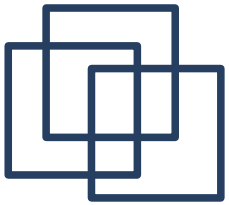
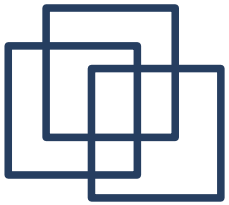
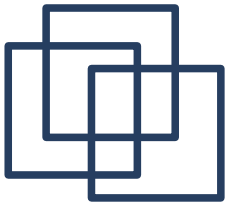


Abstracción de Datos



Tema 2. Abstracción

- Introducción
- Abstracción en programación
- Abstracciónes procedimentales
- Abstracción de datos: Tipo de Dato Abstracto(TDA)
  - TDA en C++
- Abstracción por generalización
- Abstracciónes de iteración:
  - Contenedores e iteradores



Abstracción por generalización.

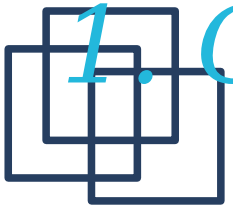
Generalizar: Abstraer lo que es común y esencial a muchas cosas, para formar un concepto general que las comprenda todas. (Diccionario RAE).

1.) Generalización de abstracciones funcionales.

Definir familias de funciones con idéntico procesamiento, independientemente de los datos que utilicen.

2.) Generalización de tipos de datos.

Definir familias de tipos de datos con una especificación común, independiente de los tipos particulares que se utilicen.



1. Generalización de abstracciones funcionales

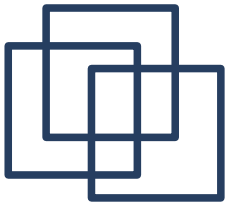
Consideremos el siguiente código

```
int minimo_enteros(int a, int b) { return (a < b ? a : b); }
```

La forma de calcular este valor es válida para cualquier par de valores para los que esté definida la operación <,

```
float minimo_float(float a, float b) { return (a < b ? a : b); }
```

```
fecha minimo_fecha(fecha a, fecha b) { return (a < b ? a : b); }
```

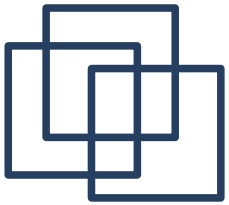


Generalización de abstracciones funcionales

Los mecanismos que ofrece C++ para permitir la generalización son dos:

a) Sobrecarga de funciones:

b) Uso de funciones patrón *(template)*

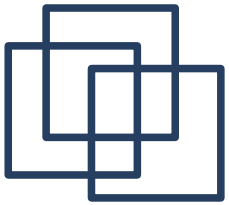


1.a) Sobrecarga de funciones

Permite que distintas funciones tengan el mismo nombre, lo único que se exige es que se puedan diferenciar mediante los argumentos que se les pasan, en número o tipo

- 1) float minimo(float a, float b) { return (a < b ? a : b); }
- 2) fecha minimo(fecha a, fecha b) { return (a < b ? a : b); }
- 3) racional minimo(racional a, racional b) { return (a < b ? a : b); }

```
float x = minimo(2.5,7.3)   // llama a la función 1
racional z = minimo( racional(4,5), racional(7,2)); // llama a la función 3
```



1.b) Funciones patrón (template)

Permiten hacer funciones genéricas, independiente del tipo de datos con el que se pueda usar.

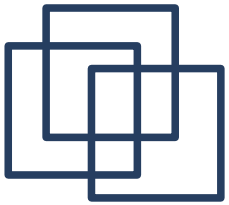
Se requiere que el código de la función idéntico para todos los tipos de datos y que éstos tengan definidas las operaciones que se utilicen.

Ventaja: Sólo se implementan una vez**.**

template <parámetros> declaración

Los parámetros de tipo de dato se designan como

typename T



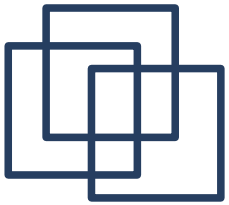
1.b) Funciones patrón (template)

```
template <typename T> T minimo(T a, T b)
{  return (a < b ? a : b); }
```

Instanciación: Cada vez que se necesita una función específica (p.ej: *minimo* para *int*), se particulariza la función genérica

```
int a,b,c;
c = minimo(a,b);
```

```
float d,e,f;
f = minimo(d,f);;
```



Ejemplo Generalización

/** Buscar menor elemento de un array

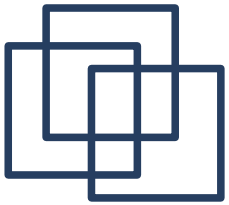
@param array: conjunto de elmentos

@param num_elems: número de componentes > 0

@param precondition debe estar definida la operación < para T

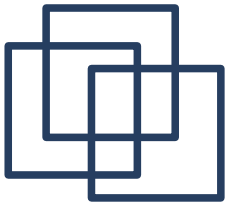
@return el menor elemento del array */

```
template<typename T>
T minimo<T array[ ], int num_elems);
```



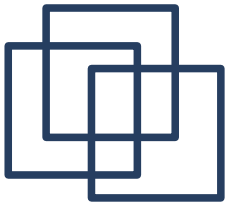
Ejemplo Generalización

```
template<typename T>
T minimo( T array[ ], int num_elems){
    T menor = array[0];
    for (int i=1; i< num_elems; i++)
            { if (array[i] < menor) menor =
     array[i]; }
    return menor;
   }
```



Instanciación:

```
int main (   ){
 int V[10];
float Z[100];
.....
int x = minimo(V,10);
float v = minimo(Z,100);
......
}
```



Ejemplo Generalización

```
/**   Dispone los elementos de 'array' en sentido
   creciente.

      @param array: array de elementos a ordenar.

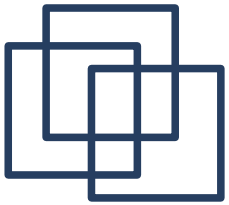
      @param num_elems: número de componentes  > 0.

      @precondition Debe estar definido el operator <
      para T

 */ template <typename T>

   void ordena(T  array[ ], int num_elems)

 {.....  código; .....}
```



Instanciación:

int V[100]; float AF[500]; racional R[1000]; fecha FE[200];

ordena(V,100); ordena(AF,500);

ordena(R,1000); ordena(FE,200);

Generalización en C++
Particularización de funciones

```
template < typename T>
T maximo( const T & a, const T
    & b)
{  return (a>b)?a:b;  }

funcionX( ){
int i1,i2;
float f1,f2;
racional r1,r2;
alumno a1,a2;
....
cout << maximo(i1,i2);
....
cout << maximo(f1,f2);
....
cout << maximo(r1,r2);
....
cout << maximo(a1,a2);
...}
```

precompilador

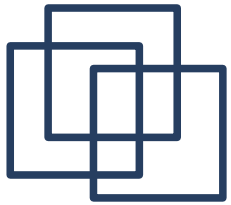
```
int maximo( const int & a, const int
    & b)
{  return (a>b)?a:b;  }

float maximo( const float & a, const
    float & b)
{  return (a>b)?a:b;  }

racional  maximo( const racional &
    a, const racional & b)
{  return (a>b)?a:b;  }

alumno maximo( const alumno & a,
                    const alumno & b)
{  return (a>b)?a:b;  }

funcionX( ){
int i1,i2;  float f1,f2; racional r1,r2;
    alumno a1,a2;
....
cout << maximo(i1,i2);
....
cout << maximo(f1,f2);
....
```



Generalización en C++
Particularización de funciones

- Es necesario que el **precompilador** conozca el codigo c++ de la funcion::

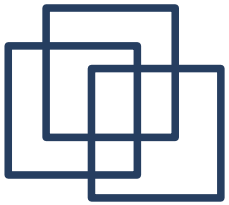
utilidad.h

```
template <typename T>
T maximo( const T & a, const T & b)
{return ( (a>b)?a:b);}

template <typename T>
T minimo ( const T & a, const T & b)
{return ( (a>b)?a:b);}
```

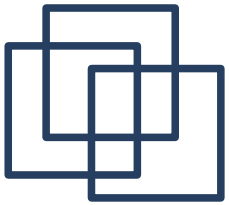
uso.cpp

```
#include “utilidad.h”
void funcionX(){
int i1,i2;
……
cout << maximo(i1,i2);
…
}
```



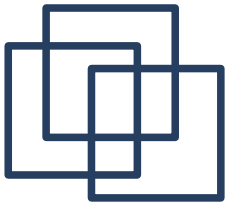
Programación genérica, un paso más

- Deseamos realizar un funciones que permitan ordenar un array de alumnos.
  - Orden Creciente
  - Orden Decreciente



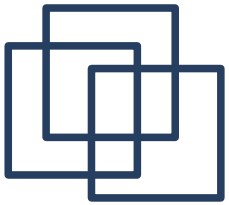
Ejemplo: Ordenación

```
void ordenar_creciente(alumno a[],int n){
int i,j,im
alumno aux;
 for(i=0;i<n-1;i++){
    im=i;
    for(j=i+1;j<n;j++)
      if (a[j] < a[im]) im=j;  // Comparar
   aux=a[i]; a[i]=a[im];  a[im]=aux;  // swap
  }
}
```



Ejemplo: Ordenación

```
void ordenar_decreciente(alumno a[],int n){
int i,j,im
alumno aux;
 for(i=0;i<n-1;i++){
   im=i;
   for(j=i+1;j<n;j++)
     if (a[j] > a[im]) im=j;  // Comparar
   aux=a[i]; a[i]=a[im];  a[im]=aux;  // swap
  }
}
```



Functores y programación genérica

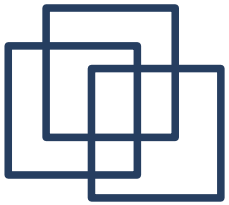
- La diferencia esencial entre ambas
  - Ordenar_creciente
  - Ordenar_decreciente

está en el criterio utilizado a la hora de comparar

Idea clave?

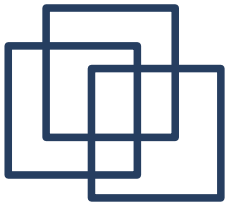
pasar el criterio de comparación como parámetro del algoritmo.

```
ordenar(V,10, creciente);
ordenar(V,10,decreciente);
```



Functor: function operator

```
template <class comparar >
void ordenar(alumno a[],int n, comparar cmp){
int i,j,im
alumno aux;
    for(i=0;i<n-1;i++){
       im=i;
       for(j=i+1;j<n;j++)
          if (cmp(a[j],a[im])) im=j;  // Comparar
    }
aux=a[i]; a[i]=a[im];  a[im]=aux;  // Intercambio
}
```



Qué es un functor

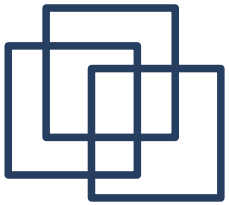
- Es un objeto función: objeto de una clase que tiene sobrecargado el operador – operator() -
  - Permite usarlo como una llamada a función
  - Se realiza como función miembro de una clase
  - Puede tener un número de parámetros variables

```
class comp_crec{

   public:

      bool operator() (const alumno & a, const
       alumno & b) const;

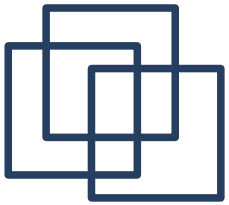
      };
```



Implemetación functor

- ….

```
bool comp_crec::operator()(const alumno & a,
                           const alumno & b) const
{
  return  ( a < b)
};
```

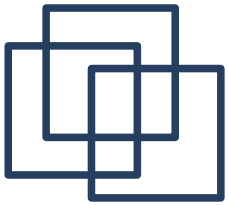


Uso del functor

```
comp_crec creciente;
comp_decrec  decreciente;
```

- creciente no es función sino un objeto de la clase sumador.
- El objeto creciente se comporta como una función, pero tiene la ventaja de que se puede pasar como párametro a otras funciones!!!

```
ordenar(V,10,creciente);
ordenar(V,10,decreciente);
ordenar(V,10, comp_crec() );
```



Orden alfabetico creciente?

```
class alf_creciente {

  public:

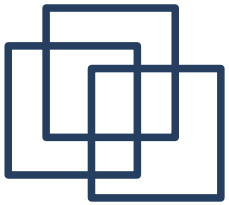
     bool operator()(const alumno & a, const
  alumno & b) const;

}

bool alf_creciente::operator()(const alumno &
  a, const alumno & b) const

{ return (a.getApellido()<b.getApellido());}


ordenar(V, 10, alf_creciente() );
```



Ejemplo: Mínimo array

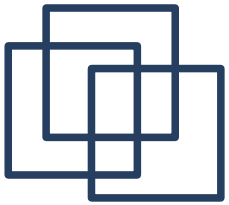
- Deseamos realizar una función que permita calcular el menor elemento de un array.
- Queremos utilizar distintos criterios para definir el menor entre dos elementos.

Solución, pasar la función de comparación como plantilla.

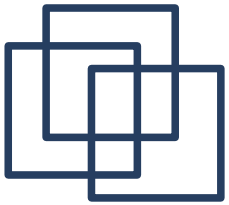
```
// Declaracion de la funcion minimo

template<class cmp>

 int minimo(int  array[], int n, cmp c );
```

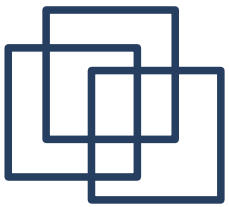


```
//Implementacion de la funcion minimo
template<class cmp>
int minimo(int array[], int n, cmp c )
{
  int menor =array[0];
  for (int  i=1;i<n;i++)
        if ( c(array[i],menor) )  menor = array[i];
  return menor;
 }
```



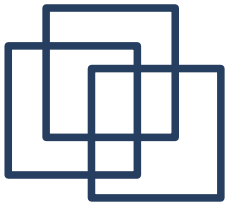
Functores y programación genérica.

```
class menor  {       //  functor C++
public:
    bool operator()( const int & A, const int & B );
};
 bool menor:: operator()( const int & A, const int & B )
{
     return (A<B); //cualquier criterio de comparacion
}
...
```



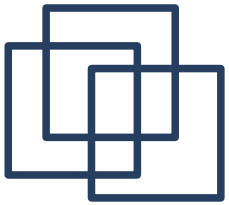
Functores y programación genérica.

```
class menor_valor_absoluto  {       //  functor C++
public:
    bool operator()( const int & A, const int & B );
};
 bool menor_valor_absoluto:: operator()( const int & A, const int & B )
{
     return (abs(A)<abs(B)); //cualquier criterio de comparacion
}
...
```



Uso del functor

```
menor criterio1;
    //Objeto de la clase menor
cout << minimo(A,10,criterio1) <<endl;
....
menor_valor_absoluto criterio2;
    //Objeto menor_valor_absoluto que utilizaria
      un criterio distinto
cout << minimo(A,10,criterio2) << endl;
```

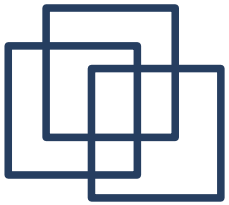


Problema:

- ordenar sólo funciona para alumno
- mínimo sólo funciona para int,

Solución:

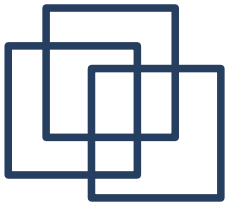
Usar plantillas para generalizarlo a cualquier tipo de datos



2. Generalización de tipos de datos.

Se aplica en situaciones donde la especificación, representación e implementación de distintos tipos de datos son todas iguales, salvo por la naturaleza específica de los elementos que lo componen.

Por tanto, podemos pensar en abstraer las características comunes.



2. Generalización de tipos de datos.

- Pares de elementos, posiblemente de tipo distinto.

  racional es un par de enteros (numerador, denominador)

  llamada es un par de enteros (hora, minuto)

  entrada_diccionario es un par de string (palabra, definicion)

  calificacion es un par de string+float (alumno, nota)

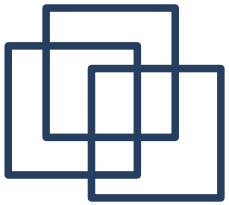
  .....

- Vector de elementos de distinto tipo

  Vector dinámico de enteros

  Vector dinámico de flotantes

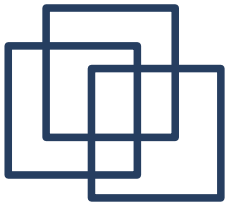
  Vector dinámico de alumnos



2. Generalización de tipos de datos (II).

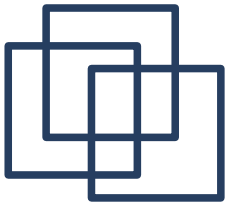
Parametrizar un tipo de dato consiste en introducir un parámetro en la definición del tipo para poder usarlo con distintos tipos.

En lugar de escribir cada especificación, representación e implementación independientemente se puede escribir una sola de cada, incluyendo uno o varios parámetros que representan *tipos de datos*. (Es el mismo mecanismo de abstracción que hizo surgir el concepto de procedimiento).



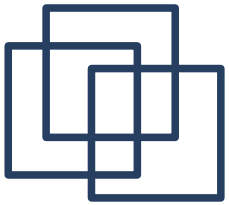
Ejemplo: TDA Pila

- La pila es un TDA que almacena elementos del mismo tipo satisfaciendo:
  - Estrategia LIFO (Last In First Out)
  - El contenido de la pila está oculto, sólo el elemento en el tope de la pila es visible.
- Ej. Uso Informática:
  - Diseño de sistemas operativos para manejo de interrupciones y llamadas a funciones del sistema
  - Son utilizadas para ejecutar el lenguaje Java.



Ejemplo: Operaciones Pila

- Push: que añade un elemento en el tope de la pila. Cuando un nuevo elemento llega, se convierte en el nuevo tope de la pila
- Pop: Elimina el elemento del tope de la pila.
- Top: Devuelve el elemento en el tope de la pila.



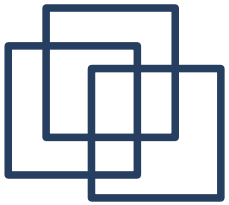
2. Generalización de tipos de datos (III).

El mecanismo que ofrece C++ para parametrizar tipo
son las clases patrón *template* de clases.

template <parámetros> declaración

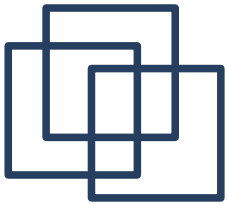
Los parámetros de la declaración genérica pueden ser:

- typename identificador. Se instancia por un tipo de dato.
- tipo-de-dato identificador. Se instancia por una constante.



Clase Pila

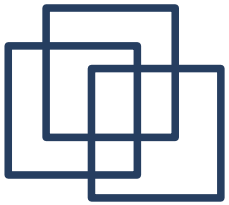
```
template<typename T>
class stack {
   public:
      stack();
      stack( const stack<T> & s);
      T top() const;
      void pop();
      void push( const T & x);
      stack<T> & operator=(const stack<T> & org);
      bool empty( ) const;
      ~stack();
```



Clase Pila

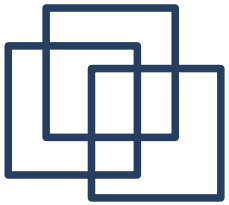
```
template<typename T>
class stack {
   public:
      ......
   private:
        T * datos;
        int max;
        int tama;

};
```



Clase Pila: Impl.

```
template<typename T> stack<T>::stack(){
                     datos = new T[10];
                     max = 10;
                     tama = 0;   }
template<typename T>
stack<T>::stack(const stack<T> & s){
                     max = s.max;
                     tama = s.tama;
                     datos = new T[max];
                     for (int i = 0; i< tama; i++)
                        datos[i] = s.datos[i];
                     }
```

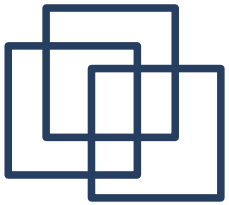


Clase Pila: Impl.

```
template<typename T> stack<T>::~stack(){
   delete [] datos;
}

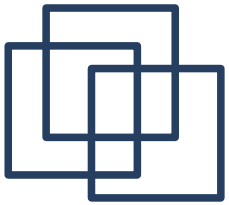
template<typename T> T stack<T>::top() const {
  return datos[tama-1];
}

template<typename T> bool stack<T>::empty() const
{ return tama == 0; }
```



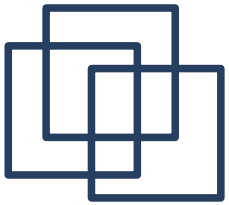
Clase Pila: Impl.

```
template<typename T> void stack<T>::push( const T & x)
{   if (tama >= max)
        { max = max * 2;
          T * aux = new T[max];
          for (int i = 0; i< tama; i++)
                aux[i] = datos[i];
          delete [] datos;
          datos = aux;
        }
        datos[tama]= x;
        tama++;
}
```



Clase Pila: Impl.

```
template<typename T>  stack<T> &
  stack<T>::operator=(const stack<T> & org)
{         max = org.max;
        tama = org.tama;
        if (this != &org)
           { delete [] datos;
             datos = new T[max];
             for (int i = 0; i< tama; i++)
                    datos[i] = org.datos[i];
           }
        return *this;
}
```



2. Generalización de tipos de datos (IV).

Para usar un tipo genérico hay que instanciarlo, indicando los tipos concretos con que se quiere particularizar.

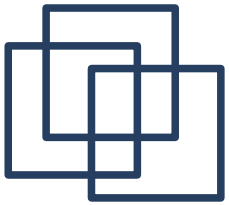
```
stack<int> x; // Pila de entero

stack<string> y; // Pila de string

stack<racional> z; // Pila de racionales

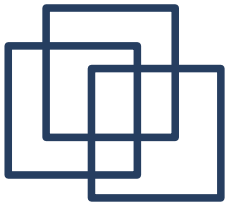
o incluso

stack< stack<int> > W; // Pila de Pilas de
  enteros
```



Uso TDA Pila

```
#include “stack.h”
stack<int> x,y;
stack<stack<alumno> > s;
for (int i = 1; i<5;i++)
        x.push(i);
    y = x;
    while (!y.empty() )
      { cout << y.top()  << " ";
        y.pop( );
        }
```



2. Generalización de tipos de datos (III).

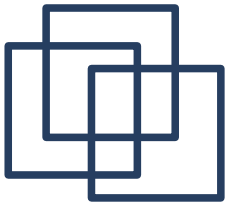
Recordemos:

Template: mecanismo que ofrece C++ para parametrizar tipos

template <parámetros> declaración

Los parámetros de la declaración genérica pueden ser:

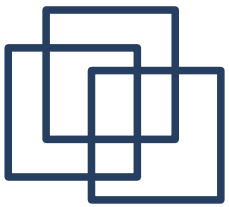
- typename identificador. Se instancia por un tipo de dato.
- tipo-de-dato identificador. Se instancia por una constante.



2. Generalización de tipos de datos (III).

- Definición de la clase:

```
template <typename T, int n>
array_n {
      public:
        array_n( );
        T elemento( int i );
          ......
        private:
          T items[n];
        };
```

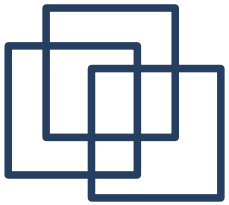


2. Generalización de tipos de datos (III).

- Definición de los métodos:

```
template<typename T, int n>
array_n<T,n>::array_n( ){
    for (int i = 0; i< n; i++)
          items[i] = -1;
  }

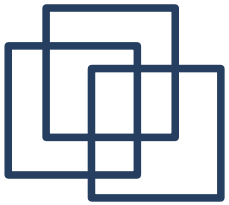
template<typename T, int n>
T array_n<T,n>::elemento(int i){
   return items[i];
}
```



2. Generalización de tipos de datos (IV).

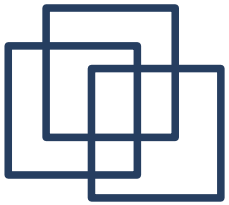
Para usar un tipo genérico hay que instanciarlo, indicando los tipos concretos con que se quiere particularizar.

```
array_n<int,10> x; // Array de 10 entero
array_n<float,5> y; // Array de 5 flotantes
array_n<racional,100> z; // Array de racionales
```



2. Generalización de tipos de datos (IV).

La especificación de un TDA genérico se hace igual que la de un TDA normal y se añaden aquellos requisitos que deban cumplir los tipos para los que se quiera instanciar. Habitualmente, la existencia de ciertas operaciones.



Ejemplo: Especificación de la clase par

/** Especificacion del par de elementos

par::par, primero, segundo, ==, <

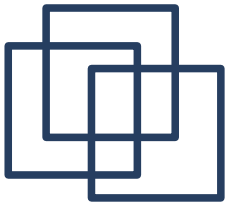
Este TDA representa el par de objetos de las clases T1,T2. Son mutables.

Requisitos para la instanciación:

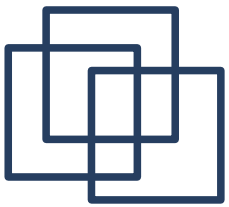
Las clases T1 y T2 debe tener las siguientes operaciones:

- Constructor por defecto y copia
- Operador menor
- Operador de asignación */

.....Veamos el fichero par.h

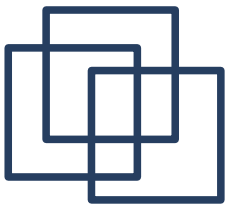


```
template <typename T1, typename T2> class par {
public:
   par();
   par(const T1 &a, const T2 &b);
   par(const par<T1,T2>& p);
   T1 primero( );
   T2 segundo( );
   bool operator<(const par<T1,T2>&y) const;
   bool operator==(const par<T1,T2>&y) const;
private:
      T1 uno;
      T2 dos;
};
```



Métodos de la clase par: par.template

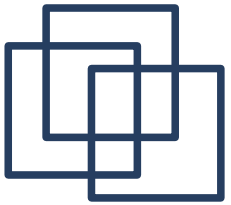
- ```
  template<typename T1, typename T2>
     par<T1,T2>::par()
     {  uno =  T1() ; dos = T2() ; }
  ```
- ```
  template<typename T1, typename T2>
     par<T1,T2>::par(const T1 &a, constT2 &b)
     { uno=a; dos=b; }
  ```
- ```
  template <typename T1, typename T2>
     par<T1,T2>::par(const par<T1,T2> &p)
     {   uno=p.uno;   dos=p.dos;    }
  ```
  ```



Métodos de la clase par: par.template

- ```
  template <typename T1, typename T2>
  bool par<T1,T2>::operator==( const par<T1,T2> &y)
  {  return ( (uno == y.uno) && (dos == y.dos;)) }
  ```

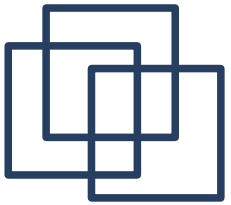
- ```
  template <typename T1, typename T2>
  bool par<T1,T2>::operator<( const par<T1,T2> &y)
  {  return (uno < y.uno) ||  ( !(y.uno < uno) &&
  dos <y.dos ); }
  ```



Uso de la Clase par

Para usar un tipo genérico hay que instanciarlo

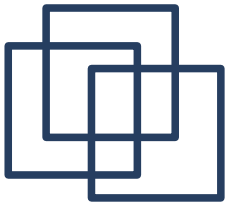
```
#include <par.h>
void funcion1() {
   par<int, int> x, y(2,3);      // instanciación
   int z = y.segundo();
   if (x<y) cout << “X es menor” <<endl;
    }
void algo(const par<int,float> & p){
   cout<< p.primero();
   .... }
```



Uso de la Clase par

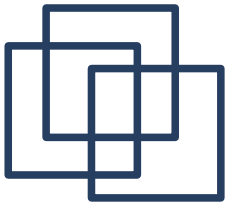
```
template <typename T1, typename T2>
void PintaPar(const par<T1,T2> & p)
  { cout<< p.primero() << p.segundo(); }
```

```
int main() {
        par<float,string> nota;  // instanciación de la clase
        par<string, alumno> definicion;//
        pinta_par(nota);
        pinta_par(definicion);
}
```



Tratamiento de los templates

- El compilador generará las definiciones de clases y sus funciones miembro correspondientes para cada instanciación que encuentre.
- La definición completa de la clase genérica debe estar disponible: tanto la definición de la clase como las de las funciones, para que el compilador pueda particularizarlas.
- Por ello, el código se organizará como siempre: interfaz en el fichero .h e implementación en el fichero .template. Pero la inclusión será el .template al final del .h.



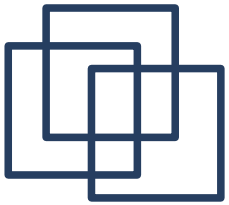
Organización del código

par.h

```
#ifndef __par_H__
#define __par_H__
template <typename T1,
   typename T2>
class par {
  ...
};
#include "par.template"
#endif
```

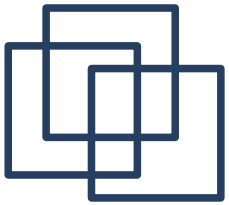
par.template

```
template <typename T1,
   typename T2>
par<T1,T2>::par()
{
 ...
}
...
```



Vector Dinámico: vdin.h

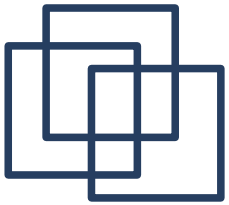
```
template<typename T> class vdin {
    public:
     vdin( int n=0 );
     vdin( const vdin<T> & org);
     ~vdin();
     int size() const;
     T & operator[](int n);
     const T & operator[ ](int n) const;
     void resize( int n);
     vdin<T> & operator=( const vdin<T> & org);
    private:   T * datos;   int tama;
};
```



Vector Dinámico: vdin.template

```
template<typename T>
vdin<T>::vdin(int n)
{ tama = n;
  if (n!=0) datos = new T[n];
  else datos = 0;
}

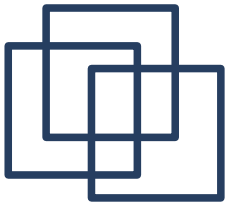
template<typename T>
vdin<T>::vdin( const vdin<T> & org)
{
  tama = org.tama;
  datos = new T[tama];
  for (int i = 0; i< tama; i++)
    datos[i] = org.datos[i];
}
```



Vector Dinámico: vdin.template

```
template<typename T>
vdin<T>::~vdin()
{  delete [] datos; }

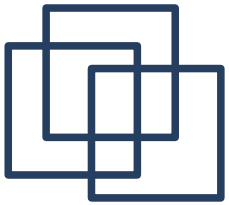
template<typename T>
int vdin<T>::size( ) const
{  return tama; }
```



Vector Dinámico: vdin.template

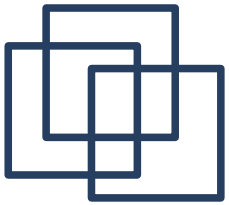
```
template<typename T>
T &  vdin<T>::operator[](int n) {
  return datos[n];
}

template<typename T>
const T &  vdin<T>::operator[](int n) const{
  return datos[n];
}
```



Vector Dinámico: Uso

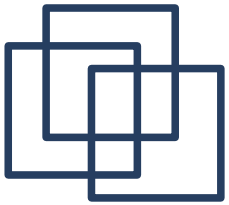
```
#include "vdin.h"
int main()
{
  vdin<int> X(10), Y;
  for (int i = 0; i<10; i++)
    X[i] = i+2;
  for (int i=0;i<10;i++)
    cout << X[i]<< " ";
  cout << endl;
  vdin<char> C(10);
```



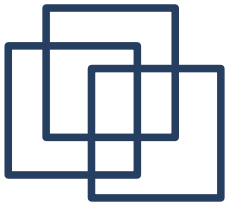
Functores Genéricos

```
template <typename T>
 class menor{    // Requisito T debe tener operator<
   public:
     bool operator()(const T & a, const T & b) const;
      };

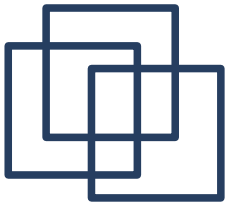

template<typename T>
bool menor<T>::operator()(const T & a, const T & b)
  const
{ return a < b; }
```



```
template <typename T, class comparar>
void ordenar(T a[],int n, comparar cmp){
int i,j,im;
T aux;
for(i=0;i<n-1;i++){
   im=i;
   for(j=i+1;j<n;j++)
      if( cmp(a[j],a[im]))    im=j;
   aux=a[i]; a[i]=a[im]; a[im]=aux;
  }
}
```



```
int main()
{
float V[10];
int X[10];
alumno Z[100];
menor<float> men;
....
ordenar(V,10,men);
ordenar(X,10,menor<int>());
ordenar(Z,100,menor<alumno>());
 }
```



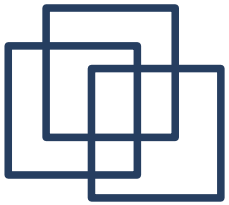
Uso genérico,

En c++ existen, entre otros,

- el functor less (equivale a menor)
- El functor greater (equivale a mayor)

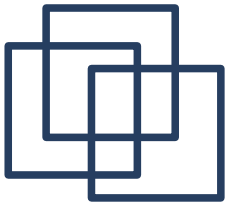
```
#include <funtional>


less<int> c_menor;
cout << minimo(Ai,100,c_menor) << endl;
ordenar(A,100,c_menor);
ordenar(Z,100, greater<alumno>() );
```



Tema 2. Abstracción

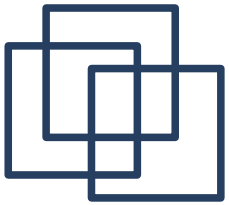
- Introducción
- Abstracción en programación
- Abstracciónes procedimentales
- Abstracción de datos: Tipo de Dato Abstracto(TDA)
- TDA en C++
- Abstracción por generalización
- Abstracciónes de iteración:
  Contenedores e iteradores



Contenedores e iteradores

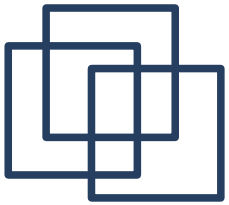
- **Contenedor:** Embalaje metálico grande y recuperable, de tipos y dimensiones normalizados internacionalmente y con dispositivos para facilitar su manejo. (Diccionario RAE)

- *Concepto de contenedor en programación*: Es una abstracción que permite almacenar (embalar) una colección de elementos de un mismo tipo, aportando un conjunto de operaciones que faciliten las labores de gestión de los mismos.
- El use de templates nos permite poder hablar de contenedores de forma genérica



Contenedores e iteradores (II)

- La especificación de un contenedor básico debería de aportar métodos para:
  - Insertar un elemento,
  - Borrar un elemento
  - Consultar un elemento
  - Moverse, dentro de un rango, sobre los elementos almacenados: Iterar. Por ejemplo, desde el primero al ultimo.



Contenedores e iteradores (II)

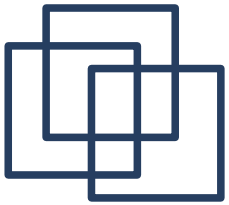
- Mediante el uso de contenedores nos abstraemos de la forma particular en la que se almacenan los datos y de la gestión necesaria para el almacenamiento.

..... Damos un paso más en la abstracción

Vector Dinámico puede verse como un contenedor

Necesitamos dotarlo de mecanismo de iteración que controle las **posiciones** del contenedor.

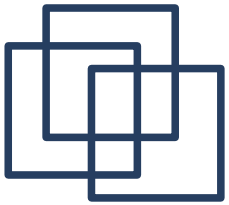
–Ventaja: podemos resolver problemas independientemente del contenedor concreto que utilicemos.



Contenedores e iteradores (III)

- Ejemplo: Eliminar palabras vacías en R.I.

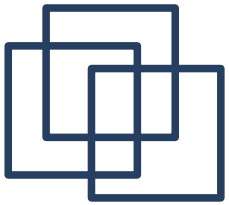
```
contenedor<termino> d_org, d_dest, stopwords;
termino dato;
posiciones_del_contenedor pos;
 .....
pos = primera posicion en d_org
while (pos != ultima posicion en d_org)
  { dato = elemento en pos;
    if (!stopwords.buscar(dato))
            d_dest.insertar(dato)
     }
```



Ejemplo II

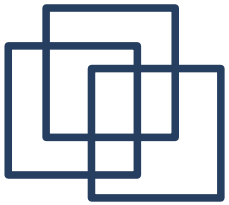
- Algoritmo que cuente el número de veces que se repite el mayor elemento del contenedor:

```
template<typename T> int repite_mayor(Contenedor<T> C)
  {  DECLARACION DE VARIABLES
        T mayor,
        int repite = 0;
        Posiciones_del_contenedor pos;
     BUSQUEDA DEL MAYOR ELEMENTO EN EL CONTENEDOR
     CONTAR CUANTAS VECES SE REPITE
    return repite;
  }
```



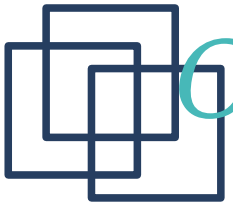
```
template<typename T> int repite_mayor ( Contenedor<T> C)
   { T mayor, int repite = 0; Posiciones_del_contenedor pos;
      pos = primera posicion del contenedor C;
       mayor = elemento almacenado en pos.
       while (pos no sea la ultima posicion del contenedor C)
         {  if (elemento almacenado en la posicion actual > mayor)
               mayor = elemento almancenado en posicion actual;
          avanza pos a la siguiente posicion;
          }
       desde la primera posición a la última del contenedor C
           if (el elemento almacenado en posición actual == mayor )
                 repite++;
       return repite;
   }
```

.....Analicemos el código con detalle



Contenedores e iteradores (IV)

```
{ T mayor;  int repite = 0;
  Posiciones_del_contenedor pos;
  pos = primera posicion del contenedor C;
  while (pos != ultima posicion del contened C)
    {
    if (elemento en posicion actual > mayor)
          mayor = elemento en posicion actual;
    avanza pos a la siguiente posicion;
    }
```



*Contenedores e iteradores (*IV*)*

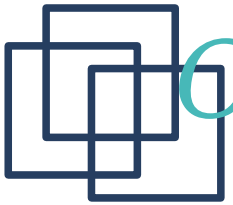
Un iterador permite trabajar con las posiciones en las que se almacenan los elementos del contenedor. Será un tipo de dato ligado al contenedor

```
contenedor<C>::iterator
```

El **contenedor debe dotarse** de los siguiente métodos:

- begin() iterador que referencia a la primera posición del contenedor.
- end() iterador que referencia a la posición fina del contenedor.

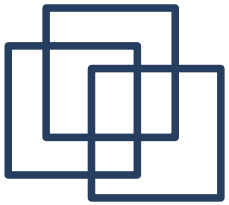
Por definición end() es la posicion siguiente al último elemento



Contenedores e iteradores (IV)

template<typename T> int repite_mayor
(Contenedor<T> C)

```
{ T mayor;  int repite = 0;
  typename contenedor<T>::iterator pos;
  pos = C.begin();
  while (pos != C.end() )
    {
    if (elemento en posicion actual > mayor)
          mayor = elemento en posicion actual;
    avanza pos a la siguiente posicion;
    }
```



Iteradores

El TDA iterator debe tener mecanismos para:

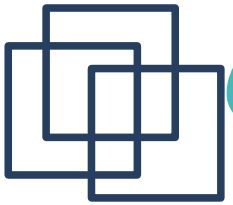
1)Construir elementos del tipo (Constructores)

2) Asignar la misma posición (operador de asignación): operator=

3) Acceder al elemento almacenado en la posición referenciada: operator*

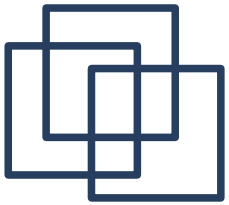
4) Comparar si dos iteradores referencian a la misma posición: operator==, operator!=

5) Poder desplazarse por las posiciones del contenedor:: operator++, operator--



Contenedores e iteradores (VI)

```
T mayor;  int repite = 0;
typename contenedor<T>::iterator pos;    (1)
        pos = C.begin();                 (2)
     mayor = *pos.     (3)
      while (pos != C.end() )                (4)
        {  if (*pos > mayor)     mayor = *pos;  (3)
           pos++;  }          (5)
```



TDA iterator

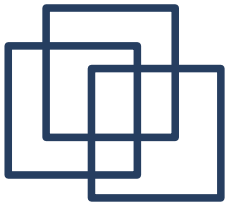
- El iterador es un TDA
  - es una nueva clase de nombre iterator
- Estará ligado estrechamente al contenedor
  - Se implementa dentro del contenedor sobre el que itera.

bool operator!= (const contenedor<T>::iterator &x) const;

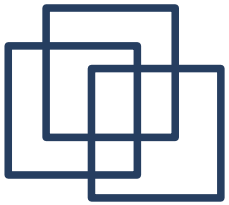
Operador de desigualdad.

T & operator*() ;

Devuelve el elemento referenciado por el iterador.



```
template <typename T> class contenedor {
  public:
      contenedor(); .....
      class iterator {
         public:
         iterator();
         bool operator==(const contenedor<T>::iterator &x) const;
         iterator operator++();
         T & operator*  ( )  .....
        private:
           ......};
      private:
      ......
};
```

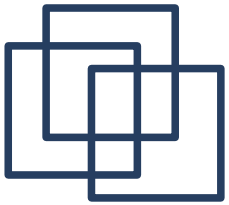


```
contenedor<T>::iterator::iterator();
```

Constructor por defecto, nos crea un iterador sobre el contenedor.

```
bool contenedor<T>::iterator::operator==
        (const contenedor<T>::iterator &x) const;
```

Operador de igualdad, devuelve verdadero cuando los dos iteradores hacen referencia al mismo elemento del contenedor, falso en caso contrario.



```
bool contenedor<T>::iterator::operator!= (const
  contenedor<T>::iterator &x) const;
```

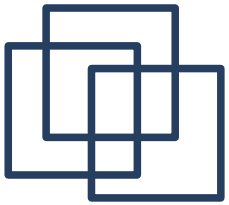
Operador de desigualdad.

```
T &  contenedor<T>::iterator::operator*() const;
```

Devuelve el elemento referenciado por el iterador.

```
contenedor<T>::iterator &
  contenedor<T>::iterator::operator=
              (const contenedor<T>::iterator &x);
```

Operador de asignación, permite que iterador referencie al mismo elemento que el iterador x.



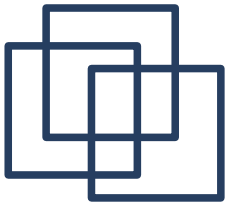
TDA iterator (II)

```
contenedor<T>::iterator
  contenedor<T>::iterator::operator++();
```

Operador de incremento, hace que el iterador referencie al siguiente elemento del contenedor. Su comportamiento no esta definido si el iterador referencia al último elemento del contenedor.

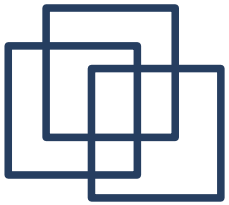
```
contenedor<T>::iterator
  contenedor<T>::iterator::operator--();
```

Operador de decremento, hace que el iterador referencia al elemento anterior. Su comportamiento es indefinido si el iterador referencia al primer elemento.



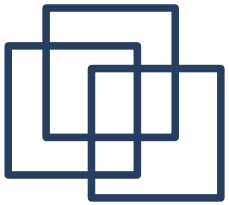
Ejemplo completo.....

```
template<typename T> int repite_mayor ( Contenedor<T> C)
{ T mayor, int repite = 0;
  typename contenedor<T>::iterator pos;
  pos = C.begin();
  mayor = *pos;
    while ( pos != C.end() )
     {  if ( *pos > mayor) mayor = *pos;
      pos ++; }
    for ( pos= C.begin() ; pos!=C.end() ; pos++)
       if (*pos == mayor )  repite++;
   return repite;
}
```



Ejemplo con functores.....

```
template<typename T, class comparar>
int repite( Contenedor<T> C, comparar cmp )
{ T mayor, int repite = 0;
 typename  contenedor<T>::iterator pos;
  pos = C.begin();
  mayor = *pos;
    while ( pos != C.end() )
     {  if ( cmp(*pos, mayor) )  mayor = *pos;
      pos ++; }
    for ( pos= C.begin() ; pos!=C.end() ; pos++)
       if (*pos == mayor )  repite++;
   return repite;
}
```



Ejemplo con functores.....

```
#include <iostream>
#include<functional>
int main(){
vector<int> X;
vector<alumno> Y;
list<float> L;
cout << repite(X, less<int>() );
cout << repite(X, greater<int>() );
cout << repite(L, less<float>() );
....
}
```