# ABSTRACCION : GENERALIZACION

MECANISMOS PARA LA ABSTRACCIÓN:

- POR ESPECIFICACIÓN
- POR GENERALIZACIÓN

. A partir de varios objetos, extraer características comunes que definen una generalización más fácil de manejar

Un ejemplo de ambos tipos es la abstracción funcional:

. Se crea una especificación para obviar los detalles de implementación de una función

. Los parámetros de una función constituyen una generalización en la que se integran todas las posibles ejecuciones sobre los distintos valores de los parámetros

* GENERALIZACION : ABSTRACCION POR PARAMETRIZACION

Funciones patrón:

```
void intercambiar (int & a, int & b) {int aux=a; a=b; b=aux;}
void intercambiar (float & a, float & b) {float aux=a; a=b; b=aux;}
void intercambiar (string & a, string & b) {string aux=a; a=b; b=aux;}
void intercambiar (T & a, T & b) {T aux=a; a=b; b=aux;}
```

¿Cómo se hace esto en C++?

↓

Con los **templates** (plantillas) que no son mas que un mecanismo de abstracción por parametrización.

```
template < class T >
    void intercambiar (T & a, T & b) {T aux = a; a = b; b = aux;}
```

Otro ejemplo: ordenación de un vector

```
template < class T >
    void ordenar_seleccion (T * vector, int n)
    {
     int i, minimo;
     for (i=0; i<n-1; i++){
          minimo = i;
          for (j=i+1; j<n; j++)
               if (vector[j] < vector[minimo])
                    minimo = j;
          intercambiar (vector[i], vector[minimo]);
    }
}
```

# Clases patrón

Establecemos para las clases el mismo mecanismo de generalización que para las funciones.

```
template <class T>
    class Vector_Dinamico{
        private:
          T * datos;
          int nelementos;
        public:
          Vector_Dinamico<T>(int n=0);
          Vector_Dinamico<T> (const Vector_Dinamico<T>
                                        & original);
          ~Vector_Dinamico<T>();
          int size() const;
          T & operator[] (int i);
          const T & operator[] (int i) const;
          void resize (int n);
          Vector_Dinamico<T> & operator = (const Vector_Dinamico<T>
                                        & original);
    };
```

```
Vector_Dinamico<int> valores;
Vector_Dinamico <string> nombres;
Vector_Dinamico <Polinomio> polinomios;
```

## Definición de los métodos

```
template <class T>
Vector_Dinamico<T>::Vector_Dinamico(int n)
{
  assert(n>=0)
  if (n>0)
    datos = new T[n];
  nelementos=n;
}
```

## Plantillas y compilación separada

Incluir el .cpp en el .h

Otras soluciones:

Incluir TODAS las posibles instanciaciones que se deseen al final del .cpp

## Compatibilidad del tipo base en la instanciación

No todas las instanciaciones son posibles

Ejemplo:

Llamar a ordenar_seleccion con T ≡ Polinomio → no se sabe comparar polinomios: ¿Como definir operator< en la clase polinomio?

# Abstracción por parametrización(1/3).

## *Funciones Patrón*

*Funciones idénticas excepto en el tipo de dato*

```
void intercambiar (int& a, int& b) { int aux= a; a= b; b= aux; }
void intercambiar (float& a, float& b) { float aux= a; a= b; b= aux; }
void intercambiar (string& a, string& b) { string aux= a; a= b; b= aux; }
```

*Parametrizamos el tipo de dato mediante una Plantilla*

```
template <class T>
void intercambiar (T& a, T& b) { T aux= a; a= b; b= aux; }
```

*Uso de la plantilla desde otra función genérica*

```
template <class T>
void ordenar_seleccion (T *vector, int n)
{
  int i,minimo;

  for (i=0;i<n-1;i++) {
    minimo= i;
    for (j=i+1;j<n;j++)
      if (vector[j]<vector[minimo])
        minimo= j;
    intercambiar(vector[i],vector[minimo]);
  }
}
```

## *Clases Patrón*

*Parametrización del tipo base de una clase*

```
template <class T>
class Vector_Dinamico {
  private:
    T * datos;
    int nelementos;
  public:
    Vector_Dinamico<T>(int n);
    Vector_Dinamico<T>(const Vector_Dinamico<T>& original);
    ~Vector_Dinamico<T>();
    int size() const;
    T& operator[] (int i);
    const T& operator[] (int i) const;
    void resize(int n);
    Vector_Dinamico<T>& operator=
                        (const Vector_Dinamico<T>& original);
};
```

*Instanciación de un tipo concreto con la declaración*

```
Vector_Dinamico<int> valores;
Vector_Dinamico<string> nombres;
Vector_Dinamico<Polinomio> polinomios;
```

# Abstracción por parametrización(2/3).

*Definición de los métodos*

```
template<class T>
Vector_Dinamico<T>::Vector_Dinamico<T>(int n[redacted])
{
  assert(n>=0);
  if (n>0)
    datos= new T[n];
  nelementos= n;
}

template<class T>
Vector_Dinamico<T>&
      Vector_Dinamico<T>::operator=
              (const Vector_Dinamico<T>& original)
{
  if (this! = &original) {
    if (nelementos>0) delete[] datos;
    nelementos= original.nelementos;
    datos= new T[nelementos];
    for (int i=0; i<nelementos;++i)
      datos[i]= original.datos[i];
  }
  return *this;
}
```

*Ejemplo de uso*

```
template <class T>
void ordenar_seleccion (Vector_Dinamico<T>& vector)
{
  int i,minimo;

  for (i=0;i<vector.size()-1;i++) {
    minimo= i;
    for (j=i+1;j<vector.size();j++)
      if (vector[j]<vector[minimo])
        minimo= j;
    intercambiar(vector[i],vector[minimo]);
  }
}
```

# Abstracción por parametrización(3/3).

*Clase Par*

```
#ifndef _utilidades_h
#define _utilidades_h

template <class T1, class T2>
struct Par {
    T1 primero;
    T2 segundo;
    /* ——— Operaciones ——— */
    Par(): primero(T1()),segundo(T2()) {}
    Par(const T1& p, const T2& s): primero(p),segundo(s) {}
    Par(const Par& p): primero(p.primero),segundo(p.segundo) {}
    template <class U1, class U2>
        Par(const Par<U1,U2>& p):
primero(p.primero),segundo(p.segundo) {}
    ~Par() {}
    Par& operator= (const Par& v)
      {primero=v.primero;segundo=v.segundo; return *this; }
    bool operator==(const Par& s) const
      {return primero==s.primero && segundo==s.segundo;}
    bool operator! =(const Par& s) const
      {return primero! =s.primero || segundo! =s.segundo;}
};

#endif                                  /* _utilidades_h */
```

*Ejemplo de uso*

```
#include <iostream>
#include <par.h>
using namespace std;

template<class T>
void intercambiar (T& a, T& b) { T aux= a; a= b; b= aux; }

template <class T>
void ordenar(Par<int,T>& a, Par<int,T>& b)
{
  if (a.primero>b.primero)
    intercambiar(a,b);
}

int main()
{
  Par<int,float> v1,v2;

  v1.primero=1; v1.segundo=2.0;
  v2.primero=0; v2.segundo=5.0;
  ordenar(v1,v2);
  cout ≪ ''El primero es
(''≪v1.primero≪'',''≪v1.segundo≪'')''≪endl;
  return 0;
}
```