Clase 22. Sobrecarga de operadores, ejemplos de polimorfismo, funciones virtuales.

Sobrecarga de operadores

Sobrecarga de operadores

En C++ se pueden sobrecargar la mayoría de los operadores. Los operadores sobrecargados son funciones con nombres especiales y la palabra **operator** seguida por el símbolo que será definido. Como cualquier otra fución el operador tiene un tipo de retorno y una lista de parámetros.

```
1  Clase operator + (const Clase &)
```

Ejemplo sobrecarga fuera de la clase

Si la función del operador recibe dos parámetros puede sobrecargarse fuera de una clase.


```
using namespace std;
typedef struct{
    int n;    char *str;
}MSTRING;

MSTRING operator + (MSTRING &str1,MSTRING &str2){
 int l1=strlen(str1.str); int l2=strlen(str2.str);
 MSTRING str3;  str3.str=new char[l1+l2+2];
 strcpy(str3.str,str1.str);  strcat(str3.str," ");
 strcat(str3.str,str2.str);  str3.n=lstr1+lstr2;
return str3;
}

int main(){
    MSTRING str1,str2,str3;
    str1=InitStr("Hola"); str2=InitStr("Mundo!");
    str3=str1+str2;
    cout << str3.str << endl;
    destroy(str1);destroy(str2); destroy(str3);
   return 0;
}
```

Ejemplo sobrecarga dentro de la clase

El operador **=** solo se puede sobrecargar si es miembro de la clase.


```
#include <iostream>
#include <string.h>
using namespace std;
class MSTRING{
    int n;  char *str;
public:
    MSTRING(){n=0; str=NULL;}

    MSTRING& operator =(const char *strc){
      int l1=strlen(strc);  n=lstr1;
      str=new char[l1+1]; strcpy(str,strc);
    return  *this;
   }
   ~MSTRING(){
        if(str)
          delete []str;
        str=NULL;
   }
   friend MSTRING operator +(MSTRING &str1,MSTRING &str2
     );
};
```

Retorno por referencia

En C++ se pueden hacer retornos por referencia. Esto quiere decir que la función implícitamente esta trabajando con la memoria de un objeto, no con el objeto mismo. Lo cual permite que el objeto sea un lvalue.

- La función devuelve str[i], str[i] es un **char**, en la devolución por copia, C++ trabaja con el valor de str[i], es decir devolvería **x**, en la de referencia, el contenido es el mismo, pero además el programa sabe donde (en la memoria) está ese contenido, y puede modificar esa posición de la memoria.
- cout considera un **char**.
- NO es una dirección perse, pero siempre se sabe cual es la dirección (por ejemplo para igualar el apuntador).

```
char str[100]={"Texto prueba"};
char &String(const char c){
 int i=0;
 while (str[i]!='\0' && str[i]!=c)
        i++;
return str[i]; //devuelve la referencia a str[i]
}
```

Retorno por referencia


```
1  #include <iostream>
   #include <string.h>
3  using namespace std;
   char str[100]={"Texto prueba"};
5  char &String(const char c){
    int i=0;
7   while (str[i]!='\0' && str[i]!=c)
          i++;
9  return str[i]; //devuelve la referencia a str[i]
   }
11
   int main(){
13   cout << String('x') << endl;
     char *x=&String('x');
15   cout << x << endl;
     return 0;
17 }
```

```
x
xto prueba
```

Retorno por referencia


```
1  #include <iostream>
   #include <string.h>
3  using namespace std;
   char str[100]={"Texto prueba"};
5  char &String(const char c){
    int i=0;
7   while (str[i]!='\0' && str[i]!=c)
          i++;
9  return str[i]; //devuelve la referencia a str[i]
   }
11 int main()
   {
13   String('T')='C';
     String('e')='a';
15   String('x')='m';
     String('t')='b';
17   String('o')='i';
     String(' ')='o';
19   cout << &String('C') << endl;
     return 0;
21 }
```

```
Cambioprueba
```

Retorno por referencia

En el ejemplo anterior la función tiene un retorno por referencia:

```
MSTRING& operator =(const char *strc){
  int lstr1=strlen(strc);
  str=new char[lstr1+1]; n=lstr1;
  strcpy(str,strc);
 return  *this;
}
```

En el retorno por referencia se indica que la memoria con la que se trabaja es memoria existente (que es a la que apunta **this** en la clase), y que esa memoria realmente solo se está modificando pero que no fue generada dentro de la función.

Código con error

```
 #include <iostream>
2 #include <string.h>
using namespace std;
4 class MSTRING{
    int n;    char *str;
6 public:
    MSTRING(){n=0; str=NULL;}
8   char *getStr(){return str;}
    MSTRING operator =(const char *strc){
10   int lstr1=strlen(strc); n=lstr1;
     str=new char[lstr1+1]; strcpy(str,strc);
12  return  *this;
   }
14 ~MSTRING(){
       if(str) delete []str;
16 }
};
18 int main(){
    MSTRING str1,str2,str3;
20  str1="Hola"; str2="Mundo";
    cout << str1.getStr() << str2.getStr()<< endl;
22 return 0;
}
```

Destructor llamado multiples veces


```
1  #include <iostream>
   #include <string.h>
3  using namespace std;
   class MCLASE{
5  public:
      MCLASE(){cout<< "construye" <<endl;}
7     ~MCLASE(){cout<< "destruye" <<endl; }
     friend MCLASE regresa(MCLASE x){ return x;}
9  };

11 int main()
   {
13    MCLASE Obj;
      regresa(Obj);
15   return 0;
   }
```

```
construye
destruye
destruye
destruye
```

Destructor llamado multiples veces 2.

```
class MCLASE{
public:
    MCLASE(){cout<< "construye" <<endl;}
    MCLASE(MCLASE &){cout<< "construye" <<endl;}
   ~MCLASE(){cout<< "destruye" <<endl; }
   friend MCLASE regresa(MCLASE x){ return x;}
};
int main(){
    MCLASE Obj;
    regresa(Obj);
   return 0;
}
```

```
construye
construye
construye
destruye
destruye
destruye
```

Código sin error, retorno por referencia


```
 #include <iostream>
#include <string.h>
using namespace std;
class MSTRING{
    int n;    char *str;
public:
    MSTRING(){n=0; str=NULL;}
    char *getStr(){return str;}
    MSTRING& operator =(const char *strc){
      int lstr1=strlen(strc); n=lstr1;
      str=new char[lstr1+1]; strcpy(str,strc);
    return  *this;
   }
   ~MSTRING(){ if(str) delete []str;  }
};

int main(){
    MSTRING str1,str2,str3;
    str1="Hola"; str2="Mundo";
    cout << str1.getStr() << str2.getStr()<< endl;
   return 0;
}
```

Sobrecarga de operadores unarios


```
 #include <iostream>
#include <string.h>
using namespace std;
class MCLASE{
  int x, val,bound;
  char name[128];
public:
  MCLASE(int b, const char *n)
  {strcpy(name,n); bound=b; x=-11; val=-1;}
  int operator=(MCLASE Obj){
      Obj.x=x; Obj.val=val; Obj.bound=bound;
      cout<< "Se copio de "<<name<<" a "<< Obj.name<<
     endl;
     return Obj.bound;
  };
};

int main(){
   MCLASE Obj1(100,"Obj1"), Obj2(5,"Obj2");
    int x=-10;
    x=(Obj1=Obj2);
    cout<< "x= "<< x<<endl;
  return 0;
}
```

Recordatorios

- El operador = solo es de asignación (usa la sobrecarga) si se llama después de instanciar, sino se llama el constructor por copia.

Recordatorios

- El operador **=** solo es de asignación (usa la sobrecarga) si se llama después de instanciar, sino se llama el constructor por copia.
- El operador sobrecargado tiene un valor de retorno, entonces si ese valor es entero el hacer **(**Obj1=Obj2) nos devuelve un número entero, si queremos que transitividad, necesitamos que devuelva un tipo **MCLASE&**.
- En muchos casos (de operadores unarios) es conveniente sobrecargar los operadores para que reciban y devuelvan valores por referencia y asi evitar llamadas al constructor/destructor innecesarias y que pueden corromper la memoria.

Operadores binarios


```
1  #include <iostream>
   #include <string.h>
3  using namespace std;
   class MCLASE{
5  public:
    int x, val,bound; char name[128];
7   MCLASE(int b, const char *n)
    {strcpy(name,n); bound=b; x=-11; val=-1;}
9   MCLASE(MCLASE&){cout<<"copia"<<endl;}
    MCLASE operator+(const MCLASE &Obj){
11   MCLASE res(0,"resultado");
     res.x=Obj.x+x; res.val=Obj.val+val; res.bound=Obj.
       bound+bound;
13   cout<< "Suma "<<name<<" a "<< Obj.name<<endl;
     return res;
15  };
   };
17 int main(){
    MCLASE Obj1(100,"Obj1"), Obj2(5,"Obj2");
19  MCLASE Obj3(0,"");
    Obj3=(Obj1+Obj2);
21  cout<<"res="<<Obj3.bound<<" nombre="<<Obj3.name<<endl;
   return 0;
23 }
```

Operadores binarios

```
MCLASE operator+(const MCLASE &Obj){
 MCLASE res(0,"resultado");
  res.x=Obj.x+x; res.val=Obj.val+val; res.bound=Obj.
    bound+bound;
  cout<< "Suma "<<name<<" a "<< Obj.name<<endl;
  return res;
 };
int main(){
 MCLASE Obj1(100,"Obj1"), Obj2(5,"Obj2");
 MCLASE Obj3(0,"");
 Obj3=(Obj1+Obj2);
 cout<<"res="<<Obj3.bound<<" nombre="<<Obj3.name<<endl;
return 0;
}
```

```
Suma Obj1 a Obj2
res= 105 nombre resultado
```

Operadores Relacionales


```
1  #include <iostream>
   #include <string.h>
3  using namespace std;
   class MCLASE{
5  public:
     int x, val,bound;
7    char name[128];
     MCLASE(int b, const char *n)
9    {strcpy(name,n); bound=b; x=-11; val=-1;}
     bool operator <(int v){return bound<v;  };
11   MCLASE& operator <(MCLASE &X){bound=bound*(bound<X.
       bound);
       return *this;  };
13 };
   int main(){
15   MCLASE Obj1(100,"Obj1"),Obj2(50,"Obj2");
     MCLASE Obj3(0,"res");
17   Obj3=(Obj2<Obj1);
     cout<< "res= "<< Obj3.bound << endl;
19   return 0;
   }
```

```
res= 50
```

Operadores sobrecargables


```
+ - * / % ^
& | ~ ! , =
< > <= >= ++ --
<< >> == != && ||
+= -= /= %= ^= &=
|= *= <<= >>= [] ()
-> ->* new new [] delete delete []
```

Operadores no sobrecargables:

```
:: .* . ?:
```

Sobrecarga de << para cout


```
1  #include <iostream>
   #include <string.h>
3  using namespace std;
   class MCLASE{
5  private:
     int x, val,bound;
7  public:
     char name[128];
9    MCLASE(int b, const char *n){strcpy(name,n); bound=b;
        x=-11; val=-1;}
     friend ostream& operator<<(ostream &out, MCLASE &);
11 };

13 ostream& operator<<(ostream &out, MCLASE &O){
      out<<"x=" << O.x<< " val=" << O.val<<" bound="<<O.
       bound <<" name="<< O.name << endl;
15    return out;
   };
17 int main()
   {
19    MCLASE Obj1(100,"Obj1");
      cout << Obj1 ;
21   return 0;
   }
```

Polimorfismo

Polimorfismo

Una forma de polimorfismo es que una función con el mismo nombre realice acciones diferentes de acuerdo con el objeto que la llama. Es decir:

Una función con los mismos parámetros de entrada, puede hacer cosas diferentes si es llamadas por la clase base, o si es llamada por la clase derivada.

Polimorfismo,ejemplo


```
 #include <iostream>
#include <string.h>
using namespace std;
class REAL{
public:
  double x;
  REAL(){x=0;};
  REAL(double val){x=val;};
  REAL operator+(REAL &Obj){
    REAL X;  X.x=x+Obj.x;
    return X;
  }
};
class INT : public REAL{
public:
  INT():REAL(){};
  INT(double val){x=(int)val
    ;};
  INT operator+(REAL &Obj){
    INT X; X.x=(int)x+(int)Obj
    .x;
    return X;
  }
};
```

```
ostream& operator<<(
    ostream &out, REAL &
    O){
   out<< O.x;
   return out;
};
int main(){
  INT Obj1(3.94), Obj2
    (6.8);
  REAL Obj3(3.5);
  Obj3=Obj3+Obj1 ;
  Obj2=Obj1+Obj2;
  cout<< Obj2<< endl;
  return 0;
}
```

Polimorfismo,ejemplo


```
#include <iostream>
using namespace std;
class Parent{
public:
 int funcion(){cout << "Padre"<<endl; }
};

class Derived:public Parent{
public:
  int funcion(){ cout << "Derivada" << endl; }
};

int main(){
   Derived Obj;
   Parent *ptr;
   ptr=&Obj;
   ptr->funcion();
  return 0;
}
```

```
Padre
```

Polimorfismo,ejemplo


```
#include <iostream>
using namespace std;
class Parent{
public:
    int funcion(){ cout << "Padre"<<endl;}
    int funcion(double X){cout << "Padre"<< X<<endl;}
};
class Derived:public Parent{
public:
  int funcion(){cout << "Derivada" << endl;}
  int funcion(int Y) { cout << "Derivada" << Y<<endl; }
};
int main(){
  Derived Obj;  Parent *ptr; ptr=&Obj;
  ptr->funcion(); Obj.funcion();
  Obj.funcion(3.5); Obj.Parent::funcion(5);
return 0;
}
```

```
Padre
Derivada
Derivada3
Padre5
```

Funciones virtuales

Otra forma de polimorfismo es definir funciones **virtuales**, estás pueden estar definidas tanto en la clase base como en la clase derivada, y de acuerdo al objeto se manda llamar la función especifica de ese objeto. En otras palabras:
Una función miembro virtual es una función que es redefinida en cada clase derivada y conserva las propiedades de la llamada cuando es pasada por referencia (apuntadores).

Polimorfismo,ejemplo, funciones virtuales


```
#include <iostream>
#include <math.h>
using namespace std;
class REAL{
protected:
  double x;
public:
 REAL(){x=0;};
 REAL(double val){x=val;};
 virtual double SQRT();
};
class INT : public REAL{
public:
 INT():REAL(){};
 INT(double val){x=(int)val;};
  double SQRT();
};
double INT::SQRT(){
  if ((x-y*y)>((y+1.0)*(y+1.0)
    -x))
     return y+1.0;
  return y;
}
```

```
double REAL::SQRT()
{ if (x<0) return 0;
  return sqrt(x);;
}

int main()
{
  INT X(7); REAL Y(7);
  REAL *PTR=&X;
  cout<< PTR->SQRT() <<
    endl;
  PTR=&Y;
  cout<< PTR->SQRT() <<
    endl;
  return 0;
}
```

Polimorfismo,ejemplo, funciones puramente virtuales


```
#include <iostream>
#include <math.h>
using namespace std;
class REAL{
protected:
  double x;
public:
  REAL(double val=0){x=val;};
  virtual double SQRT()=0;
};
class INT : public REAL{
public:
  INT(double val=7){x=(int)val
    ;};
  double SQRT();
};
double INT::SQRT(){
  if ((x-y*y)>((y+1.0)*(y+1.0)
    -x))
    return y+1.0;
  return y;
}
```

```
int main()
{
  INT X;
  REAL *PTR=&X;
  cout<< PTR->SQRT() <<
    endl;
  return 0;
}
```

Note como los constructores reciben parámetros por default en la **declaración**, aunque en este caso se realiza la declaración al mismo tiempo que la definición.

Tarea

Tarea 12.1 Defina una clase matriz, con los operadores de suma, resta y multiplicación sobrecargados y virtuales (no puramente virtuales). Verifique que no haya perdida de memoria (memory leaks). Defina una clase derivada matriz simétrica, que redefina los operadores de suma, resta y multiplicación, coniderando la simetría (realizar menos operaciones). Agregue un caso de prueba al programa (en archivo o en el mismo main), para una matríz simétrica y una no-simétrica.