جَامعة
المَنارة
MANARA UNIVERSITY

مقرر برمجة ٢
الجلسة الثامنة عملي

التحميل الزائد للمعاملات باستخدام التوابع الأعضاء:

أنشئ صف يمثل عدد عقدي Complex يملك البيانات الأعضاء الخاصة
الجزء الحقيقي والجزء التخيلي.
أنشئ باني يقوم بتهيئة المتحولات الأعضاء.
أضف على الكود التالي توابع تحميل معاملات الدخل والخرج

```
#include<iostream>
using namespace std;
class Complex {
private:    int real, imag;
public:
Complex(int r = 0, int i = 0) {real = r; imag = i;}

};
int main()
{
return 0;}
```

```
#include<iostream>
using namespace std;
class Complex
{private:
int real;
int image;
public:
Complex(int r = 0, int i = 0) {real = r; imag = i;}
friend istream &operator>>(istream &in, Complex &C);
friend ostream &operator<<(ostream &out, Complex  C);
 };
Istream  &operator>>(istream &in, Complex &C)
{//cout<<"enter the Real part";
in >>C.real;
//cout<"enter The imaginary part";
in >>C.image;return in; }
 ostream &operator<<(ostream &out, Complex C)
{out<< C.real<<"+i"<< C.image<<endl;return out;}
 int main()
{Complex c1;
cin>>c1;    cout<<c1;
return 0;}
```

التحميل الزائد المعاملات الدخل والخرج:

التحميل الزائد لمعاملي الزيادة والإنقاص:

أنشئ صنف اسمه count يحوي قيمة صحيحة value .
له باني افتراضي يقوم بإسناد القيمة 5 لمتحول الصنف.
تابع display لطباعة قيمة value.
حمل بشكل زائد معامل الزيادة الأحادي اللاحق ++ (postfix).
حمل بشكل زائد معامل الزيادة الأحادي السابق ++ (prefix).

```
#include <iostream>
using namespace std;
class Count {
private:
    int value;
public:
   Count(){value=5;}
   // Overload ++ when used as prefix
   void operator ++ () {++value;}
 // Overload ++ when used as postfix
   void operator ++ (int) {value++;}
   void display(){cout<<"Count: "<<value<<endl;}
};
int main() {
   Count count1;
   // Call the "void operator ++ (int)" function
   count1++;  count1.display();
   // Call the "void operator ++ ()" function
   ++count1;  count1.display();
   return 0;}
```

```
#include <iostream>
using namespace std;
class point {
int x,y;
public:
point (){x=0; y=0;} ;
void SET_x(double a){x=a;}
void SET_y(double b){y=b;}
void print1 (){cout << x<<endl<< y<<endl;} };
class circle {
int x,y,r ;
public:
circle (){x=0; y=0;r=0;} ;
void SET_x(double a){x=a;}
void SET_y(double b){y=b;}
void SET_r (double c){r=c;}
double area (){return 3.14*r*r;}
double circular (){return 2*3.14*r;}
void print ()
{cout << x<<endl<< y<<endl<<r<<endl;}};
```

```
int main()
{point p1 ; circle c1;
p1.SET_x(2); p1.SET_y(5); p1.print1();
c1.SET_x(2); c1.SET_y(5); c1.SET_r(3);
cout << c1.area()<< endl<<c1.
circular()<<endl;
c1.print();
return 0;}
```

ليكن لدينا برنامج للتعامل مع الأشكال الهندسية يستخدم كلاً من النقطة والدائرة حيث تعرف النقطة بإحداثياتها x,y كصنف أما الدائرة فتعرف بمركزها وهو نقطة احداثياتها x,y ونصف قطر r كصنف آخر. نلاحظ ما يلي:

هناك تشابهاً كبيراً بين الصنفين من حيث الأعضاء والتوابع العامة وقد تم تكرار مقاطع برمجية متعددة في الصنفين. ان هذا التكرار لا يعد من الممارسات البرمجية الجيدة وقد قدمت البرمجة غرضية التوجه مفهوم الوراثة لتجنبه

```
#include <iostream>
using namespace std;
class point {
private:
int x,y;
public:
point(){x=0; y=0;} ;
void SET_x(double a){x=a;}
void SET_y(double b){y=b;}
void print1(){cout <<x<<endl<<y;}};
class circle : public point {
int r ;
public:
circle (){r=0;} ;
void SET_r (double c){r=c;}
double area (){return 3.14*r*r;}
double circular (){return 2*3.14*r;}
void print (){cout <<r<<endl;} };
```

```
int main()
{circle c1;
c1.SET_x(2);
c1.SET_y(5);
c1.SET_r(3);
cout << c1.area()<< endl;
cout<<c1.circular()<<endl;
c1.print();
c1. print1();
return 0;}
```

بالعودة الى مثالنا point / circle : يصبح البرنامج كما يلي بتطبيق الوراثة العامة.

لاحظ ما يلي:

1- عدم التكرار في الكود

2- ماذا لو وضعنا التابع print في الصنف circle بالصياغة التالية:

```
void print (){cout << x<<endl<<
y<<endl<<r<<endl;}
```

جَامعة
المَنارة
MANARA UNIVERSITY

انتهت الجلسة