Plantillas de clases (Templates)

1. Plantilla de funciones o funciones genéricas
2. Métodos genéricos

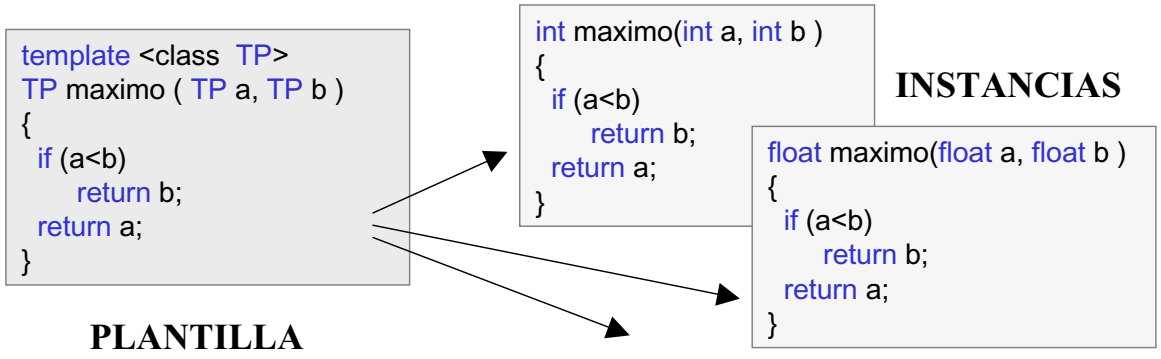
3. Plantillas de clases o clases genéricas
4. Sintaxis
5. Creación de objetos
6. Representación en UML
7. Miembros de las clases genéricas
8. Consejos

Plantillas de clases (Templates)

Plantillas de funciones (funciones genéricas)

Hemos visto anteriormente, que las funciones genéricas son un mecanismo C++ que permite definir una función mediante uno o varios parámetros (tipos genéricos). A partir de estas plantillas el compilador es capaz de generar código de funciones distintas que comparten ciertas características.

Las funciones así generadas se denominan instancias o especializaciones de la plantilla.



Métodos genéricos

Las funciones genéricas pueden ser miembros (métodos) de clases.

```
class MyClase
{
   public:
      template <class  Tipo>
      Tipo maximo ( Tipo a, int b );
   private:
        …
        …
};
```

Prototipo

```
template <class  Tipo>
Tipo Myclase::maximo ( Tipo a, Tipo b )
{
  if (a<b)
       return b;
  return a;
}
```

Implementación

Ejemplo

```
class ClaseA
{
   private:
        int x ;
   public:
      ClaseA (int a);

      template <class  Tipo>
      void func ( Tipo a, Tipo b );
};
```

```
ClaseA :: ClaseA ( int a )
{
  x = a;
}
```

```
template <class  Tipo>
void  ClaseA::func ( Tipo a, Tipo b )
{
  cout  << a ;
  cout  << b ;
  cout  << x ;
}
```

```
void main( )
{
 ClaseA obj1(4), obj2(5);
 obj1.func ( 'm' , 'k' );
 obj2.func ( 100, 200 );
}
```

Salida

```
m , k , 4
100, 200, 5
```

Plantillas de clases o Clases genéricas

Son un artificio C++ que permite definir una clase mediante uno o varios parámetros. Este mecanismo es capaz de generar infinitas clases distintas pero compartiendo un diseño común.

Al igual que en plantillas de funciones, las clases así generadas se denominan instancias o especializaciones de la plantilla.

```
class Punto
{
   private:
      int coorx ;
      int coory ;
   public:
      ...
} ;
```

Podemos diseñar la clase Punto con atributos de tipo **int**

En el caso de que necesitemos que el tipo de los puntos sea de otro tipo, tendremos que crear otra clase distinta e implementar todos los métodos.

Pérdida de tiempo

Posibilidad de error

Plantillas de clases o Clases genéricas

```
class Puntoentero
{
   private:
      int coorx ;
      int coory ;
   public:
      ...
} ;
```

El sistema de plantillas nos permite definir una clase genérica que permita crear especializaciones o instancias de dicha plantilla para cualquier tipo especificado por un parámetro.

INSTANCIAS

```
class Puntofloat
{
   private:
      float coorx ;
      float coory ;
   public:
      ...
} ;
```

```
class Puntolong
{
   private:
      long int coorx ;
      long int coory ;
   public:
      ...
} ;
```

```
template <class  TIPO>
class Punto
 {
    private:
       TIPO coorx ;
       TIPO coory ;
    public:
       ...
 } ;
```

PLANTILLA

Sintaxis

Las plantillas de clases o clases genéricas tienen la siguiente sintaxis:

```
template <lista de tipos genéricos>
class Nombre_clase
{
  …
};
```

La *lista de tipos* contiene los tipos genéricos a los que precede la palabra **class**

- La palabra reservada template indica que se va a declarar una plantilla.
- Las plantillas se declaran normalmente en un archivo de cabecera.
- La *lista de tipos* contiene los tipos genéricos separados por comas.

Creación de objetos

¿Cómo se crea un objeto de la clase Punto con atributos de tipo float?

```
template <class  TIPO>
class Punto
 {
   private:
     TIPO coorx ;
     TIPO coory ;
   public:
      ...
 } ;
```

PLANTILLA

```
 ...
Punto <float> obj ;
...
```

Se escribe el nombre de la plantilla de clase seguido por los tipos con los que se declara entre < >

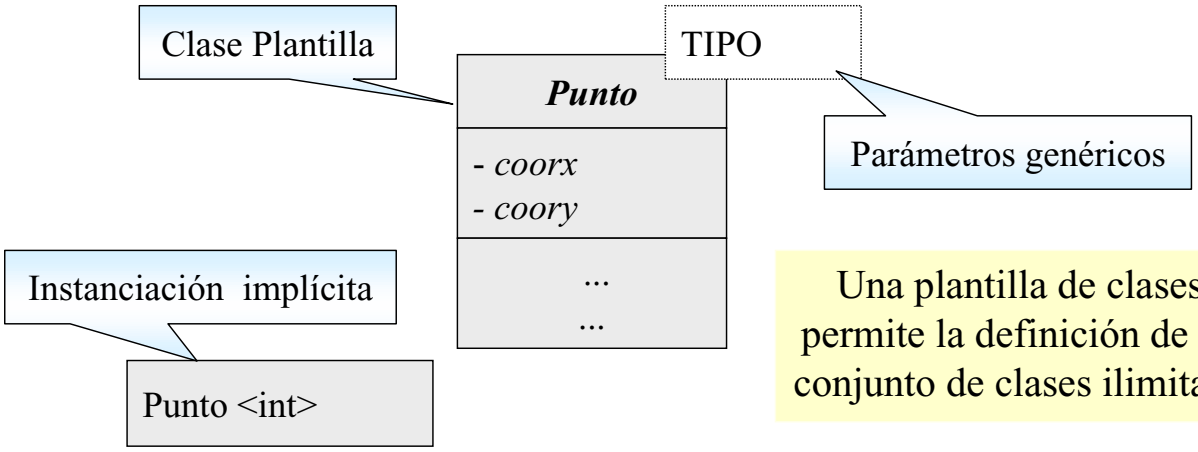
```
 class Punto
{
   private:
     float coorx ;
     float coory ;
   public:
      ...
} ;
```

INSTANCIA

Esta instrucción genera una clase normal a partir de la plantilla de clase

Instanciación implícita

Representación de una plantilla de clase en UML



Una plantilla de clases permite la definición de un conjunto de clases ilimitada

Podemos, al igual que con las plantillas de funciones, generar todas las clases que queramos.

```
...
Punto <float> obj1 ;
Punto <int> obj2 ;
Punto <long int> obj3 ;
...
```


Miembros de las clases genéricas

Los miembros de las clases genéricas se definen y declaran igual que los de las clases normales.

```
template <class  TIPO>
class Punto
 {
   private:
     TIPO coorx ;
     TIPO coory ;
   public:
      Punto(TIPO a, TIPO b);
      TIPO acc_x( );
      TIPO acc_y( );
      void mu_x( TIPO a);
      void mu_x( TIPO a);
      void visualizar( );
} ;
```

Hay que señalar que la funciones miembro, son a su vez plantillas con los mismos parámetros que la clase genérica a la que pertenecen.

```
template <class  TIPO>
Punto<TIPO>::Punto(TIPO a, TIPO b)
{
   coorx = a;
   coory = b;
}
```

```
template <class  TIPO>
TIPO Punto<TIPO>::acc_x( )
{
   return coorx;
}
```

Miembros de las clases genéricas

```
template <class  TIPO>
class Punto
 {
    private:
       TIPO coorx ;
       TIPO coory ;
    public:
        Punto(TIPO a, TIPO b);
        TIPO acc_x( );
        TIPO acc_y( );
        void mu_x( TIPO a);
        void mu_x( TIPO a);
        void visualizar( );
 } ;
```

```
template <class  TIPO>
void Punto<TIPO>::mu_x(TIPO a )
{
    coorx = a;
}
```

```
template <class  TIPO>
void  Punto<TIPO> :: visualizar( )
{
    cout << coorx ;
    cout << coory;
}
```


CONSEJOS

- Es aconsejable realizar el diseño y una primera depuración con una clase normal antes de convertirla en una clase genérica.

  Es más fácil imaginarse el funcionamiento referido a un tipo concreto que a entes abstractos.

- Es más fácil entender los problemas que pueden presentarse si se maneja una clase concreta.

- En estos casos es más sencillo ir de lo particular a lo general.

Ejemplo

```
template <class  T1, T2>
class ClaseB
{
   private:
      T1 x ;
      T2 y;
   public:
       ClaseB (T1 a, T2 b);
       void visualizar ( );
};
```

```
template <class  T1, T2>
ClaseB< T1, T2> :: ClaseB (T1 a, T2 b)
{
  x = a;
  y = b;
}
```

```
template <class  Tipo>
void  ClaseB < T1, T2> :: visualizar ( );
{
  cout  << "Valor de x : " ;
  cout  << x << endl ;
  cout  << "Valor de y : " ;
  cout  << y ;
}
```

```
void main( )
{
 ClaseB <int, char> obj1(5, 'p') ;
 ClaseB <bool, float> obj2(true, 9.0) ;
 obj1.visualizar( );
 obj2.visualizar( );
}
```

Salida

```
Valor de x:5
Valor de y: p
```