Programación II
2011
Clases en C++

Objetivo de la Guía

Construcción de clases en C++

Bibliografía

http://cppreference.com y *http://www.cplusplus.com/*

Creación de clases en C++

LA CLASE VECTOR PLANO:

```
//   vector_plano.h
using namespace std;

class vector_plano{
  private:
     float ii;
     float jota;
     float modulo;
     float fi;
     bool cartesiano;
     void toCart();
     void toPolar();
  public:
     vector_plano(float x, float y, bool cart=true);
     float i();
     float j();
     float mod();
     float phi();
     vector_plano operator +(vector_plano x);
     vector_plano operator *(float x);
    friend ostream& operator <<(ostream &os, vector_plano v);
 };
```

En el ejemplo anterior se ve el archivo cabecera *vector_plano.h* .Es la definición, de una clase que implementaremos en C++. En la línea 5 aparece el modificador *private*. Todos los *miembros* que aparecen desde aquí hasta que aparece el modificador *public* pueden ser usados solamente por otros miembros de esta clase y por *friends*. Hay dos tipos de miembros: *data member* que recuerdan los *fields* de *pascal* y *function members* o métodos (*methtods*) que son las funciones que pertenecen a la clase. Los *friends* son funciones a las que, si bien no pertenecen a la clase, se les permite acceder a los miembros privados. (Existe una tercer clase

de miembros, los *protected*, que juegan un papel importante en la herencia pero es tema a desarrollar en Programación Orientada a Objetos).

Como no podemos acceder a los miembros de datos privados, no hay forma de cambiar el valor de las componentes de un vector y; si quiero acceder al valor de alguna de ellas debo usar las funciones `i(), j(), mod() y phi()`. A estos métodos se los conoce como métodos de recuperación.

Se sobrecarga (*overload)* al operador + para permitir la suma de 2 vectores. Se define un operador `friend` que sobrecarga al << (extracción) y permite enviar un resultado a un `ostream` (consola o archivo).

La definición del constructor del `vector_plano` puede recibir 2 o 3 parámetros. Los 2 primeros son 2 números reales y el tercero indica si está en forma cartesiana o en forma polar. Si no se especifica, se supone cartesiana por *default*. En realidad, esto lo deducimos del significado de los identificadores, para saberlo con certeza deberíamos ver como está construida la clase.

El código que sigue es un ejemplo acerca de como se usa la clase:

```
1. #include "includes.h"
2. #include "vector_plano.h" ,
3.
4. int main(){
5.vector_plano uno(3,4);
6.vector_plano dos(3,0.7,false);
7. cout << "La suma de "<<uno<<\n mas "<<os<<"\n
  es"<<uno+dos<<endl;
8. }
```

Se crea un vector y está dado en coordenadas cartesianas, el segundo vector está dado en polares. Se hace uso del operador *friend*. Cuando se pide la impresión de `uno+dos` se está pidiendo la impresión de un *nuevo* `vector_plano` sin nombre cuyo valor sea la suma de uno mas dos. La definición del operador + deberá manejar las conversiones en forma interna y ocultar estos detalles a quien usa la clase vector plano (principio de *information hiding* propio de los ADT).

El archivo `includes.h` tiene los `#include` necesarios y es el que sigue.

```
1. #include <string>
2. #include <iostream>
3. #include <cmath>
```

Y, finalmente, la implementación de los métodos de la clase `vector_plano.cc`:

```
1.//    vector_plano.cc
2.#include "includes.h"
3.#include "vector_plano.h"
4.
5.vector_plano::vector_plano(float x, float y, bool cart){
6.   if (cart){ ii=x; jota=y;}
```

```
7.        else {modulo=x; fi=y;}
8.   cartesiano=cart;
9.}
10.
11.float vector_plano::i(){toCart();return ii;}
12.float vector_plano::j(){toCart();return jota;}
13.float vector_plano::mod(){toPolar();return modulo;}
14.float vector_plano::phi(){toPolar();return fi;}
15.
16.void vector_plano::toPolar(){
17.  if (cartesiano){
18.       modulo=sqrt(ii*ii+jota*jota);
19.       fi=atan2(jota,ii);
20.       cartesiano=false;
21.       }
22.}
23.
24.void vector_plano::toCart(){
25.  if (!cartesiano){
26.       ii=modulo*cos(fi);
27.       jota=modulo*sin(fi);
28.       cartesiano=true;
29.       }
30.}
31.
32.vector_plano vector_plano::operator+(vector_plano x){
33.  toCart();
34.  x.toCart();
35.  return vector_plano(ii+x.i(),jota+x.j());
36.}
37.
38.vector_plano vector_plano::operator*(float x){
39.  toPolar();
40.  return vector_plano(modulo*x,fi,false);
41.}
42.
43.ostream& operator <<(ostream &os, vector_plano v){
44.  v.toCart();
45.  os <<"i="<<v.i()<<";j="<<v.j()<<"(";
46.  v.toPolar();
47.  os <<"modulo="<<v.mod()<<";angulo="<<v.phi()<<" rad o "
48.       <<v.phi()*360/(2*M_PI)<<" grados)";
49.  return os;}
50.
```

Los métodos que pertenecen a la clase `vector_plano` aparecen precedidos por un

operador de ambiente o *scope* `::` .Este operador sirve para diferenciar a un método, por ejemplo `phi()` de la clase `vector_plano`, de algún otro de alguna otra clase (por ejemplo `complejo`) que pudiera existir. También nos recuerdan que deben ser llamados en el contexto de un objeto como por ejemplo `a.phi()` .

Los métodos de conversión no pueden ser usados por el público en general (llamados clientes de la clase). La sobrecarga al operador `+` pasa ambos vectores a coordenadas cartesianas y crea uno nuevo con su suma, antes se pasa a cartesianas *éste* `vector_plano` (el objeto en cuyo contexto está llamado el método). El operador `+` usa notación infija, el contexto es el del primer objeto del par que interviene en la suma después se pasa a cartesianas al segundo objeto, `x`. Por tratarse de un método de la clase `vector_plano`, puede acceder a los métodos privados como `toCart()` aunque sean de otro objeto.

Finalmente el operador de extracción permite una impresión significativa en cartesiana, polares con radianes y grados.

Desarrollo en Linux.

Para usar la clase se debe primero compilar `vector_plano.cc` con el comando:

```
g++ -c vector_plano.cc
```

Obteniendo `vector_plano.o` esto también se puede lograr con el comando

```
make vector_plano.o
```

Y el `make` buscará un archivo con una extensión que le sea conocida (y `.cc` lo es) y llamará al compilador correspondiente.

Cuando compilamos el programa de prueba (`pruvec.cc`) debemos indicarle al *link editor* donde se encuentra la clase `vector_plano`. Recién la compilamos y la guardamos en `vector_plano.o` . Así que usamos el comando

```
g++  pruvec.cc -o pruvec vector_plano. o
```

Observar que el archivo con la clase va después de las opciones. Para hacer esto con *make* debemos escribir las reglas en un archivo, pero esto queda para la próxima guía.

Algunos detalles

Si se incluyen dos veces un mismo *header*, es común que aparezcan errores, en general en los *headers* de las bibliotecas.

Hay tres tipos de constructores:

Default o constructor sin parámetros (no hace falta declararlo, salvo qeu se quiera dejar bien definidos los *data members*.

Constructor con parámetros.

Copy constructor que recibe un objeto del mismo tipo y los copia.

Los constructores no tienen tipo y tienen el mismo nombre que la clase.

Los destructores se invocan cuando un objeto es desalojado de la memoria.

Siguiendo la ejecución del siguiente ejemplo, explicar donde y porqué ocurre cada una de sus impresiones:

```
1.//  includes.h
```

```
2.#include <iostream>
3.#include <string>
4.using namespace std;

1.// Cosa h
2.#include "includes.h"
3.
4.class Cosa{
5.  private:
6.    string s;
7.    int n;
8.  public:
9.  Cosa();
10.  Cosa(string s1,int n);
11.  Cosa(Cosa&c);
12.  ~Cosa();
13.  string getS(){return s;};
14.  int getN(){return n;};
15.  friend istream& operator >>(istream& is,Cosa& c);
16.  friend ostream& operator <<(ostream& os,Cosa c);
17.};

1.// Cosa.cpp
2.#include "cosa.h"
3.
4.Cosa::Cosa(){
5.  n=0;s="";
6.  cout<<"Constructor sin parametros"<<endl;
7.}
8.Cosa::Cosa(string s1, int n1){
9.  s=s1;n=n1;
10.  cout<<"Constructor con parametros s="<<
11.  s<<" y n="<<n<<endl;
12.}
13.Cosa::Cosa(Cosa &c){
14.  n=c.n;
15.  s=c.s;
16.  cout<<"Constructor de copia s="<<c.s<<endl;
17.}
18.Cosa::~Cosa(){
19.cout<<"Destructor s="<<s<<endl;
20.}
21.istream& operator>>(istream& is,Cosa& c){
22.  return is>>c.s>>c.n;
23.}
```

```
24.ostream& operator<<(ostream& os,Cosa c){
25.  return os<<"s="<<c.s<<" ,n="<<c.n;
26.}

1.// main.cpp
2.#include "cosa.h"
3.
4.void swap (Cosa&c1, Cosa&c2){
5.    Cosa aux=c2;
6.    c2=c1;
7.    c1=aux;
8.    }
9.int main(){
10.  Cosa c1;
11.  Cosa c2("c-dos",2);
12.  Cosa c3(c2);
13.  Cosa c4("c-cuatro",4);
14.  cout<<"Estan c1=("<<c1<<") c2=("<<c2<<") c3=("<<c3
15.        <<" y c4=("<<c4<<")\n";
16.    swap (c2,c4);
17.    cout <<"Ahora c2=("<<c2<<") y c4="<<c4<<")\n";
18.    system("pause");
19.}
```

Bien, es hora de ponernos a programar:

Extendiendo la clase `vector_plano`

Ud. deberá añadir los siguientes métodos a la clase `vector_plano`, Para ello deberá modificar tanto el archivo *.h como el *.cc y recompilar todo. Por cada método nuevo programe un *driver* y un *tester*.

1. Añada métodos de acceso que permitan modificar alguna componente del `vector_plano`. Estos métodos se llama generalmente `setPhi(...)` , `setJ(...)` etc, (a los métodos de recuperación se les suele anteponer `get` como `getPhi(...)` ...). Cuide de mantener la coherencia de la representación (de nada sirve modificar un ángulo de un vector que está en cartesiano, *primero* se lo pasa a polar y *luego* se modifica).
2. Añada un método que indique en que cuadrante está el vector.
3. Añada un método para multiplicar un vector por un escalar.
4. Añada un método para el producto interno.

Ahora ya debe estar en condiciones de crear sus propias clases. Pruebe entonces:

1. Crear una clase *instante* para operar con horas, minutos y segundos. Debe permitir sumar y restar instantes y obtener el valor en horas con decimales, minutos con decimales y en segundos.

2. A la clase anterior póngale además métodos para comparar instantes.
3. ¿Recuerda la aritmética de polinomios? haga una clase para manejar polinomios de hasta tercer grado.

Un poco de búsqueda en la Internet le va a ayudar a conocer algunas cosas divertidas ya crear clases para

1. Sumar y restar longitudes que pueden venir expresadas en cm, Pies Ingleses, Pies Sajones, yardas, codos sagrados (*sacred cubit*), también obtener su valor en alguno de esos "sistemas".
2. Cuando la precisión es limitada (por ejemplo, solo se pueden representar números entre O y 255) hay dos soluciones sin recurrir a los *overflow* o *underflow*: aritmética modular y aritmética de saturación. Impleméntelas en dos clases (operaciones de acceso, recuperación, suma y multiplicación)
3. Programe una clase capaz de guardar las coordenadas de un punto en el plano. Una operación deberá permitir obtener la *Distancia Manhattan* entre dos puntos.
4. Programe una clase estadística, para la que se tiene la siguiente cabecera:

```
1.{ class Estadistica {
2.   public:
3.   Estadistica( );
4.   void prox(double r)1 // ingresa un numero a las
  estadisticas.
5.   void reset( ); // borra las estadisticas
6.   int cuenta( ) ; // cuantos numeros se ingresaron
7.   double sum( );      double media ( ) ;
8.   double minimo ( ) ;   double maximo( ) ;
9.   Estadistica operator +(estadistica el);
10.   friend bool operator ==(estadistica sl,estadistica
  s2);
11.   private:
12.  ......queda a su criterio.....
13.}
```

1. Defina una clase punto que guarde las coordenadas de un punto en el plano. Dado un conjunto pequeño *n* de puntos (*n<100)* encontrar cuál es el punto en el cual la suma de las distancias al resto de los puntos es mínima.
2. Defina una clase segmento con lugar para 2 puntos. Defina una clase arreglo que pueda contener de 0 a 30 segmentos. Escriba una función que indique si los puntos en ese arreglo forman un polígono convexo. Además una función que indique si un punto está o no dentro del polígono.
3. Defina una clase *rectángulo* dado por un punto, su altura y su ancho. Suponga positivo el sentido de los ejes en el primer cuadrante. Escriba una función para saber si un punto

está dentro del rectángulo, otra para saber si un segmento intersecta al rectángulo, otra para saber si un rectángulo intersecta a otro y otra para saber si un rectángulo está contenido en otro.

4.