Polimorfismo

Polimorfismo. Métodos virtuales

- El polimorfismo indica que una variable pasada o esperada puede adoptar múltiples formas. Cuando se habla de polimorfismo en programación orientada a objetos se suelen entender dos cosas:

  1. La primera se refiere a que se puede trabajar con un objeto de una clase sin importar de qué clase se trata. Es decir, se trabajará igual sea cual sea la clase a la que pertenece el objeto. Esto se consigue mediante jerarquías de clases y clases abstractas.

  2. La segunda suele referirse a la posibilidad de declarar métodos con el mismo nombre que pueden tener diferentes argumentos dentro de una misma clase.

- La capacidad de un programa de trabajar con más de un tipo de objeto se conoce con el nombre de polimorfismo

Polimorfismo

- Hasta ahora la herencia se ha utilizado solamente para heredar los miembros de una clase base, pero también existe la posibilidad de que un método de una clase derivada se llame como método de la clase base pero tenga un funcionamiento diferente.

- Por ejemplo, tomemos la clase disco_musica que hereda de la clase base disco:

```
class disco{
protected:
int capacidad;
char * fabricante;
int num_serie;
public:
disco (int c, char * f, int n){
capacidad = c; strcpy(fabricante, f);
num_serie=n;}
```

Polimorfismo

```
  void imprimir_capacidad(){
  cout << capacidad; }
  void imprimir_fabricante(){
  cout << fabricante; }
  void imprimir_num_serie(){
  cout << num_serie; }
};

  class disco_musica : public disco{
  private:
  char * tipo; //CD,vinilo, mini-disc, etc.
  cancion * lista_canciones;//Suponemos la
  //existencia de una clase “cancion”.
  public:
  disco_musica (int c, char* f, int n,
  char* t) : disco (c,f,n){
  strcpy(tipo,t);
  lista_canciones=NULL;}
  };
```

Polimorfismo

- Podemos redefinir el método imprimir del ejemplo anterior de la siguiente forma:

```
void imprimir_fabricante(){
cout <<  “Este disco de musica ha
sido grabado por: ”  << fabricante; }
//fabricante ha de ser protected!
```

- Si tenemos, por ejemplo,

```
void main(){
disco_musica d1(32,“EMI”,423,“CD”);
disco * ptr = & d1;
d1.imprimir_fabricante();
//Invoca al de la clase derivada.
ptr -> imprimir_fabricante();
//Invoca al de la clase base.
}
```

- Cuando se invoca un método de un objeto de la clase derivada mediante un puntero a un objeto de la clase base, se trata al objeto como si fuera de la clase base.

Polimorfismo

- Este comportamiento puede ser el deseado en ciertos casos, pero otras veces tal vez se desee que el comportamiento de la clase desaparezca por completo. Es decir, si el objeto de la clase `disco_musica` se trata como un objeto de la clase `disco`, se quiere seguir conservando el comportamiento de la clase `disco_musica`. Para ello se utiliza la palabra reservada **virtual**, si se quiere que el comportamiento de un método de la clase `disco` desaparezca se ha de declarar el método como **virtual.**

- Si queremos modificar la clase `disco` para que se pueda sobrecargar su comportamiento cuando se acceda a la clase `disco_musica` a través de un puntero a `disco` pondríamos:

Polimorfismo

```
class disco{
  protected:
  int capacidad;
  char * fabricante;
  int num_serie;
  public:
  disco (int c, char * f, int n){
  capacidad = c; strcpy(fabricante, f);
  num_serie=n;}
  void imprimir_capacidad(){
  cout << capacidad; }
  virtual void imprimir_fabricante(){
  cout << fabricante; }
  void imprimir_num_serie(){
  cout << num_serie; }
};
```

Polimorfismo

- Así, la clase disco ahora sobre el mismo extracto de programa anterior produce otro efecto:

```
 void main(){
 disco_musica d1(32,“EMI”,423,“CD”);
 disco * ptr = & d1;
 d1.imprimir_fabricante();
 //Invoca al de la clase derivada.
 ptr -> imprimir_fabricante();
 //Invoca al de la clase derivada.
}
```

- En este caso el objeto será siempre el de la clase derivada independientemente de cómo se pase el objeto a otras funciones. A esto es a lo que suele referir el término polimorfismo.
- Los métodos virtuales heredados son también virtuales en la clase derivada, por lo que si alguna clase hereda de la clase derivada el comportamiento será similar al indicado.

Polimorfismo

- Los únicos métoso que no pueden ser declarados como virtuales son los **constructores**, los métodos estáticos, y los operadores **new** y **delete**.

Clase virtual pura

Hay veces en las que no va a ser necesario crear objetos de la clase base, o simplemente no se desea que quien utilice la clase pueda crear objetos de la clase base. Para ello existen lo que suele llamarse en POO clases abstractas. Esta clase define el interfaz que debe tener una clase y todas las clases que heredan de ella. En C++ el concepto de clases abstractas se implementa mediante **funciones virtuales puras.** Estas funciones se declaran igual que cualquier otra función anteponiendo la palabra **virtual** y añadiendo al final de la declaración =0. Para estas funciones no se proporciona implementación. En C++ la clase abstracta debe tener uno o más métodos virtuales puros.

- Dada una clase abstracta, no se pueden crear objetos de esa clase base. Se pueden crear punteros que a objetos de la clase base abstracta que realmente apunten a objetos de la clase derivada.

Polimorfismo

Ejemplos/Ejercicios:

1) Dadas las clases:

```
class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   char *VerNombre(char *n) { strcpy(n, nombre);
   return n;}
  protected:
   char nombre[30];
};
class Empleado : public Persona {
  public:
   Empleado(char *n) : Persona(n) {}
   char *VerNombre(char *n) { strcpy(n, "Emp:
   "); strcat(n, nombre); return n;}
};
class Estudiante : public Persona {
  public:
   Estudiante(char *n) : Persona(n) {}
   char *VerNombre(char *n) { strcpy(n, "Est:
   "); strcat(n, nombre); return n;}
};
```

¿Qué salida tiene el programa siguiente?:

Polimorfismo

```
int main()
{
   char n[40];
   Persona *Pepito = new
  Estudiante("Jose");
   Persona *Carlos = new
  Empleado("Carlos");

   cout << Carlos->VerNombre(n) <<
  endl;
   cout << Pepito->VerNombre(n) <<
  endl;
   delete Pepito;
   delete Carlos;
   system("pause");
   return 0;
}
```

Polimorfismo

Solución:
La salida es:
Carlos
Jose
Presione cualquier tecla para continuar . . .

Vemos que se ejecuta la versión de la función "VerNombre" que hemos definido para la clase base

2) Si modifcamos en el ejemplo anterior la declaración de la clase base "Persona“ como:

```
class Persona {
  public:
   Persona(char *n) { strcpy(nombre, n); }
   virtual char *VerNombre(char *n) { strcpy(n,
   nombre); return n;}
  protected:
   char nombre[30];
};
```

¿Qué salida tiene el programa anterior?

Polimorfismo

Solución:

La salida es como ésta:

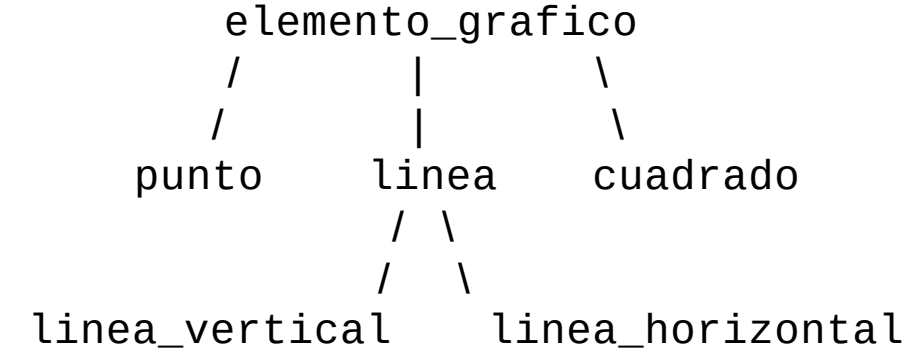
Emp: Carlos

Est: Jose

Presione cualquier tecla para continuar . . .

Ahora, al llamar a "Pepito->VerNombre(n)" se invoca a la función "VerNombre" de la clase "Estudiante", y al llamar a "Carlos->VerNombre(n)" se invoca a la función de la clase "Empleado"

3) Se quiere crear una jerarquía de elementos gráficos como en la siguiente figura:



Crear las clases necesarias para poder pintar los elementos gráficos de la jerarquía. Para completar la funcionalidad proporcionada por esta jerarquía, crear un array de objetos de la clase elementos_graficos y haga que se muestre por pantalla.

Polimorfismo

```
class elemento_grafico{
public:
virtual void pinta()const =0;
};

class punto : public elemento_grafico{
public:
void pinta()const{
cout << “.”;}
};

class linea: public elemento_grafico{
protected:
  int longitud;
public:
  linea (int longitud):linea(longitud){}
};
```

Polimorfismo

```
  class linea_vertical: public linea {
  public:
  linea_vertical(int longitud):linea(longitud){}

  void pinta()const{
   for(int i=0; i<longitud; i++)
      cout << “|” << endl;}
};

 class linea_horizontal: public linea {
  public:
  linea_horizontal(int longitud)
   :linea(longitud){}

  void pinta()const{
   for(int i=0; i<longitud; i++)
      cout << “-”;}
};
```

Polimorfismo

```
 class cuadrado: public elemento_grafico{
 protected:
 int lado;
 public:
 cuadrado(int lado):lado(lado){}
 void pinta() const {
 //pintar lado superior
 for(int i=0; i< lado; i++)
   cout << “-”;
   cout << endl;
 //pintar lados laterales
 for(int i=0; i<lado-2; i++){
    cout << “|”;
    for(int j=0; j<lado-2;j++)
     cout << “ ”;
    cout << “|”;
    cout << endl;
}
//pintar lado inferior
 for (int i=0; i<lado; i++){
 cout << “-”; }
}
};
```

Polimorfismo

Crear un array de punteros a elemento_grafico que referencien objetos de las clases derivadas distintos.

```
void main (){
  elemento_grafico * lista[4];

  lista[0] = new punto;
  lista[1] = new linea_horizontal(10);
  lista[2] = new linea_vertical(5);
  lista[3] = new cuadrado(6);

  for(int i=0; i<4; i++){
      lista[i]->pinta();
      cout << endl; }
  for (int i=0; i<4; i++)
     delete lista[i];
}
```

Polimorfismo

Este programa crea un array de punteros a elementos `elemento_grafico`, que debido a la herencia pueden apuntar a cualquier objeto de una clase que heredase directa o indirectamente de la clase `elemento_grafico`. A continuación, se llama al método `pinta()` particular de cada objeto, para conseguirlo el método `pinta()` de la clase `elemento_grafico` se ha declarado como **virtual.** Además, ya no se pueden crear objetos de la clase `elemento_grafico`, ya que la clase `elemento_grafico` es abstracta, por ser su método `pinta()` **virtual puro.**