Date فيما إذا كانت صالحة والمشتق بالمؤشر يمكن كتابة المقطع التالي:

```
if (d->month>12 || d->month<1) cout<<"Not a valid month";
```

أو

```
if (d.(*month)>12 || d.(*month)<1) cout<<"Not a valid month";
```

كما يجب التنويه بأهمية وجود \* ما بين القوسين لتحقيق اولوية التنفيذ.

## Structs in memory



## تخزين السجلات في الذاكرة

• الوصول إلى عنصر من عناصر المصفوفة يتم على مرحلتين:

✓ الأولى معرفة عنوان المصفوفة ضمن الذاكرة،

✓ الثانية إجراء انزياح ضمن الذاكرة بمقدار حجم النوع مضروب بعدد العناصر السابقه لهذا العنصر.

✓ بشكل مشابه تماماً، يتم تخزين السجلات ضمن الذاكرة ولكن مع فارق أن عملية الوصول إلى عنصر ما ضمن السجل قد يكون أكثر تعقيداً لأن عناصر السجل قد تكون من أنواع مختلفة وبالتالي تحتاج إلى حجوم مختلفة ضمن الذاكرة.

✓ لتوضيح هذه العملية، ليكن لدينا السجل Employee الذي قمنا بتعريفه في الفقرة السابقة، وقمنا بالتصريح عن كائن من نوع هذا السجل كما يلي:

Employee manger;

بفرض أن القيم int تحتاج إلى 4bytes لكي تخزن، القيم char تحتاج إلى 1byte. إن التصريح عن السجل manger يطلب من المنقح أن يقوم بحجز قطاع من الذاكرة حجمه 154bytes لتخزين قيم هذا السجل، أول عنوان في هذا القطاع يدعى عنوان الأساس base address، البايتات الأربعة بدءاً من العنوان الأساس مخصصة للعضو manger.ID، البايتات الخمس والعشرون التالية مخصصة للعضو manger.firstName... وهكذا.

## Structs within structs



## تعريف السجلات ضمن السجلات

بما أن أعضاء السجل يمكن أن تكون من أي نمط، فإنها ليس بالضرورة أن تكون من أنماط بسيطة، وإنما يمكن أن تكون من أنماط بيانات بنيوية كالمصفوفات أو السجلات.

يمكن تطوير التعريف السابق للسجل Employee بإضافة العنصرين: تاريخ الميلاد، تاريخ التعيين كما يلي:

```
struct Employee
{
    int ID;
    char firstName[25];
    char lastName[25];
    char department[100];
    Date birth;
    Date hire;
};
```

طبعاً يجب أن يكون السجل Date قد تم تعريفه قبل أن يتم استخدامه.

في هذه الحالة يمكن الوصول إلى عناصر السجل من النوع Date لعرض قيمتها على سبيل المثال، كما يلي:

```
Employee manager;    cout<<manager.birth.month;
```

تدعى السجلات التي تحتوي في تعريفها على سجلات أخرى باسم السجلات المتداخلة `nested structures` وتدعى أحياناً السجلات الشجرية `.hierarchical structures`.

سنقوم بتعريف تابعين يستخدمان النمط Time، الأول printUniversal لطباعة الوقت بالصيغة العسكرية والثاني printStandard لطباعة الوقت بالصيغة المعيارية.

لاحظ أن تابعي الطباعة يأخذان كوسيط عناوين البنية السجل Time على شكل ثوابت مما يؤدي إلى تمرير هذه السجلات إلى تابع الطباعة من خلال عناوينها (دون القيام بنسخها كما هو الحال عند تمريرها بالقيمة) ثم يستخدم هذان التابعان الواصف const لمنع تعديل القيم المرتبطة بهذا السجل.

```
#include <iostream>
#include <iomanip>
using namespace std;
// structure definition
struct Time {
    int hour;        // 0-23 (24-hour clock format)
    int minute;     // 0-59
    int second;     // 0-59
    char sep;       // :
}; // end struct Time
```

```

void printUniversal( const Time & ) ; // prototype
void printStandard( const Time & ) ; // prototype

int main()
{
    Time dinnerTime; // variable of new type Time
    dinnerTime.hour = 18; // set hour member of dinnerTime
    dinnerTime.minute = 30; // set minute member of dinnerTime
    dinnerTime.second = 0; // set second member of dinnerTime
    dinnerTime.sep = ':';

    cout << "Dinner will be held at "; printUniversal( dinnerTime ) ;
    cout << " universal time,\nwhich is "; printStandard( dinnerTime ) ;
    cout << " standard time.\n";

    dinnerTime.hour = 29; // set hour to invalid value
    dinnerTime.minute = 73; // set minute to invalid value

```

نموذج التابع printUniversal ←

نموذج التابع printStandard ←

تعريف كائن dinnerTime ←

اسناد قيم الحقول

```

cout << "\nTime with invalid values: ";    printUniversal( dinnerTime) ;
    cout << endl;    system ("pause");    return 0;
} // end main

// print time in universal-time format
void printUniversal( const Time &t)
{    cout << setfill( '0') << setw( 2) << t.hour <<t.sep
    << setw( 2) << t.minute <<t.sep<< setw( 2) << t.second;
} // end function printUniversal

// print time in standard-time format
void printStandard( const Time &t)
{    cout << (( t.hour == 0 || t.hour == 12) ?
    12: t.hour % 12) <<t.sep << setfill( '0')
    << setw( 2) << t.minute <<t.sep<< setw( 2) << t.second
    << (t.hour < 12 ? " AM": " PM") ;
} // end function printStandard

```

← كود تابع printUniversal

← كود تابع printStandard