المركز الوطني للمتميزين
National Centre for the Distinguished

الصفوف في C++

Class in C++

تقديم: محمود قره فلاح

بتاريخ: 3 – 1 – 2015

الفهرس

فهرس الصور

مقدمة

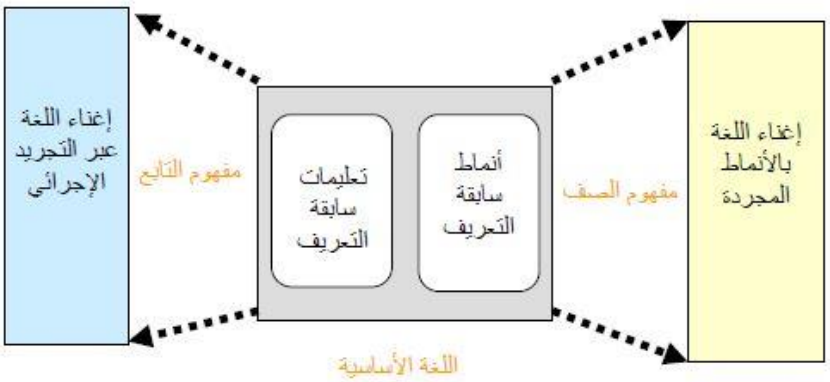
بلا أي شك، جميعنا على دراية بجميع أنواع المتحولات الموجودة في لغة C++ (char, int,.....) والموجودة في أغلب لغات البرمجة الأقدم منها والأحدث. ولكن هنالك عقبة صغيرة أظنّ أن أغلبنا قد ظهرت له؛ ماذا لو احتجنا لمتحولٍ من نوعٍ جديد؟ أي ماذا لو أردْتُ تعريف متحول من نوع ليس موجود في اللغة التي أستخدمها؟ من المؤكد أنّ هذا السؤال قد خطر للعديد من المبرمجين وأولهم مصمم اللغة. وكان حل هذه المشكلة باستخدام الصفوف class.

بالتالي، فإن لغة C++ -كغيرها من اللغات-تقدم للمبرمج وسائل تعريف أنماط جديدة. وإن بناء أنماط جديدة قابلة للتكامل التام مع ما هو موجود في اللغة، هو خطوة نحو بناء برمجيات أفضل جودةً، ونحو قابلية إعادة استعمال الترميز الموجود، باستغلال أنماط المعطيات التي يبنيها المبرمج أثناء عمله.

وكما نعلم، يمكن للجميع أن يقوم ببناء عمليات جديدة إضافة إلى العمليات المعرفة سابقاً في اللغة، وذلك باستخدام مفهوم التابع. وتدعى هذه الآلية بالتجريد الإجرائي Procedural Abstraction، وهي آلية تسمح بها غالبية لغات البرمجة الإجرائية.

غير أنّ التجريد الإجرائي لا يشمل إلا جانباً واحداً من جوانب السيطرة على التطوير البرمجي.

وفي الحقيقة، من المفيد غالباً أن نتمكن من زيادة قوة تعبير اللغة ببناء أنماط جديدة، وذلك اعتماداً على ما يسمى ببنى المعطيات المجردة Abstract Data Types.



مخطط يوضح أهمية الصفوف والتوابع في لغات البرمجة

وكما سنرى لاحقاً، فإن المفهوم الذي يسمح لنا بتنفيذ أنماط المعطيات المجردة هو مفهوم الصف Class.
وإن السبب في تسمية هذه الأنماط بالأنماط المجردة إلى أنّ مستعملها لا يعرف بالضرورة تمثيلها الداخلي.
وهذه ميزة هامة تفيد أثناء كتابة البرامج التي تستعمل هذه الأنماط. فعندما يتعامل المبرمج مع الأنماط السابقة

التعريف في لغة البرمجة، فإنه يستعمل وضوحاً أنماطاً مجردة. ذلك أنه عندما يعطي المبرمج نمطاً ما لمتحول، يصبح للمتحول بنية داخلية معرفة جيداً، وتكون العمليات المسموح بها على هذه البنية مشروحة في دليل اللغة، دون الإشارة إلى تمثيلها الداخلي.

وهناك العديد من الأسباب التي تدعم العمل مع أنماط المعطيات منها:

- يعكس البرنامج العمل الذهني للمبرمج على وجه أفضل، عندما نسمح له بالعمل في مستوى إدراكي أعلى.
- يعزز تكرار استعمال الأنماط بالوقاية من الأخطاء، إذ يعلن المبرمج نواياه بصراحة بتحديد عدد القيم والعمليات.
- يسهل استعمال أنماط المعطيات بصيانة البرنامج.
- يتيح استعمال الأنماط المجردة للمعطيات التفكير بعمق أكبر بوظائف النمط بدلاً من بنية المعطيات الواجب تنفيذها لتحقيق هذا النمط المجرد.

أساسيات:

والآن سنتعرّف على مفهومي الغرض Object والصف Class.

الغرض Object:

وهو المفهوم الأساسي في البرمجة الغرضية التوجه. فالأغراض هي الكيانات الأساسية عند تنفيذ البرنامج. فالغرض يحتل مكاناً في الذاكرة الرئيسية، وتحدّد المعطيات المتضمنة فيه حالته. ولكن الغرض يتضمن إضافة إلى حالته جميع التوابع التي تحدد سلوكه behavior، وهذا ما نسميه عادة بالكبسلة encapsulation.

وتسمح الكبسلة بجمع المعطيات والتوابع التي تُنْشِئ كياناً. وبذلك يمكن رؤيتها ككل. وبالتالي تفيد الكبسلة في تقليص تشتت المعطيات والتوابع، وتُمَكّن المبرمج من فهم المسألة فهماً أفضل، وهذا مما يؤدي إلى تقليص تعقيدها.

ومع أن جميع المعلومات تُخَزَّن ضمن الغرض، فإنّ ذلك لا يعني أنّ بالإمكان الوصول إليها جميعها. إذ تُقْسَم معلومات غرض ما إلى قسمين:

- قسم عام Public: مرئي ويمكن الوصول إليه من خارج الغرض.
- قسم خاص Private: غير مرئي ولا يمكن الوصول إليه من خارج الغرض. ولكن يمكن الوصول إلى معلومات هذا القسم واستخدامها من قبل التوابع الأعضاء members في الغرض.

الصنف Class:

يُعَرِّف الصف مجموعة ممكنة من الأغراض. أي يَنْتُجُ كل غرض -عند تنفيذ البرنامج- من استنساخ صف موجود. فالصف هو تمثيل ساكن، جامد (في البرنامج) لمجموعة ممكنة من الأغراض المستنسخة.

وأقرب مثال لتوضيح العلاقة بين الغرض والصف: مصنع (صف) ينتج عدة منتجات (أغراض). وبإمكان المبرمج تحديد الطريقة التي يجب اتباعها لإنتاج الغرض.

لا نرى في البرنامج إلا الصفوف، في حين لا يوجد عند التنفيذ إلا الأغراض. يمكن مقارنة هذا السلوك بذلك الموافق للنمط، فتعريف صف يكافئ تعريف نمط جديد.

بالتالي، فإن الصف هو نمط يتضمن وصفاً موحَّداً لـ:

- الواصفات وهي مجموعة من المعطيات التي ترتبط فيما بينها، وتكون غالباً من أنماط مختلفة، وتسمّى بالمعطيات الأعضاء.
- السلوك: وهو مجموعة من الطرائق، وتسمّى بالتوابع الأعضاء والتي تسمح بالتعامل مع المعطيات الأعضاء.

يمكن تعريف الصف باستخدام الكلمة المفتاحية class، ويقع جسم الصف بين العلامتين {}، وينتهي تعريف الصف بـ ';'

ولدينا المثال الآتي عن الصف student :

```
class student{
      public:
            student();
            void read(int,string,int);
            void write();
      private:
            int age;
            string name;
            int num;
};
```

وبشكل عام، يوجد في الصف أربعة أنواع من التوابع الأعضاء:

- الباني Constructor: ويفيد في إعطاء قيم ابتدائية للأغراض في الصف.
- الهادم Destructor: ويسمح بهدم الغرض في نهاية حياته.
- النوافذ Accessors: تسمح بالنفاذ إلى المعطيات الأعضاء الخاصة والمحتواة في الغرض.
- المعدّلات Modificators: تسمح بتغيير حالة المعطيات الأعضاء الداخلية في الغرض.

وعند تعريف كل صف، تكون جميع الأعضاء (المعطيات والتوابع) خاصة private، وذلك إن لم تُوضع أسفل عبارة public. وفي حال وضعناها في القسم العام public، عندها نكون قد جعلنا الأعضاء قابلة للنفاذ من الخارج.

وهناك أيضا القسم الخاص private وفيه لا يمكن النفاذ للأعضاء إلا عن طريق التوابع الأعضاء الموجودة في الصف نفسه.

باختصار، الأعضاء الغير موجودة في قسم public ستعامل كما لو أنها في قسم private وذلك إن وُجِدَت به أم لا.

ويمكننا -أثناء تعريف الصف- تكرار الكلمات public و private عدة مرات و من دون ترتيب. حيث يمتد كل قسم من بداية الكلمة المحجوزة حتى الكلمة المحجوزة التالية.

وتستطيع التوابع الأعضاء النفاذ إلى جميع المعطيات في الصف سواء أكانت خاصة private أم لا. ولا يمكن النفاذ إلى الأعضاء الخاصة إلا من خلال التوابع الأعضاء العامة public.

تعريف التوابع الأعضاء:

عند تعريف التوابع الأعضاء، نستطيع تعريفها ضمن الصف الموجودة به. ولكن الأغلب يقوم بتعريفها خارج الصف. وإذا أردنا تعريف تابع عضو خارج الصف الخاص به، يجب أن يُسبَق معرِّف التابع باسم الصف الذي يُصَرِّح عن التابع. ويستعمل الرمز "::" لهذه الغاية، ويسمى بمعامل تمييز المدى الثنائي binary scope resolution operator. حيث أنه يحدد الصف الذي يمتلك التابع العضو. ونستنتج من ذلك إنه يمكن للتوابع الأعضاء في الصفوف المختلفة أن تمتلك نفس الاسم. وطبعاً عند تعريف التابع العضو داخل الصف لا داعٍ لهذا المعامل.

كيفية النفاذ لأعضاء الصف:

الآن أصبحنا نعرف أنه لا يحق لنا النفاذ إلى الأعضاء الخاصة أو استدعاء التوابع الخاصة. أما التوابع العامة، فيمكننا ذلك باستخدام المعامل "." .

أما التوابع الأعضاء في الصف، فتستطيع النفاذ إلى جميع المعطيات في الصف سواء أكانت خاصة أم لا، دون استخدام النقطة.

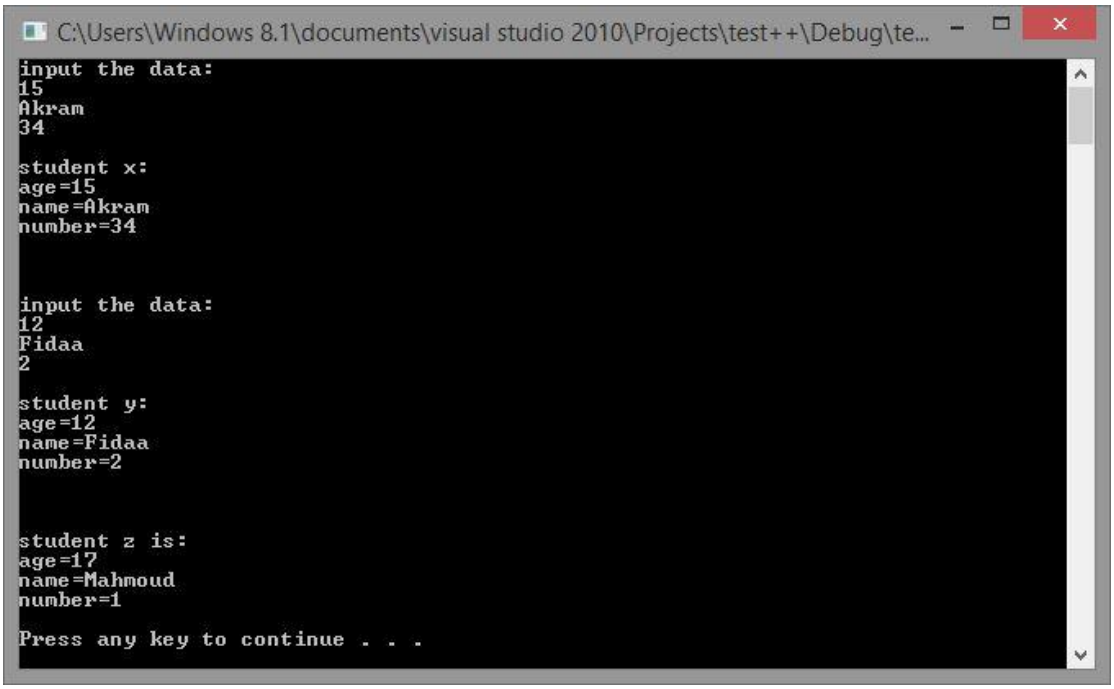
ولدينا المثال الآتي الذي يوضح الصف student:

```
#include<iostream>
#include<cstring>
#include<string>
using namespace std;
class student{
	public:
		student();
		void read(int,string,int);
		void write();
	private:
		int age;
		string name;
		int num;
};

student::student(){
	age=17;
	name="Mahmoud";
	num=1;
}
void student::read(int a,string b,int c){
	age=a;
	name=b;
	num=c;
}
```

```
void student::write(){
      cout<<"age="<<age<<"\nname="<<name<<"\nnumber="<<num<<"\n\n";
}
void main(){
      student x,y,z;
      int a,c;
      string b;
      cout<<"input the data:\n";
      cin>>a>>b>>c;
      x.read(a,b,c);
      cout<<"\nstudent x:\n";
      x.write();
      cout<<"\n\ninput the data:\n";
      cin>>a>>b>>c;
      y.read(a,b,c);
      cout<<"\nstudent y:\n";
      y.write();
      cout<<"\n\nstudent z is:\n";
      z.write();
      system("pause");
}
```

وسيكون خرج البرنامج:



خرج class student

البواني والهوادم:

البواني Cosntructors:

ما الذي يحدث عندما نصرح عن متحول من نمط الصف student مثلاً؟

يقوم المترجم compiler تابعاً خاصاً من التوابع الأعضاء يدعى بالباني constructor، فيقوم ببناء الغرض وإعطائه قيمة ابتدائية.

يحمل التابع الباني اسم الصف نفسه، وهو لا يندرج تحت النمط void وأيضاً ليس من التوابع التي ترد قيمة.

نستنتج مما سبق أن التابع الباني constructor هو تابع عضو خاص يقوم بإنشاء غرض من الصف وإعطاء قيماً ابتدائية للمعطيات الأعضاء في هذا الغرض. ويسمى بنفس اسم الصف، ولا يمكن أن يعيد قيمة.

ففي المثال السابق، التابع الباني للصف student هو student(). حيث قمنا بوضع قيمٍ ابتدائية للمعطيات age=17, name=Mahmoud, num=1، ثم قمنا في نهاية البرنامج بإظهار المتحول z بدون أن نعطيه قيماً، فأخذ القيم الابتدائية التي أعطاها له التابع student().

كما يمكن أن يتوفر عدة بوانٍ في الصف، وبالطبع، يختلف نموذج هذه البواني رغم كونها جميعاً تشترك بالاسم نفسه. وأيضاً يسمح بتعريف بوان تقبل قيماً افتراضية لمحدداتها.

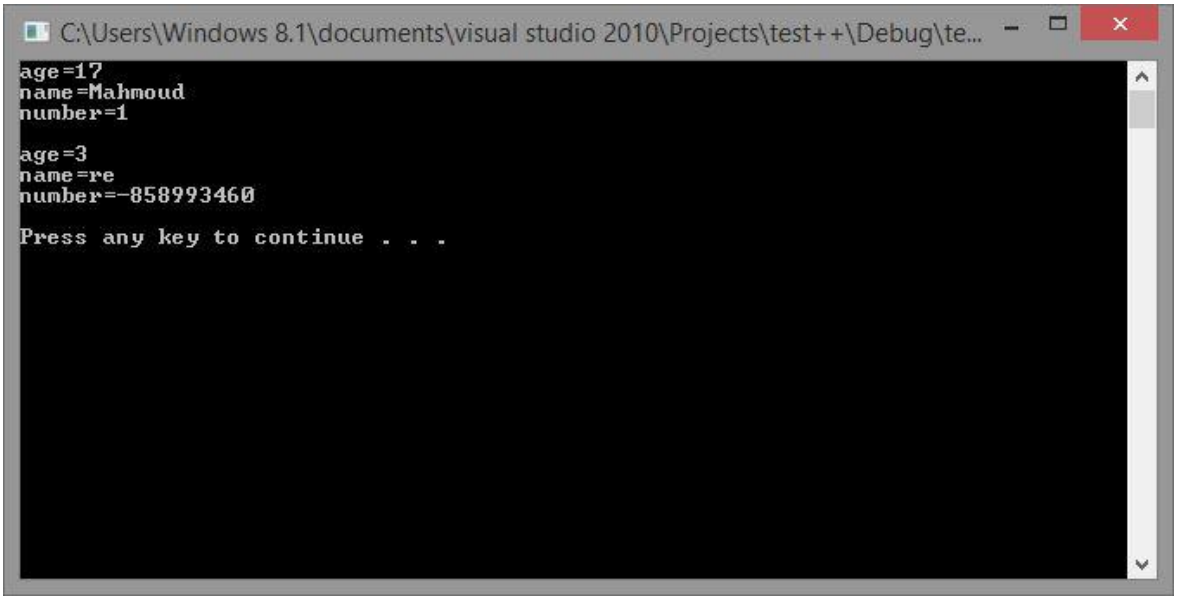
لنأخذ المثال الآتي:

```
#include<iostream>
#include<cstring>
#include<string>
using namespace std;
class student{
	public:
		student(){
			age=17;
			name="Mahmoud";
			num=1;
		}
		student(int a,string b){
			age=a;name=b;
		}
		void write(){

	cout<<"age="<<age<<"\nname="<<name<<"\nnumber="<<num<<"\n\n";
		}
	private:
		int age;
		string name;
		int num;
};
void main(){
	student x1,x2(3,"re");
	x1.write();
	x2.write();
```

```
        system("pause");
}
```

ويسكون الخرج:



خرج برنامج يحوي عدة توابع بانية

بالعودة للمثال، نلاحظ أننا في التابع main عرفنا متحولين x1, x2 من النوع student حيث أن x1 لم نمرر له أي قيمة، أما x2 فقد كُتِبَ بالشكل x2(2,"re"). وبالعودة لـ class student نلاحظ وجود تابعين بانيين، أحدهما بلا بارامترات والثاني يأخذ بارامترين int و string. بالتالي سيقوم الـ compiler مباشرة بمطابقة كل متحول حسب التابع الباني الذي يناسبه.

ملاحظة: بالعودة للخرج، نلاحظ أنه في المتحول x2، لم يأخذ المتحول num الخاص بـ x2 قيمة معينة، يمكننا تفسير ذلك بالعودة للبرنامج.

أنواع البواني:

الباني الافتراضي default constructor:

وهو الباني الذي لا يتضمن أية بارامترات. أي هو التابع student() في المثال السابق.

إذا لم يحدد المبرمج في تعريفه للصف أي بانٍ، يقوم الـ compiler بتوليد بانٍ افتراضي، وفائدة ذلك في كون الباني الافتراضي يقوم بانتظام باستدعاء جميع بواني المعطيات الأعضاء إن وُجِدَت. ويعتبر هذا الأمر مفيداً عندما تكون المعطيات الأعضاء في الصف من نمط صفوف أخرى. (وسندرس ذلك لاحقاً).

بالتالي، سيكون التابع student() هو التابع الباني للمتحول x1.

البواني ذوات المحددات constructors with arguments:

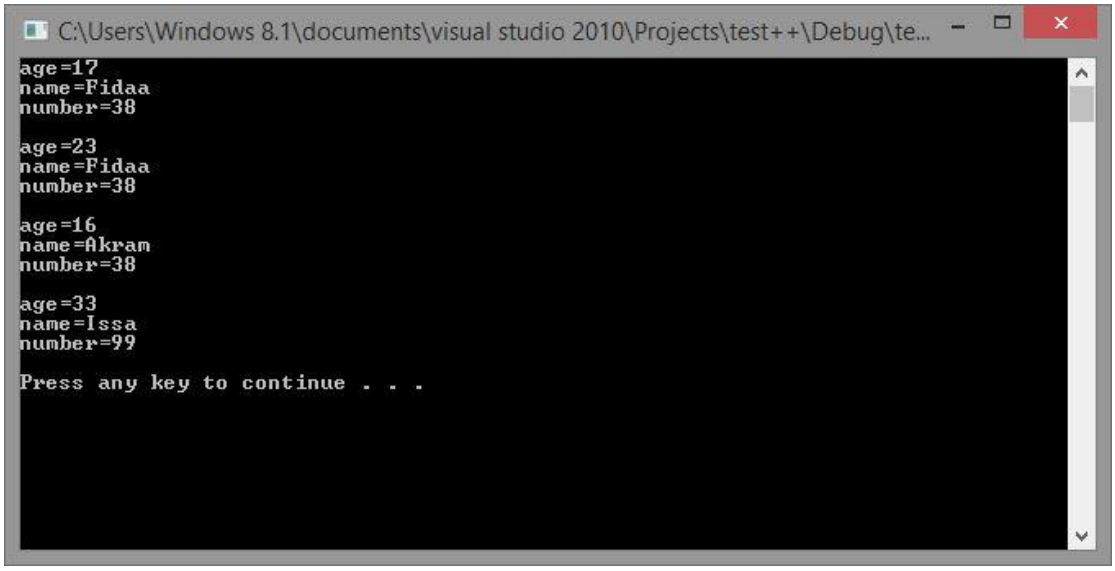
في بعض الأحيان، عند تعريف غرض ما من صف معين، قد نرغب بتحديد القيم الابتدائية التي نريد أن يأخذها. وبالتالي يلزمنا تابع بان جديد يأخذ بارامترات معينة. ومثال على ذلك، التابع student(int a,string b) في المثال السابق.

وكأي تابع في C++، يمكن أن نعطي قيماً ابتدائية لبارامترات التابع الباني.

لدينا المثال الآتي:

```
#include<iostream>
#include<cstring>
#include<string>
using namespace std;
class student{
	public:
		student(int=17,string="Fidaa",int=38);
		void write();
	private:
		int age;
		string name;
		int num;
};
student::student(int a,string b,int c){
	age=a;
	name=b;
	num=c;
}
void student::write(){
	cout<<"age="<<age<<"\nname="<<name<<"\nnumber="<<num<<"\n\n";
}
void main(){
	student x1,x2(23),x3(16,"Akram"),x4(33,"Issa",99);
	x1.write();
	x2.write();
	x3.write();
	x4.write();
	system("pause");
}
```

وسيكون الخرج:



خرج برنامج التابع الباني بالبارامترات الافتراضية

وعند كتابة الباني، يمكننا كتابته كما هو موجود في الأمثلة السابقة، أو يمكننا كتابته بالشكل:

```
student::student(int a,string b,int c):age(a),name(b),num(c){}
```

الهوادم Destructors :

أو كما يعرف أيضاً بالتابع المدمر. وهو كالباني، تابع عضو خاص. يحمل اسم الصف، ولكن مسبوقاً بالرمز "~". ومعنى ذلك أن التابع الهادم له دور متمم للتابع الباني.

ويستخدم التابع الهادم عندما يريد المستخدم تدمير الغرض، حيث يقوم بإنهاء الاحتفاظ بالذاكرة المخصص للغرض object حيق نستطيع تخزين أغراض جديدة.

باختصار، دور التوابع المدمرة هو التخلص من الأغراض القديمة لتخزين أغراض جديدة.

وإن التوابع الهادمة لا تأخذ بارامترات أو وسطاء، ولا يمكن إعادة قيم بواسطته. ولكل صف تابع مدمر وحيد خاص به.

و بشكل عام، يتم استدعاء التوابع الهادمة بشكل معاكس لاستدعاءات التوابع البانية.

الأغراض الثابتة:

جميعنا يعلم بالمتحولات الثابتة، بالتالي، فكما هنالك متحولات ثابتة، يجب أن يكون هناك أغراض ثابتة.

مثال:

```
const student x(15,"NEW",0);
```

وذلك يعني أننا عرّفنا غرضاً ثابتاً من الصف student وأعطيناه قيماً ابتدائية.

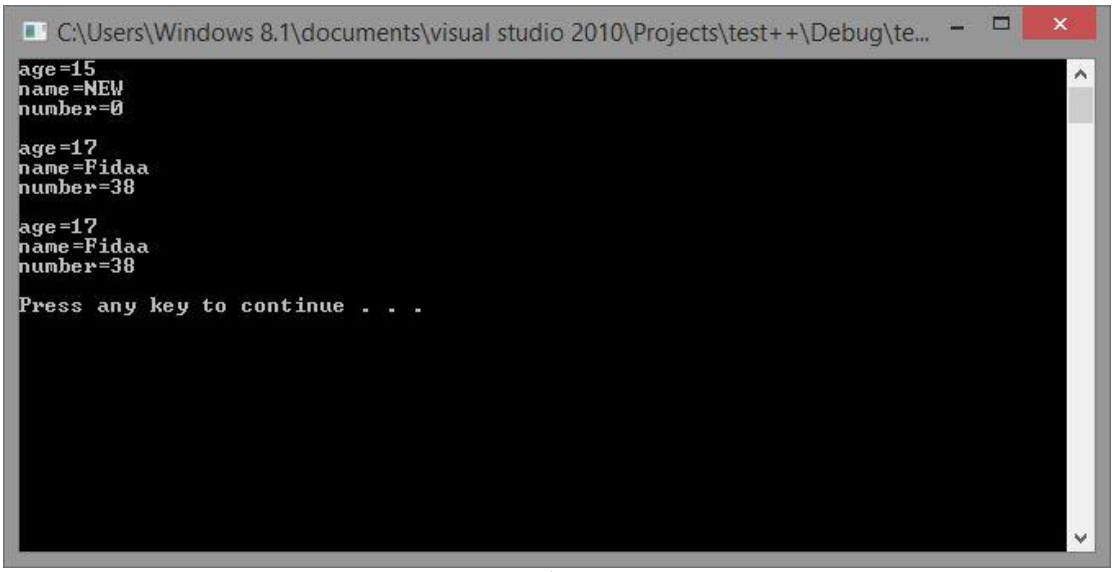
ولكن لا يمكننا التعامل مع الأغراض الثابتة إلا بتوابع أعضاء ثابتة، حيث لا تستطيع التوابع الثابتة تعديل الغرض.

لدينا المثال الآتي:

```
#include<iostream>
#include<cstring>
#include<string>
using namespace std;
class student{
	public:
		student(int=17,string="Fidaa",int=38);
		void write()const;
	private:
		int age;
		string name;
		int num;
};
student::student(int a,string b,int c):age(a),name(b),num(c){}
void student::write()const{
	cout<<"age="<<age<<"\nname="<<name<<"\nnumber="<<num<<"\n\n";
}
void main(){
	const student x0(15,"NEW",0),x1;
	student x2;
	x0.write();
	x1.write();
	x2.write();

	system("pause");
}
```

وسيكون الخرج:



خرج برنامج الأغراض الثابتة

ونستنتج من ذلك أن الأغراض الثابتة لا يمكن أن تتعامل إلا مع التوابع الثابتة، أما الأغراض الغير ثابتة، فيمكن أن تتعامل مع التوابع الثابتة والغير ثابتة. بالإضافة إلى أن التابع الباني ليس بتابع ثابت، ولكن يمكننا استخدامه لتعريف أغراض ثابتة. كما ظهر لنا في المثال.

الصفوف المركبة:

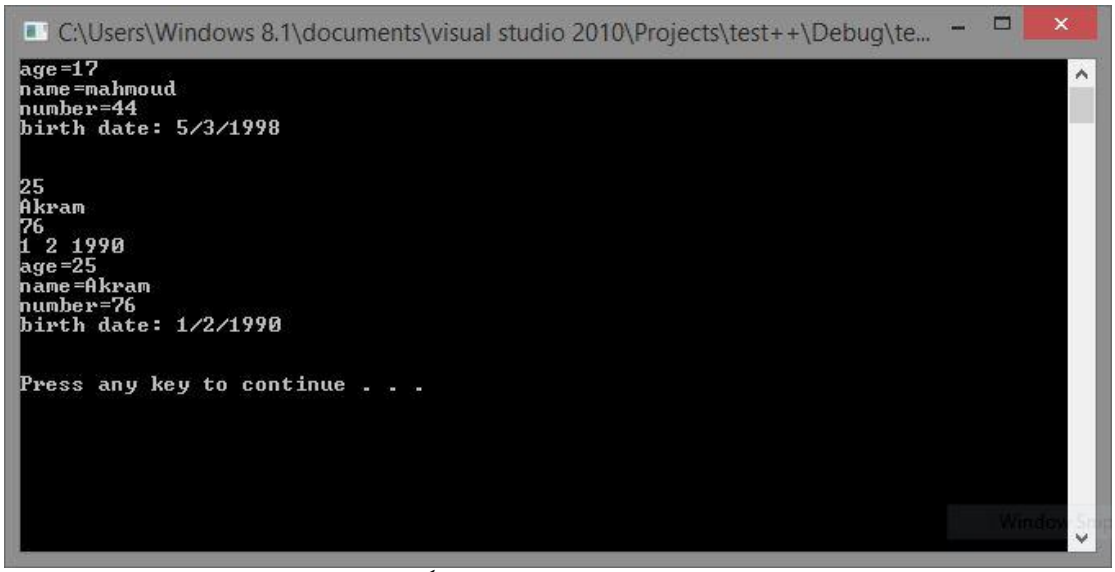
حتى الآن، جميع الصفوف التي مرت معنا كانت أنماط معطياتها بسيطة (char, double,....). ولكن ماذا لو احتجنا لمعطيات من نوع غير موجود؟. فمثلاً في الصف student ماذا لو احتجنا لمتحول عضو من نوع date مثلاً؟. كان حل ذلك باستخدام الصفوف المركبة.

لدينا المثال الآتي:

```
#include<iostream>
#include<cstring>
#include<string>
using namespace std;
class date{
	public:
		date(int,int,int);
		void write();
		void read(int,int,int);
	private:
		int day,month,year;
};
date::date(int a,int b,int c):day(a),month(b),year(c){};
void date::write(){
	cout<<day<<'/'<<month<<'/'<<year<<"\n\n\n";
}
void date::read(int a,int b,int c){
	day=a;month=b;year=c;
}
class student{
	public:
	student(int=17,string="Fidaa",int=38,int=1,int=1,int=1990);
		void write();
		void read(int,string,int,int,int,int);
	private:
		date birth;
		int age;
		string name;
		int num;
};
```

```
student::student(int a,string b,int c,int d,int e,int
f):birth(d,e,f),name(b),age(a),num(c){};
void student::write(){
      cout<<"age="<<age<<"\nname="<<name<<"\nnumber="<<num<<"\nbirth
date: ";
      birth.write();
}
void student::read(int a,string b,int c,int d,int e,int f){
      age=a;name=b;num=c;
      birth.read(d,e,f);
}
void main(){
      student x(17,"mahmoud",44,5,3,1998);
      x.write();
      student x1;
      int a,c,d,e,f;
      string b;
      cin>>a>>b>>c>>d>>e>>f;
      x1.read(a,b,c,d,e,f);
      x1.write();
      system("pause");
}
```

والخرج:



خرج برنامج يوضح الصفوف المركبة

ومن الأخطاء الشائعة أثناء استخدام صف مركب (يحوي غرض عضو)، أنه لا يُعطى الغرض العضو قيمة ابتدائية باستخدام تابع عضو لإعطاء قيم ابتدائية. وإذا لم يتوفر ضمن صف الغرض العضو تابع بناء افتراضي (إذا كان يوجد ضمنه أكثر من تابع بناء ولا يوجد تابع بناء افتراضي).

التوابع الصديقة والصفوف الصديقة:

لقد مر معنا أن التوابع الأعضاء هي فقط من يحق لها النفاذ إلى المعطيات الأعضاء. وأن كبسلتها معاً ضمن الصف هو الذي يؤمن خاصية إخفاء المعلومات هذه.

ولكن، وفي بعض الحالات، نجد حاجة لخرق هذه القاعدة، وبشروط خاصة جداً. نستخدم لهذا الغرض التوابع الصديقة friend function والصفوف الصديقة friend class.

ويمكن أن تنفذ التوابع الصديقة والصفوف الصديقة إلى الأعضاء الخاصة private في صف آخر بالرغم من أن التوابع الصديقة ليست توابع أعضاء في الصف، بل تُعَرَّف خارج مدى الصف.

وإن أهم خصائص الصداقة أنها تُمْنَح ولا تُؤخَذ. أي إذا كان A صديق B، فليس بالضرورة أن يكون B صديق A. وأيضاً إن الصداقة ليست علاقة متعدية، أي إذا كان A صديق B، وB صديق C، فليس بالضرورة أن يكون A صديق C.

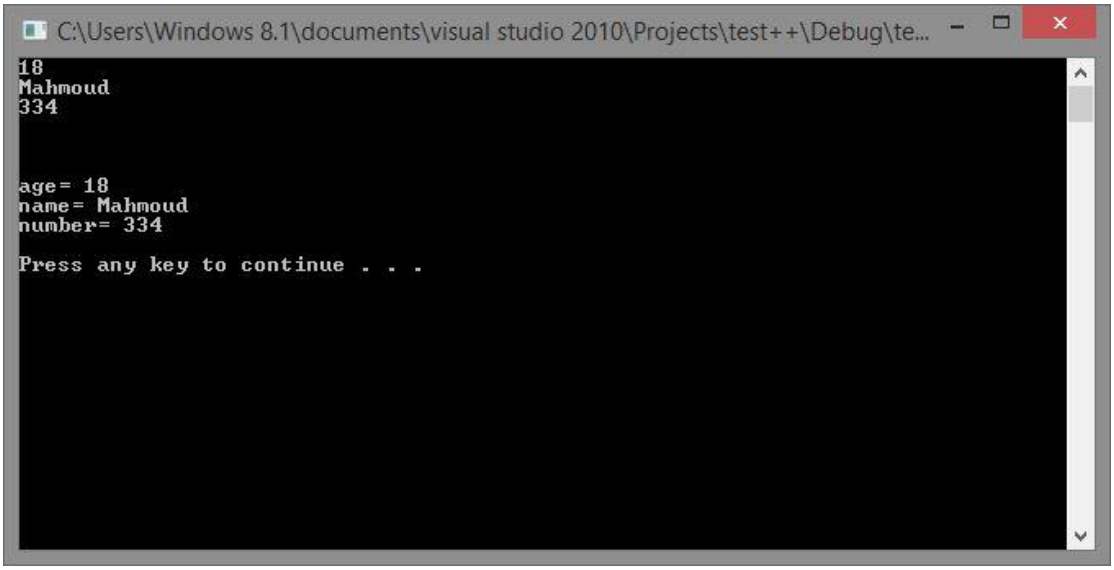
وللتصريح عن تابع صديق لصف ما يجب أن نكتب ترويسة التابع ضمن مدى الصف ونذكر قبلها كلمة friend.

ولدينا المثال الآتي الذي يوضح التوابع الصديقة:

```
#include<iostream>
#include<cstring>
#include<string>
using namespace std;
class student{
	friend void read(student &,int,string,int);
	public:
		student();
		void write();
	private:
		int age;
		string name;
		int num;
};
student::student(){
	age=-1;name="unknown";num=-1;
}
void student::write(){
	cout<<"age= "<<age<<"\nname= "<<name<<"\nnumber=
"<<num<<"\n\n";
}
void read(student &q,int a,string b,int c){
	q.age=a;q.name=b;q.num=c;
}
void main(){
	student x;
	int a,c;string b;
	cin>>a>>b>>c;
	read(x,a,b,c);
	cout<<endl<<endl<<endl;
```

```
	x.write();
	system("pause");
}
```

وسيكون الخرج:



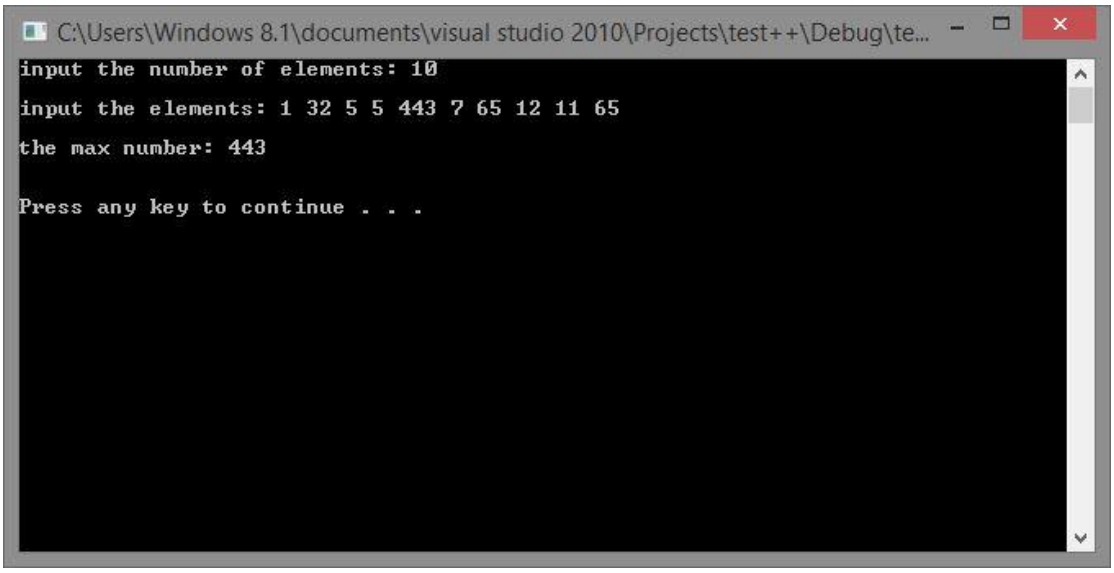
خرج برنامج يوضح التابع الصديق لصف معين

ولدينا المثال الآتي الذي يوضح الصفوف الصديقة:

```
#include<iostream>
using namespace std;
class values{
	friend class maxx;
	public:
		void read(int[],int);
	private:
		int x[100];
};
void values::read(int a[],int n){
	for(int i=0;i<n;i++)
		x[i]=a[i];
}
class maxx{
	public:
		int s(values,int);
};
int maxx::s(values q,int n){
	int t=q.x[0];
	for(int i=1;i<n;i++)
		if(t<q.x[i]) t=q.x[i];
	return t;
}
void main(){
```

```
	int x[100],n;
	cout<<"input the number of elements: ";
	cin>>n;
	cout<<"\ninput the elements: ";
	for(int i=0;i<n;i++)
		cin>>x[i];
	values z;
	z.read(x,n);
	maxx w;
	cout<<"\nthe max number: "<<w.s(z,n)<<endl<<endl<<endl;
	system("pause");
}
```

وسيكون الخرج:



خرج يوضح الصفوف الصديقة

حيث قمنا بالإعلان عن الصف maxx كصف صديق للصف values عندما كتبنا:

```
friend class maxx;
```

بالتالي، استطعنا الوصول إلى الأعضاء الخاصة الموجودة بالصف values من قبل الصف maxx.

وبشكل عام، تستعمل الصفوف الصديقة إذا كان هناك صفين مرتبطين ببعضهما لدرجة أن أحدهما يحتاج للوصول إلى بيانات الآخر بشكل مباشر.

تحميل العوامل بشكل زائد Operators Overloading:

لنفرض أن لدينا الأغراض x, y, z من نوع الصف time. ثم كتبنا التعليمة الآتية:

```
z=x+y;
```

سيخطر لنا مباشرة أن هذا صحيح. ولكن أذا فكرنا قليلاً، نلاحظ أنه لا يمكننا جمع غرضين، لأن لكل غرض بيانات و متحولات أعضاء خاصة به، أي ما الذي سوف يُجمَع؟

ولكن، فإن لغة C++ توفر ذلك، حيث تتيح العمل على الأغراض باستخدام العوامل نفسها التي تستخدم في الأنواع الأساسية.

وندعو العامل الذي أُعطِيَ القدرة على العمل على أنواع بيانات جديدة بالعامل المُحَمَّل بشكل زائد. ويتم تحميل العوامل بشكل زائد بكتابة توابع تحمل اسم العامل. فمثلاً: ترويسة التابع + المحمل بشكل زائد:

```
values operator+(values x);
```

حيث يقوم هذا التابع برد قيمة من نوع values وأخذ بارامترات من نوع values، حيث أن values هو صف عرفناه سابقاً.

وعند التحميل بشكل زائد، يجب مراعاة بعض القضايا:

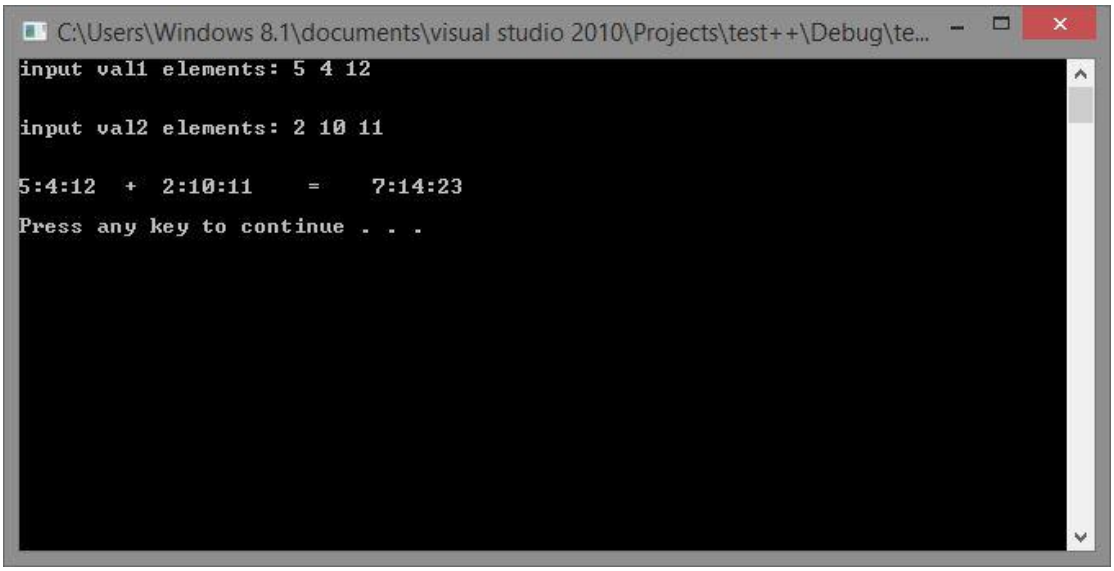
- هناك بعض العوامل التي لا تحمل بشكل زائد، وهي: "?:", "::", "*.", "."
- لا يمكن تغيير عدد المعاملات (البارامترات) التي يأخذها العامل.
- لا نستطيع إنشاء عوامل غير موجودة أصلاً في اللغة المستخدمة.
- تبقى أولوية precedence العامل كما هي .
- عند تطبيق العامل المحمل بشكل زائد على الأغراض (أي ليس على الأنواع الأساسية)، يمكنه تنفيذ أي عملية يريدها المبرمج. حيث يمكن للعامل + المحمل بشكل زائد أن يقوم بعملية طرح أو ضرب أو يمكن أن يقوم بإدخال قيم.

ولدينا المثال الآتي:

```
#include<iostream>
using namespace std;
class values{
	public:
		values operator+(values);
		void read(int,int,int);
		void write();
	private:
		int a,b,c;
};
void values::read(int x,int y,int z){
	a=x;b=y;c=z;
}
void values::write(){
	cout<<a<<":"<<b<<":"<<c;
}
values values::operator+(values v){
	values sum;
	sum.a=v.a+a;
	sum.b=v.b+b;
```

```
        sum.c=v.c+c;
        return sum;
}
void main(){
        values val1,val2;
        int a,b,c;
        cout<<"input val1 elements: ";
        cin>>a>>b>>c;
        val1.read(a,b,c);
        cout<<"\n\ninput val2 elements: ";
        cin>>a>>b>>c;
        val2.read(a,b,c);
        cout<<"\n\n";
        val1.write();
        cout<<"  +  ";
        val2.write();
        cout<<"    =    ";
        (val1+val2).write();
        cout<<"\n\n";
        system("pause");
}
```

وسيكون الخرج:



خرج برنامج يوضح التحميل الزائد للعوامل

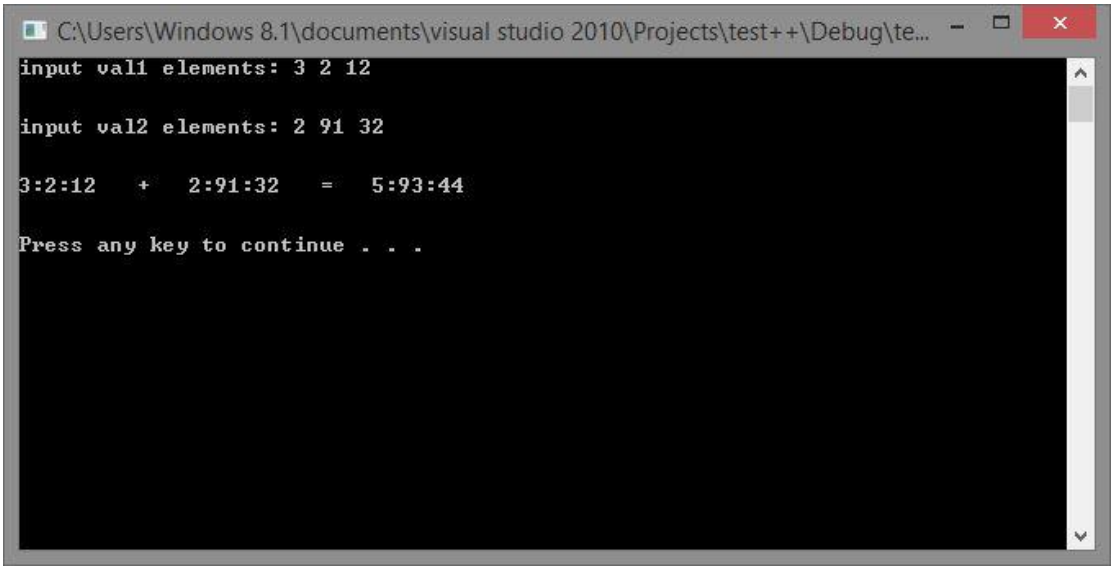
وكما رأينا في المثال السابق، إن التابع operator+() له وسيط واحد على الرغم من أنه يقوم بتحميل عامل الجمع الثنائي الذي يعمل على قيمتين. والسبب في ذلك أن المعامل على يسار العلامة + يتم تمريره إلى التابع بواسطة المؤشر this والمعامل على يمين العلامة هو الذي يتم تمريره كوسيط للتابع ولذلك يتم كتابة التابع بالشكل الآتي:

```
values operator+(values);
```

ويمكننا أيضاً تحميل المعاملات بشكل زائد باستخدام التوابع الصديقة.
ولدينا المثال الآتي يوضح ذلك:

```
#include<iostream>
using namespace std;
class values{
	friend values operator+(values,values);
	public:
		void read(int,int,int);
		void write();
	private:
		int a,b,c;
};
void values::read(int x,int y,int z){
	a=x;b=y;c=z;
}
void values::write(){
	cout<<a<<":"<<b<<":"<<c;
}
values operator+(values v1,values v2){
	values sum;
	sum.a=v1.a+v2.a;
	sum.b=v1.b+v2.b;
	sum.c=v1.c+v2.c;
	return sum;
}
void main(){
	values val1,val2;
	int a,b,c;
	cout<<"input val1 elements: ";
	cin>>a>>b>>c;
	val1.read(a,b,c);
	cout<<"\n\ninput val2 elements: ";
	cin>>a>>b>>c;
	val2.read(a,b,c);
	cout<<"\n\n";
	val1.write();
	cout<<"   +   ";
	val2.write();
	cout<<"   =   ";
	(val1+val2).write();
	cout<<"\n\n\n";
	system("pause");
}
```

وسيكون الخرج:



خرج برنامج يوضح تحميل المعاملات بشكل زائد باستخدام التوابع الصديقة

الخاتمة

وهكذا، نكون قد أسقطنا ضوءاً على أحد أهم الركائز الأساسية في عالم البرمجة. ونكون قد ساهمنا في لفت النظر إلى بحث مهم في مجال تطوير البرامج والبرمجيات الغرضية التوجه...

المراجع

كيف تبرمج بلغة C++، د. صلاح الدوه جي، دار شعاع للنشر والعلوم – سوريا.

بنى البيانات في C++، المهندس ساري علي الحاج حسين، سنة ٢٠١٠