

**A**lgoritmos y

**E**structuras

**D**e

**D**atos

# **TIPOS DE DATOS**

## **ABSTRACTOS**

### **(TDA)**

# ESTRUCTURAS DE DATOS: Clasificaciones

- Según donde se almacenan
  - Internas (en memoria principal)
  - Externas (en memoria auxiliar)
- Según tipos de datos de sus componentes
  - Homogéneas (todas del mismo tipo)
  - No homogéneas (pueden ser de distinto tipo)
- Según la implementación
  - Provistas por los lenguajes (básicas)
  - Abstractas (TDA - Tipo de dato abstracto que puede implementarse de diferentes formas)
- Según la forma de almacenamiento
  - Estáticas (ocupan posiciones fijas y su tamaño nunca varía durante todo el módulo)
  - Dinámicas (su tamaño varía durante el módulo y sus posiciones también)

# ESTRUCTURAS DE DATOS: Clasificaciones

- Según la implementación:

- **Provistas por los lenguajes:** Básicas



Array  
Struct  
String  
File

- **Abstractas:** TDA - Tipo de dato abstracto que puede implementarse de diferentes formas.



Un **TDA** es un tipo de dato **definido por el programador** que se puede manipular de un modo **similar a los tipos de datos provistos por el lenguaje**.

Un TDA está formado por **un conjunto de valores** válidos de datos y **un conjunto de operaciones primitivas** que se pueden realizar sobre esos valores.

Los usuarios pueden **crear** variables con valores del conjunto válido y **operar** sobre esos valores.

# Definición de TDA

Los **TDA** proporcionan un mecanismo adicional mediante el cual se realiza una **separación** clara **entre la interfaz y la implementación del tipo de dato**.

La **implementación** de un TDA consta de:

- La **representación**: elección de las estructuras de datos
- Las **operaciones**: elección de los algoritmos

La **interfaz** del TDA se asocia con las operaciones y datos del TDA, y es visible al exterior.

# Revisemos algunos tipos de datos básicos

- En C++, el **tipo int** (enteros) se corresponde con el **conjunto de valores** del siguiente rango:  $-2.147.483.648$  a  $+2.147.483.647$ ; y las **operaciones primitivas** posibles son:

Operadores de asignación:  $=$ ,  $*=$ ,  $/=$ ,  $\%=$ ,  $+=$ ,  $-=$

Operadores aritméticos:  $+(\text{signo})$ ,  $-(\text{signo})$ ,  $*$ ,  $/$ ,  $\%$ ,  $+$ ,  $-$

Operadores de incremento y decremento:  $++$ ,  $--$

Operadores relacionales:  $==$ ,  $!=$ ,  $>$ ,  $<$ ,  $>=$ ,  $<=$

- Similarmente podemos emplear el **tipo float** (tipo de coma flotante) para manipular números decimales (reales).
  - Con su conjunto de valores y operaciones permitidas

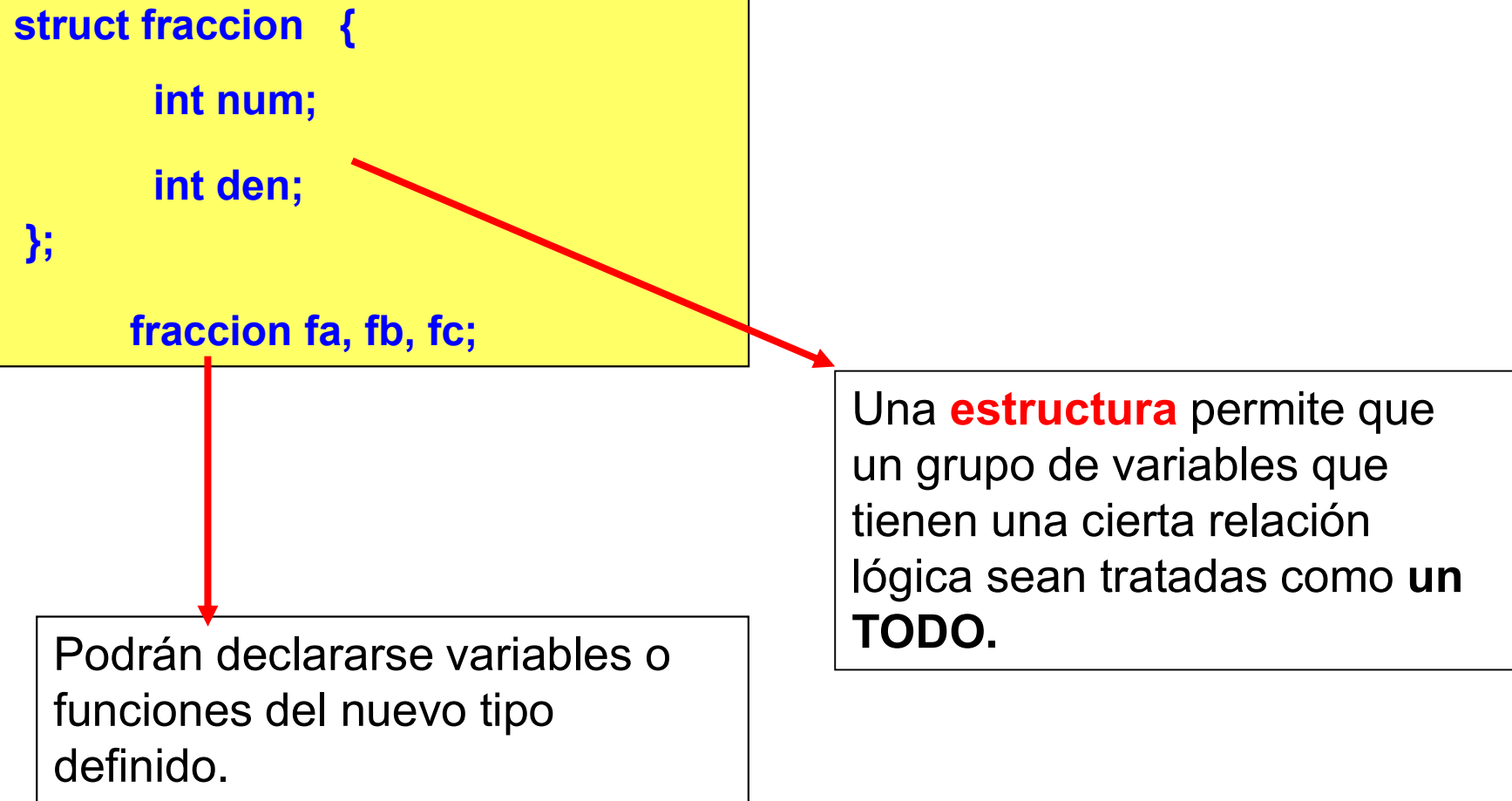
## Tipo de dato: fracción

- En C++ **no está predefinido** el tipo de datos **fracción**.
- Si necesitamos almacenar y operar con fracciones, deberemos **definir un tipo de dato:**
  - nuevo
  - diseñado y definido por el programador
  - que no preexiste (será abstracto),
  - creado con los datos y operaciones primitivas que provee el sistema.

## Tipo de dato: fracción

- Una forma de definirlo es pensar en **representar** a una fracción como una estructura (struct) compuesta de dos campos de tipo int: numerador y denominador.

```
struct fraccion {  
    int num;  
    int den;  
};  
  
fraccion fa, fb, fc;
```



Una **estructura** permite que un grupo de variables que tienen una cierta relación lógica sean tratadas como **un TODO**.

Podrán declararse variables o funciones del nuevo tipo definido.



## Tipo de dato: fracción

- Con esta definición **fracción** se corresponde con el conjunto de valores formado por todos los pares ordenados de números enteros:
  - Rango:  $-2.147.483.648$  a  $+2.147.483.647$
- También debe definirse un conjunto de operaciones primitivas (**interfaz**) que se pueden realizar sobre datos del tipo fracción.
- Para describir un TDA, y en particular la **interfaz** (operaciones permitidas), se puede usar una notación informal o una especificación formal. Optamos por la primera, indicando:

*Operación (tipos de argumentos) → resultado, descripción*

# Tipo de dato fracción: Operaciones primitivas

- Para el TDA fraccion, el **conjunto de operaciones primitivas (interfaz)** puede ser:

*fracIguales* (fraccion, fraccion) → boolean; determina si dos fracciones son iguales, esto es si tienen el mismo numerador y denominador.

*fracEquiv* (fraccion, fraccion) → boolean; determina si dos fracciones son equivalentes, es decir si tienen el mismo valor.

*asigFrac* (entero, entero) → fraccion; devuelve una fracción que tiene como numerador al 1er. argumento y como denominador al 2do.

*sumFrac* (fraccion, fraccion) → fraccion; realiza la operación suma de dos fracciones.

*multFrac* (fraccion, fraccion) → fraccion; realiza la operación multiplicación de dos fracciones.

*restFrac* (fraccion, fraccion) → fraccion; realiza la operación resta de dos fracciones.

*divFrac* (fraccion, fraccion) → fraccion; realiza la operación división de dos fracciones.

*printFrac* (fraccion) → void; realiza la impresión de una fracción con el formato literal "numerador / denominador".

## Tipo de dato: fracción

- Luego de definido el tipo de dato abstracto (**valores y operaciones**) y teniendo declaradas las variables fa, fb y fc, un módulo (código de usuario) que emplee este nuevo tipo de datos sería, por ejemplo:

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

int main (void) {
    int n, d;
    fraccion fa, fb,fc;
    cout << "Ingrese una fraccion como num seguido de den: ";
    cin >> n >> d;
    fa = asigFrac (n,d);
    fb = asigFrac (3,5);
    fc = multFrac (fa,fb);
    cout << endl << "El resultado de la multiplicacion por 3/5 es: ";
    printFrac (fc);
    system("pause");

    return 0;
}
```

Falta declarar previamente las funciones que corresponden a las operaciones primitivas!!!

# Tipo de dato: fracción

- Así por ejemplo la **declaración (implementación)** de las funciones *asigFrac*, *fracEquiv*, *multFrac* y *sumFrac*, podría ser:

/\* Devuelve una fracción \*/

```
fraccion asigFrac (int x,int y)
{
    fraccion aux;
    aux.num = x;
    aux.den = y;
    return aux;
}
```

/\* Multiplica dos fracciones \*/

```
fraccion multFrac (fraccion a, fraccion b)
{
    fraccion aux;
    aux.num = a.num * b.num;
    aux.den = a.den * b.den;
    return aux;
}
```

/\* Suma dos fracciones \*/

```
fraccion sumFrac (fraccion a,fraccion b)
{ fraccion aux;
  aux.den = a.den * b.den;
  aux.num = b.den * a.num +
            a.den * b.num;
  return aux;
}
```

/\* Determina si dos fracciones son equivalentes \*/

```
int fracEquiv (fraccion a, fraccion b)
{
    return (a.num *b.den == a.den *b.num);
}
```

## Tipo de dato: fracción

- La función definida para sumar devuelve una fracción que no está reducida (simplificada). Si se tiene definida una función que calcule el mínimo común múltiplo entre 2 enteros, la función *sumFrac* podría definirse como:

```
fraccion sumFrac (fraccion a, fraccion b)

{ fraccion aux;
  aux.den = mcm (a.den, b.den);
  aux.num = aux.den / a.num +
            aux.den / b.num;
  return aux;
}
```

- Si necesitan sumarse 3 fracciones puede emplearse la composición de funciones. Ej:  $fd = \text{sumFrac}(fa, \text{sumFrac}(fb, fc))$

## Tipo de dato: fracción

Tener en cuenta que:

El usuario define:

- cómo va a representar el nuevo tipo de datos que está creando (TDA),
- cuáles van a ser las operaciones primitivas (interfaz) y
- el código (implementación) de dichas operaciones.

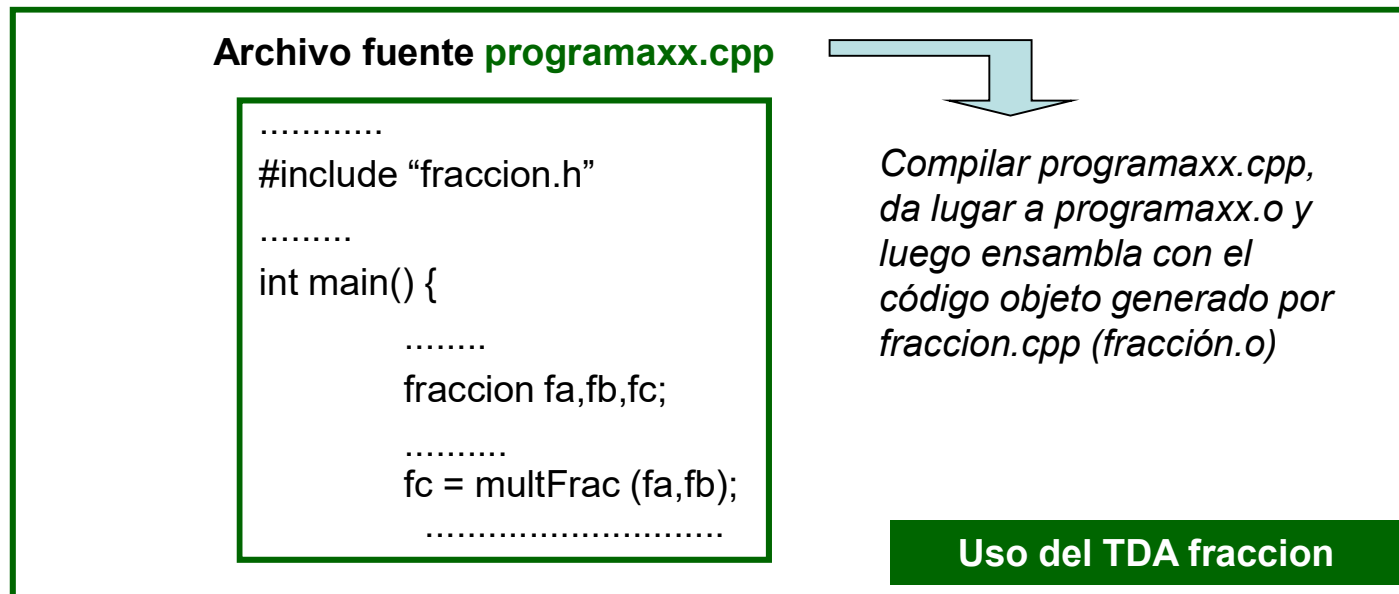
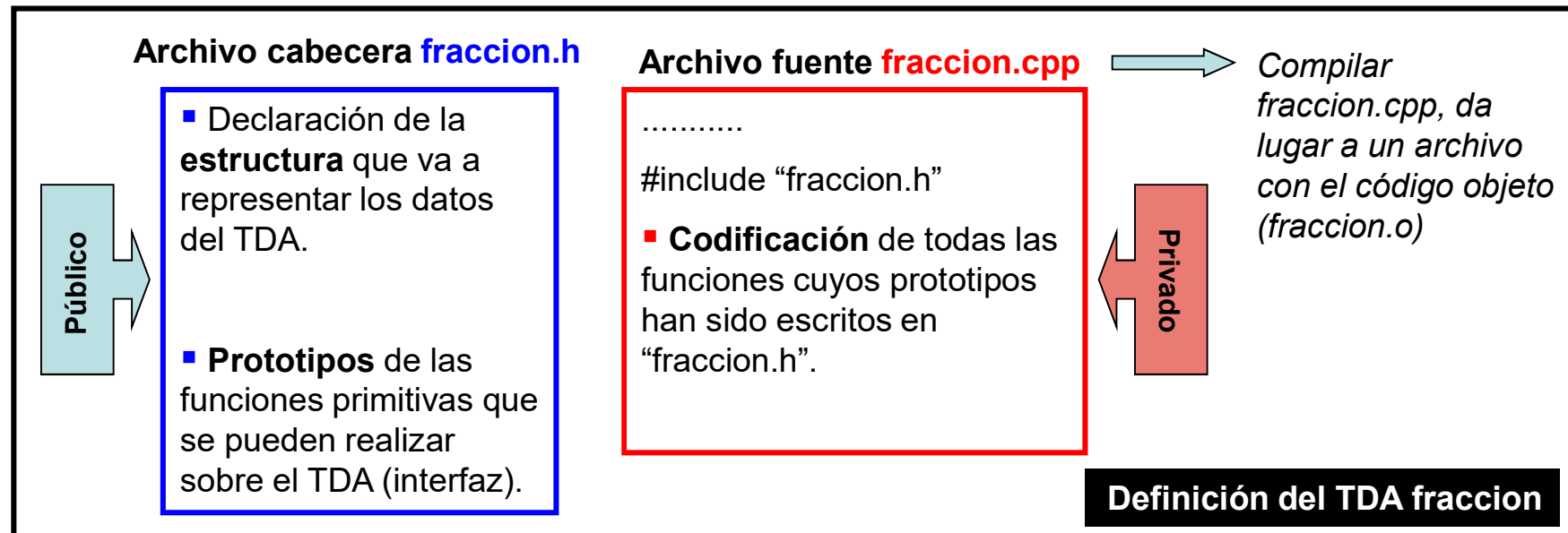
El programador que utilice el TDA debe conocer la representación y la interfaz.

La implementación queda oculta para el programador “cliente”.

# Implementación de TDA en C++

- Una de las características de C++ que permite implementar TDA son los archivos de inclusión o cabecera que se utilizan para agrupar en ellos variables externas, declaraciones de datos comunes y prototipos de funciones.
- Estos archivos de cabecera se incluyen (mediante la directiva al preprocesador *#include*) en los archivos que contienen la codificación de las funciones, archivos fuente y también en los archivos de código que hagan referencia a algún elemento del archivo de inclusión.
- Al implementar un TDA en C++ se agrupa en este archivo la representación de los datos y la interfaz del TDA (representada por los prototipos de las funciones).
- Luego, en los archivos de programas que vayan a utilizar el TDA se debe escribir la directiva: ***#include "tipoTDA.h"***

# Definición y Uso de TDA en C++





## Algunas consideraciones de Uso de TDA en C++

- Los programas fuente que vayan a utilizar un TDA necesitan conocer los **datos públicos** del mismo, es decir su **representación** y las **funciones primitivas** que se tienen disponibles, por ello se debe incluir el archivo de cabecera (.h).
- La implementación de las funciones primitivas **es transparente** para el programa que usa el TDA, si bien debe existir y ensamblarse, el programador no necesita conocer ese código.
- Más aún, el archivo fuente (.cpp) que tiene el código de las funciones, puede cambiarse y solo será necesario recompilar, sin que esto afecte al programa que usa el TDA, siempre que la representación e interfaz se mantengan.
- Los compiladores de C++ disponen de herramientas para el manejo cómodo de aplicaciones modulares.

## Otro ejemplo: Un número complejo

- Otro tipo de dato numérico que puede ser necesario manejar son los **números complejos**, que no son tipos básicos (primitivos). Se puede generar un TDA en forma similar a las fracciones:

### Archivo cabecera **complejo.h**

```
struct complejo{
    float preal;
    float pimag;
};

complejo sumComp (complejo a,complejo b);
complejo asigComp (float x,float y);
int complgiales (complejo a, complejo b);
.....
```

Números complejos: describen la suma de un número real y un número imaginario

### Archivo fuente **complejo.cpp**

```
.....
#include "complejo.h"

complejo sumComp (complejo a,complejo b)
{ complejo aux;
  aux.preal = a.preal + b.preal;
  aux.pimag = a.pimag + b.pimag;
  return aux;
}

complejo asigComp (float x,float y)
.....

int complgiales (complejo a, complejo b)
{
  return( (a.preal == b.preal) && (a.pimag ==b.pimag));
}
.....
```

## Otro ejemplo: Un número complejo

- Luego, un programa que maneje números complejos puede ser:

Archivo fuente **programayy.cpp**

```
.....  
#include "complejo.h"  
.....  
{ float r, i;  
  complejo ca, cb, cc;  
  cin >> r >> I ;  
  ca = asigComp (r,i);  
  cb = asigComp (3.0,5.5);  
  cc = sumComp (ca,cb);  
  cout <<"El resultado de la suma es : " << endl ;  
  printComp (cc);  
  system("pause");  
}
```

Esta función imprime: **cc.preal**, **"+"**, **cc.pimag**, **"i"**

## Otro ejemplo: Un punto del plano

- Si debemos trabajar con **puntos del plano**, podemos también construir un TDA:

### Archivo cabecera **puntop.h**

```
struct puntop
{ float abs;
  float ord;
};

puntop sumVect (puntop a,puntop b);
puntop asigPtop (float x,float y);
int ptopIguales (puntop a, puntop b);
int distancia (puntop a, puntop b);
.....
```

### Archivo fuente **puntop.cpp**

```
.....
#include "puntop.h"

puntop sumVect (puntop a, puntop b)
{ puntop aux;
  aux.abs = a.abs + b.abs;
  aux.ord = a.ord + b.ord;
  return aux;
}

int ptopIguales (puntop a, puntop b)
{
  return( (a.abs == b.abs) && (a.ord ==b.ord));
}

float distancia (puntop a, puntop b)
{
  return( sqrt (pow((a.abs - b.abs),2) +
    pow ((a.ord-b.ord),2) );
} .....
```

## Otro ejemplo: Una recta del plano

- Si debemos trabajar con **rectas del plano**, podemos considerar que una recta está caracterizada por tres coeficientes de su ecuación implícita ( $ax+by+c=0$ ) y así construir un TDA:

### Archivo cabecera **recta.h**

```
struct recta
{ float a; float b; float c;
  };

recta asigRecta (float x,float y,float z);
int esVertical (recta r);
int esHorizon (recta r);
int sonParalelas (recta r, recta q);
int sonPerpend (recta r, recta q);
int sonOblicuas (recta r, recta q);
float pendiente (recta r);
float ordorigen (recta r);
float intersEjex (recta r);
float absIntersec (recta r, recta q);
float ordIntersec (recta r, recta q);
.....
```

### Archivo fuente **recta.cpp**

```
.....
#include "recta.h"

int esVertical (recta r)
{ return ( r.b==0); }

int sonParalelas (recta r, recta q)
{return( r.a*q.b == q.a * r.b) ; }

float ordorigen (recta r, recta q)
{return (- r.c / r.b);}

float intersEjex (recta r)
{return (- r.c / r.a);}

float absIntersec (recta r, recta q)
{return( ((q.c/q.b)-(r.c/r.b)) / ((r.a/r.b)-(q.a/q.b)) );}
```

## Otro ejemplo: Una recta del plano

- Un programa que use rectas:

Archivo fuente **programazz.cpp**

```
.....
#include "recta.h"

{ float m,n,o;
  recta r1,r2;

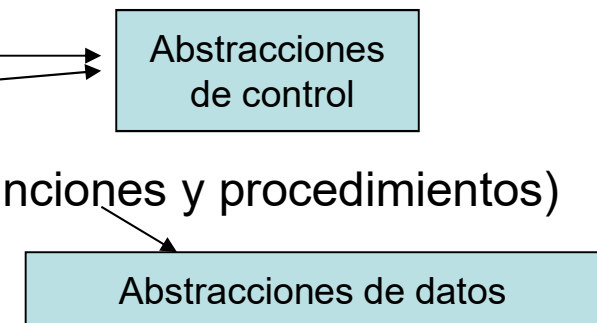
  cin >>m >>n >>o ;
  r1= asigRecta (m,n,o);
  if (! esVertical (r1))
    {cout <<"pendiente: " << pendiente(r1) ;};

  cin >> m >>n >> o;
  r2= asigRecta (m,n,o);
  if (! sonParalelas(r1,r2))
    {cout <<"el punto de interseccion de las rectas es : ( " << absIntersec (r1, r2) << " , "
      << ordIntersec (r1,r2) << ")" ;};

  system("pause");
}
```

# El rol de la abstracción

- La **abstracción** ha sido clave en la Programación, como mecanismo para poder controlar problemas de mayor complejidad (“los humanos hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad: abstraernos de ella”)
- Ante la dificultad de dominar en su totalidad los objetos complejos, **se ignoran los detalles no esenciales**, tratando en su lugar con el **modelo ideal** del objeto y centrándonos en el estudio de sus aspectos esenciales
- La abstracción es la capacidad para **encapsular y aislar información** del diseño y ejecución (ocultamiento de información)
- En la **historia del software** la abstracción ha sido clave:
  - Nombres mnemotécnicos para las instrucciones, en lugar de representación binaria
  - Macroinstrucciones
  - Estructuras de control
  - Procedimientos y funciones
  - TDA = Representación (datos) + Operaciones (funciones y procedimientos)



## Ventajas de los TDA

- Permite una **mejor conceptualización y modelización** del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.
- **Mejora la robustez** del sistema. Los TDA permiten la comprobación de tipos para evitar errores en tiempo de ejecución.
- **Mejora el rendimiento** (optimizar tiempos de compilación).
- **Separa la implementación de la especificación**. Permite la modificación y mejora de la implementación sin afectar la interfaz pública del TDA.
- **Permite la extensibilidad del sistema**. Los componentes de software reusables son más fáciles de crear y mantener.



## Ejemplo: TDA Conjunto

“conjuntoarr.h”, manejo de conjuntos con arreglos de tamaño acotado

```
#define M 10

struct Conjunto{
    int arr [M];
    int tam;
};

void conjuntoVacio(Conjunto &c);
int esVacio(const Conjunto c);
void annadir(Conjunto &c, int);
void retirar(Conjunto &c, int);
int pertenece(const Conjunto c, int);
int cardinal(const Conjunto c);
Conjunto unionC(const Conjunto c1, const Conjunto c2);
void mostrar(const Conjunto c);
```

## Ejemplo: TDA Conjunto

```
#include<iostream>
#include <stdlib.h>
using namespace std;
#include "conjuntoarr.h"

void conjuntoVacio(Conjunto &c){
    c.tam = 0;
}

int esVacio(const Conjunto c){
    return (c.tam == 0);
}

void annadir(Conjunto &c, int elemento){
    if (!pertenece(c, elemento)){
        if (c.tam < 10 ){
            c.arr[c.tam] = elemento;
            c.tam ++;
        }
    }
    return ;
}
```

**“conjuntoarr.cpp”**,  
manejo de  
conjuntos con  
arreglos de  
tamaño acotado

Una parte.....

## Ejemplo – TDA Conjunto

```
#include "conjuntoarr.h"

int main(){
    int j,r,k;
    Conjunto c,d,t;
    conjuntoVacio(c);
    cout << "Ingresar tamaño conjunto C: ";
    cin >> k;
    cout << "Ingrese los " << k << " datos:" << endl;
    for (j = 0; j < k; j++){
        cin >> r;
        annadir(c,r);
    }
    cout << endl << "El conjunto C: ";
    mostrar(c);
    cout << endl << endl << "Elemento a eliminar: ";
    cin >> r;
    retirar(c,r);
    cout<<endl<<endl<< "El conj. C luego de eliminar: ";
    mostrar(c);
    conjuntoVacio(d);
    cout<<endl<<endl<< "Ingresar tamaño conjunto D: ";
    cin >> k;
    cout << "Ingrese los " << k << " datos" << endl;
    for (j = 0; j < k; j++){
        cin >> r;
        annadir(d,r);
    }
    cout << endl << "El conjunto D: ";
    mostrar(d);
    t = unionC(c,d);
    cout << endl << "El conjunto CUD: ";
    mostrar(t);
    return 0;
}
```

“**main.cpp**”,  
emplea conjuntos  
manejados con  
arreglos de  
tamaño acotado

Una parte.....

## **LEER**

**“Abstracción en lenguajes de programación”, pág. 21**

**“Tipos abstractos de datos”, pág. 23**

**Libro: “Estructura de datos en C++” – Luis Joyanes Aguilar**