Programación Avanzada y Estructuras de Datos

2. Tipos abstractos de datos en C++

Víctor M. Sánchez Cartagena

Grado en Ingeniería en Inteligencia Artificial
Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

25 de septiembre de 2024

Índice

Características

- Programación imperativa:
  - Elemento principal: datos (variables)
  - Funciones facilitan operar con los datos
  - Datos y funciones están separados
- Programación orientada a objetos (POO):
  - Elemento principal: objetos
  - Objeto $=$ datos (*atributos*) $+$ funciones (*métodos*) que operan sobre ellos
  - Datos y funciones están unidos

El concepto de objeto

Un objeto es una entidad con un estado (atributos o variables de instancia) y unas funciones (métodos) que pueden acceder y modificar este estado

- Para evaluar las funciones hay que enviar un mensaje al objeto
- Sólo es posible consultar el estado de un objeto mediante alguno de sus métodos
- Barrera de abstracción
- Ocultación de información

Modelado de dominio

La POO permite modelar un dominio (problema a programar, un simulador de deportes, por ejemplo) de una forma muy cercana a la realidad

- Los objetos del programa simulan los objetos (sustantivos) del dominio (por ejemplo, bicicleta, marchas, carretera, etc.)
- Los métodos de los objetos permiten modelar perfectamente las acciones (verbos, por ejemplo cambiar de marcha, pedalear, etc.) que pueden realizar

Características de los lenguajes orientados a objetos

- Objetos (dinámicos) y clases (estáticas).
- Ocultación de información: objetos agrupan datos y funciones
- Los métodos se invocan mediante mensajes
- Herencia: clases se pueden definir usando otras clases como base
- Enlace dinámico: los objetos determinan qué código ejecutar

Índice

Primera clase en C++

Coordenada

- Datos: `x`, `y`, `z`
- Métodos:
  - `obtenerX`
  - `obtenerY`
  - `obtenerZ`
  - `ponerX`
  - `ponerY`
  - `ponerZ`

Primera clase en C++

- Instanciación de la clase `Coordenada`:

```
//Declaración de un objeto de la clase
Coordenada c;

//Declaración de un array estático
Coordenada c[10];

//Declaración de un puntero
Coordenada *ptC = &c;
```

Primera clase en C++

- Declaración de una clase:

```
class NombreClase {

//Atributos y métodos

};
```

- Clase `Coordenada`:

```
class Coordenada {
   int x,y,x;
};
```

Primera clase en C++

- Poniéndolo todo junto:

```
class Coordenada {
    int x,y,z;
};

int main(){
   int i;
   Coordenada c;
   return 0;
}
```

Compilación

```
g++ -std=c++11 -Wall -g prog.cc -o prog
```

Acceso a los miembros de una clase

- Mediante . para variables estáticas

```
Coordenada c;
cout << "Componente x"  << c.x << endl;
cout << "Componente y"  << c.y << endl;
cout << "Componente z"  << c.z << endl;
```

- Mediante -> para punteros

```
Coordenada c;
Coordenada *ptC = &c;
cout << "Componente x"  << ptC->x << endl;
cout << "Componente y"  << ptC->y << endl;
cout << "Componente z"  << ptC->z << endl;
```

Acceso a los miembros de una clase

- Pero el siguiente código da error de compilación: `error: ‘int Coordenada::x’ is private within this context`

```
class Coordenada {
    int x,y,z;
};

int main(){
   Coordenada c;
   c.x=1;
   c.y=2;
   c.z=1;
   return 0;
}
```

Control de acceso

- `public`: Accesible tanto desde la propia clase como desde otros métodos y funciones
- `private`: Accesible exclusivamente desde métodos de la propia clase y funciones y clases amigas
- `protected`: Los accesos permitidos con `private`, más acceso desde clases derivadas (herencia)

Control de acceso

- Implementación:

```
class MiClase{
public:
    //Parte pública
    //Normalmente, sólo métodos y no atributos

protected:
    //Parte protegida

private:
    //Parte privada
    //Normalmente, atributos y métodos auxiliares (no visibles desde e

};
```

Control de acceso

- Versión *arreglada* del código anterior:

```
#include<iostream>
using namespace std;
class Coordenada {
    public:
    int x,y,z;
};
int main(){
   Coordenada c;
   c.x=1;
   c.y=2;
   c.z=1;
   cout << "(" << c.x << ", " << c.y << ", " << c.z << ")" << endl;
   return 0;
}
```

- Atención: rompe ocultación de información

Separación de interfaz e implementación

- La declaración de clase se incluye un fichero `NombreClase.h`
- La implementación de los métodos en un fichero `NombreClase.cc`

Ventajas:

- Permite distribuir fichero `.h` y fichero `.cc` compilado $\rightarrow$ `.o`
- Se puede cambiar la implementación de los métodos sin afectar al programa que utiliza la clase

Desventajas:

- Cambios en atributos privados obligan a cambiar fichero `.h`

Separación de interfaz e implementación

- `Coordenada.h`:

```
  class Coordenada {
    private:
    int x,y,z;
    public:
    void setX(int);
    void setY(int);
    void setZ(int);
    int getX();
    int getY();
    int getZ();
};
```

Separación de interfaz e implementación

- Coordenada.cc:

```
#include "Coordenada.h"
void Coordenada::setX( int ix ){
    x=ix;
}
void Coordenada::setY( int iy ){
    y=iy;
}
void Coordenada::setZ( int iz ){
    z=iz;
}
int Coordenada::getX(){
    return x;
}
int Coordenada::getY(){
    return y;
}
int Coordenada::getZ(){
    return z;
}
```

Separación de interfaz e implementación

- Programa principal. `main.cc`:

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada c;
    c.setX(1);
    c.setY(2);
    c.setZ(1);
    cout<< "("<< c.getX()<< ", "<< c.getY()<< ", "<< c.getZ()<< ")"<<
    return 0;
}
```

Compilación

- Compilación de la clase `Coordenada` junto con el programa principal:

```
g++ -std=c++11 -Wall -c Coordenada.cc
g++ -std=c++11 -Wall -c main.cc
g++ -std=c++11 -Wall -o main Coordenada.o main.o
```

Compilación

Fases de la compilacón con `gcc`:

1. Preprocesado: se procesan las directivas que empiezan por `#`: `#include`, `#define`, etc.
   - opción `-E`
2. Compilación y ensamblado: se genera código máquina
   - opción `-S`: compila y no ensambla $\rightarrow$ se puede ver el código ensamblador
   - opción `-c`: compila y ensambla pero no enlaza $\rightarrow$ ficheros con código máquina (`.o`) diferentes para cada clase y programa principal
3. Enlazado: une el código máquina de las diferentes partes (incluidas bibliotecas del sistema, p.ej. `iostream`)

Diseño orientado a objetos

- ¿Se puede hacer este programa más *orientado a objetos*?

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada c;
    c.setX(1);
    c.setY(2);
    c.setZ(1);
    cout<< "("<< c.getX()<< ", "<< c.getY()<< ", "<< c.getZ()<< ")"<<
    return 0;
}
```

Diseño orientado a objetos

- ¿Se puede hacer este programa más *orientado a objetos*? **Sí**.

`main.cc`:

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada c;
    c.setX(1);
    c.setY(2);
    c.setZ(1);
    c.imprimir();
    return 0;
}
```

Diseño orientado a objetos

- ¿Se puede hacer este programa más *orientado a objetos*? **Sí**.
`coordenada.cc`:

```
#include<iostream>
#include "Coordenada.h"
....
void Coordenada::imprimir(){
    cout<< "("<< c.getX()<< ", "<< c.getY();
    cout<< ", "<< c.getZ()<< ")"<< endl;
}
....
```

Compilación

- ¿Qué pasa si tengo 100 clases?:

```
g++ -std=c++11 -Wall -c Coordenada.cc
g++ -std=c++11 -Wall -c OtraClase.cc
g++ -std=c++11 -Wall -c OtraClaseMas.cc
...
g++ -std=c++11 -Wall -c main.cc
g++ -std=c++11 -Wall -o main Coordenada.o ...
```

La herramienta `make`

- `make` es una herramienta que permite automatizar la construcción de unos ficheros a partir de otros (en la mayoría de los casos, para compilar, pero no exclusivamente)
- Las reglas para construir los ficheros se escriben en un fichero llamado `makefile`
- Sólo hay que teclear la orden `make` para que el proyecto se compile

La herramienta `make`

- Hay que definir el fichero que se quiere generar, qué ficheros se necesitan y la orden a ejecutar:

```
Objetivo: Dependencia1 Dependencia2 ...
        orden a ejecutar
```

- Si una dependencia no existe, se busca una regla para generarla, y así se pueden ejecutar cientos de órdenes

```
Dependecia1: Dependencia11 Dependencia12
        orden a ejecutar
Dependecia2: Dependencia21 Dependencia22
        orden a ejecutar
Dependecia21: Dependencia3 Dependencia4
        orden a ejecutar
```

La herramienta `make`

Ejercicio

Escribe un fichero `makefile` para compilar el programa principal que emplea la clase `Coordenada`

La herramienta `make`

Ejercicio

Escribe un fichero `makefile` para compilar el programa principal que emplea la clase `Coordenada`

```
main: main.o Coordenada.o
        g++ -o main Coordenada.o main.o
main.o: main.cc Coordenada.h
        g++ -std=c++11 -Wall -c main.cc
Coordenada.o: Coordenada.cc Coordenada.h
        g++ -std=c++11 -Wall -c Coordenada.cc
```

La herramienta `make`

Ejercicio

Escribe un fichero `makefile` para compilar el programa principal que emplea la clase `Coordenada`

```
main: main.o Coordenada.o
        g++ -o main Coordenada.o main.o
main.o: main.cc Coordenada.h
        g++ -std=c++11 -Wall -c main.cc
Coordenada.o: Coordenada.cc Coordenada.h
        g++ -std=c++11 -Wall -c Coordenada.cc
```

- Problema: demasiada repetición de opciones

La herramienta `make`

- Se pueden emplear variables para agrupar opciones que se van a repetir mucho

```
COMP=g++
OPC=-std=c++11 -Wall
OBJS=main.o Coordenada.o

main: $(OBJS)
        $(COMP) -o main $(OBJS)
main.o: main.cc Coordenada.h
        $(COMP) $(OPC) -c main.cc
Coordenada.o: Coordenada.cc Coordenada.h
        $(COMP) $(OPC) -c Coordenada.cc
```

Sobrecarga de funciones

- Se pueden definir funciones con el mismo nombre, siempre que tengan distinto número, orden y/o tipo de parámetros

```
#include<iostream>
using namespace std;
int minimo(int a, int b){
    if(a < b)
        return a;
    else
        return b;
}
int minimo(int a, int b, int c){
    return min(a, min(b,c));
}
int main(){
    cout << minimo(1,2,3) << endl;
}
```

Sobrecarga de funciones

- Si la única diferencia es el tipo de salida, se producirá un error:

```
#include<iostream>
using namespace std;
int minimo(int a, int b){
    if(a < b)
        return a;
    else
        return b;
}
int minimo(int a, int b, int c){
    return min(a, min(b,c));
}
float minimo(int a, int b, int c){
    return min(a, min(b,c));
}
int main(){
    cout << minimo(1,2,3) << endl;
}
```

- `error: ambiguating new declaration of 'float minimo(int, int, int)`

Forma canónica de una clase

Métodos que debemos implementar obligatoriamente para asegurarnos de que los objetos de la clase se comportan como variables "normales":

- Constructor
- Destructor
- Constructor de copia
- Operador de asignación

Constructor

- Se invoca automáticamente cada vez que se crea un objeto de la clase
- Debe iniciar todo lo necesario para operar con el objeto $\rightarrow$ recordad cuál es el valor de las variables sin inicializar
  - Inicializar los atributos a valores razonables
  - Reservar memoria, si es necesario
- Declaración:
  - Mismo nombre que la clase
  - No devuelve nada (ni `void`)
  - Normalmente en la parte pública
  - Se puede sobrecargar

Constructor

- Constructor por defecto para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
....

//En Coordenada.cc
Coordenada::Coordenada(){
  x=0; y=0; z=0;
}
```

Constructor

- Constructor sobrecargado para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
Coordenada(int, int, int);
....

//En Coordenada.cc
Coordenada::Coordenada(int px, int py, int pz){
  x=px; y=py; z=pz;
}
```

- Uso del nuevo constructor en `main.cc`:

```
Coordenada c(2,4,6);
c.imprimir();
```

Constructor de copia

- Se invoca automáticamente cada vez que se copia un objeto de la clase:
  - Cuando declaramos un objeto a partir de otro
  - Al pasar por valor un objeto de la clase a una función
  - Cuando una función devuelve (por valor) un objeto de la clase
- Debe copiar los atributos de un objeto a otro, o reservar memoria si es necesario
- Si no se declara, el compilador crea uno que simplemente copia los atributos
- Declaración:
  - Debe recibir un objeto de la clase por *referencia constante*

Constructor de copia

- Constructor de copia para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
Coordenada(int, int, int);
Coordenada(const Coordenada&);
....

//En Coordenada.cc
Coordenada::Coordenada(const Coordenada& c){
  x=c.x; y=c.y; z=c.z;
}
```

- Uso del nuevo constructor en `main.cc`:

```
Coordenada c(2,4,6);
Coordenada c2(c);
c.imprimir();
```

Destructor

- Se invoca automáticamente cada vez que se elimina un objeto de la clase:
  - Cuando sale de ámbito
  - Cuando se invoca al operador `delete` sobre un puntero a un objeto de la clase
- Debe asegurarse de liberar todos los recursos asociados al objeto: `delete` si hay memoria dinámica asociada al objeto
- Si no se declara, el compilador crea uno que no hace nada
- Si hay atributos estáticos, no es necesario hacer nada con ellos, se liberarán automáticamente
- Declaración:
  - $\sim$ seguido del nombre de la clase: `~Coordenada()`
  - No recibe ni devuelve nada

Destructor

- Destructor para `Coordenada`:

```
//En Coordenada.h
public:
Coordenada();
Coordenada(int, int, int);
Coordenada(const Coordenada&);
~Coordenada();
....

//En Coordenada.cc
Coordenada::~Coordenada(){
  x=0; y=0; z=0;
}
```

Constructores, destructores y memoria dinámica

Ejercicio

Imagina que imprimimos un mensaje en el constructor y destructor de `Coordenada`. ¿Qué se imprimiría al ejecutar este programa?

```
#include<iostream>
#include "Coordenada.h"
using namespace std;
int main(){
    Coordenada array[3];
    Coordenada* ptC=nullptr;

    cout << "Reserva memoria" << endl;
    ptC= new Coordenada[3];
    if(ptC == nullptr)
        return 1;

    cout << "Fin del programa" << endl;
    return 0;
}
```

El puntero `this`

- Sólo se puede usar dentro de métodos de una clase, no dentro de funciones
- Contiene la dirección de memoria del objeto que ha invocado al método:

```
Coordenada::Coordenada(int x, int y, int z){
    this->x=x;
    this->y=y;
    this->z=z;
}
```

El puntero `this`

- También se emplea en operadores, para permitir aplicarlos en cascada: `a=b=c`
- Y para evitar que un objeto se asigne a sí mismo
- Es de sólo lectura
  - El siguiente código produciría un error de compilación:

```
Coordenada::Coordenada(const Coordenada &c){
     this=&c;
}
```

El modificador `const`

- Para declarar constantes:

```
const int TAMANO=100;
```

- Para indicar que un parámetro pasado por referencia no va a cambiar:

```
Coordenada::Coordenada(const Coordenada &c){
     x=c.getX();
     ....
```

- Para indicar que un método de una clase no modifica el objeto:

```
class Coordenada{
...
public:
int getX() const;
int getY() const;
int getZ() const;
...
```

El modificador `const`

- Un objeto constante solo puede invocar a métodos marcados con `const`
- Un objeto no constante puede invocar a cualquier método
- El siguiente código fallaría si `getX` no se marca como método constante:

```
Coordenada::Coordenada(const Coordenada &c){
     x=c.getX();
     ....
```

El modificador `const`

- Hay que marcar el método como `const` tanto en el fichero `.h` como en el `.cpp`:

```
//Coordenada.h
class Coordenada{
...
public:
int getX() const;
...

//Coordenada.cc
int Coordenada::getX() const{
   return x;
```

- Un método marcado como `const` no puede modificar el objeto. Error de compilación:

```
//Coordenada.cc
int Coordenada::getX() const{
   x--;
   return x;
}
```

Sobrecarga de operadores

- Sería interesante poder sumar coordenadas empleando el mismo símbolo + que para tipos básicos:

```
Coordenada a(1,2,3);
Coordenada b(4,5,6);
Coordenada c;
c=a+b;
```

Sobrecarga de operadores

- Sería interesante poder sumar coordenadas empleando el mismo símbolo + que para tipos básicos:

```
Coordenada a(1,2,3);
Coordenada b(4,5,6);
Coordenada c;
c=a+b;
```

- Se puede conseguir implementando métodos `operator<símbolo>`:

```
// Equivale a c=a+b;
c.operator=( a.operator+(b) );
```

Operador de asignación

- Es recomendable sobrecargarlo siempre → forma canónica
- Si no se define, se produce una copia atributo a atributo que puede dar problemas con memoria dinámica
- Devuelve un objeto de la clase *por referencia*: siempre `*this`
  - Para poder hacer: `a=b=c` → `a.operator=(b.operator=(c))`
- Recibe un objeto de la clase por referencia constante

Operador de asignación

- `Coordenada.h`:

```
public:
Coordenada& operator=(const Coordenada&);
```

- `Coordenada.cc`:

```
Coordenada& Coordenada::operator=(const Coordenada& c){
    (*this).~Coordenada();
    x=c.x;
    y=c.y;
    z=c.z;
    return (*this);
}
```

- Atención: error si intentamos hacer `a=a;`

Operador de asignación

- Solución: comprobar la dirección de memoria para evitar auto-asignación

```
Coordenada& Coordenada::operator=(const Coordenada& c){
    if(this != &c){
        (*this).~Coordenada();
        x=c.x;
        y=c.y;
        z=c.z;
    }
    return (*this);
}
```

Operadores aritméticos

- No modifican a los operandos y devuelven un nuevo objeto
- Devuelven el tipo del objeto por valor
- Es recomendable que el método sea marcado como constante (operando izquierdo) y el argumento sea una referencia constante (operando derecho)
- Normalmente implican crear un objeto temporal

Operadores aritméticos

- Coordenada.h:

```
public:
Coordenada operator+(const Coordenada&) const;
```

- Coordenada.cc:

```
Coordenada Coordenada:::operator+(const Coordenada& c) const{
   Coordenada cn;
   cn.setX(x+c.x)
   cn.setY(y+c.y)
   cn.setZ(z+c.z)
   return cn;
}
```

Operadores de comparación

- Devuelven un dato de tipo `bool`
- No modifican los operandos
- `Coordenada.h`:

```
public:
bool operator==(const Coordenada&) const;
```

- `Coordenada.cc`:

```
bool Coordenada::operator==(const Coordenada& c) const{
   return (x==c.x && y==c.y && z=c.z)
}
```

Índice

Relaciones entre clases en C++

- Asociación
- Todo-parte
  - Agregación
  - Composición
- Herencia

Asociación

- Relación (unidireccional o bidireccional) entre los objetos
- Los objetos existen de forma independiente
- Si uno de los dos objetos se destruye, el otro se mantiene
- Ejemplo: relación entre una casilla y una pieza en un juego de ajedrez
- Se implementa en C++ como un puntero:

```
class Pieza{
private:
Casilla* casilla;
public:
....
};
```

Agregación

- Todo-parte: un objeto está formado por varios objetos de otro tipo
- Los objetos “parte” pueden pertenecer a varios objetos “todo”
- Si el objeto “todo” se destruye, los objetos “parte” siguen existiendo
- Ejemplo: relación entre un equipo y sus miembros
- Se implementa en C++ como una colección de punteros:

```
class Equipo{
private:
Persona* miembros[MAX_MIEMBROS];
public:
....
};
```

Composición

- Todo-parte: un objeto está formado por varios objetos de otro tipo
- Los objetos “parte” no pueden pertenecer a varios objetos “todo“ ni existir al margen del mismo
- Si el objeto “todo” se destruye, los objetos “parte” también deben destruirse
- Ejemplo: relación entre un libro y sus capítulos
- Se implementa en C++ mediante layering:

```
class Libro{
private:
Capitulo introduccion;
Capitulo cap1;
...
public:
....
};
```

Herencia

- Es-un: una clase es un subtipo o especialización de otra clase
- La clase derivada `hereda` los atributos y métodos de la clase base y añade los suyos propios
- Ejemplo: en un sistema de gestión de datos médicos, un paciente (clase derivada) es una persona (clase base)
- Mecanismo muy potente en C++ que no vamos a estudiar en la asignatura

Layering

- Mecanismo que permite incluir objetos de una clase dentro de otro objeto
- Ejemplo: `Linea.h`:

```
class Linea{
private:
Coordenada c1, c2;
public:
Linea();
Linea(const Coordenada&, const Coordenada&);
Linea(const Linea&)
....
};
```

Layering

- Constructores: se ejecuta automáticamente el código del constructor por defecto de `Coordenada` antes de cualquier constructor de `Linea`
  - Excepto si se usa un *inicializador* en el fichero `.cc`

```
Linea::Linea(const Coordenada& a,const Coordenada& b ):c1(a),c2(b
....
Linea::Linea(const Linea& l ):c1(l.c1),c2(l.c2){
....
```

- Destructor: se ejecuta automáticamente el código del destructor `Coordenada` después de destructor de `Linea`

Guardas de inclusión

- `Linea.h`:

```
#include "Coordenada.h"
```

- `main.cc`:

```
#include "Linea.h"
#include "Coordenada.h"
```

- Resultado: error de compilación porque `Coordenada` se ha definido dos veces

Guardas de inclusión

- En cada fichero `.h`:

```
#ifndef __NOMBRE_CLASE__
#define __NOMBRE_CLASE__
...
#endif
```

- `Coordenada.h`:

```
#ifndef __COORDENADA__
#define __COORDENADA__

class Coordenada{
...

};

#endif
```

Índice

Tipos abstractos de datos

- Concepto precursor de la programación orientada a objetos
- **Tipo de datos**: clasifica los objetos de los programas (variables, parámetros, constantes) y determina
  - Los valores que pueden tomar
  - Las operaciones que se pueden realizar
- **Abstracto**: los usamos en nuestros programas según su **especificación** (qué hace) e independientemente de su **implementación**
- Una especificación $\rightarrow$ múltiples implementaciones

Tipos abstractos de datos (TADs)

- Especificación: defínción de las propiedades y operaciones
  - Especificaciones algebraicas formales (en desuso)
  - Especificaciones informales en lenguaje natural: operaciones (métodos de la clase) $+$ explicación de lo que hacen
- Implementación:
  - Determinar la representación
  - Codificar las operaciones en base a esa representación

Ejemplo

Especificación del TAD `Hora`

Definición: una instancia del TAD `Hora` almacena una hora del día. La hora puede corresponder a cualquier fecha, y no nos interesa almacenar ni representar la fecha. Emplearemos una precisión de segundos, por que nos interesará saber la hora en términos de horas, minutos y segundos, por ejemplo, `13:24:06`. La hora por defecto será `00:00:00`.

Ejemplo

Especificación del TAD `Hora`

Operaciones:

- Modifica la hora siempre que: el parámetro `h`, que representa las horas, esté entre 0 y 23; el parámetro `m`, que representa los minutos, esté entre 0 y 59; y el parámetro `s`, que representa los segundos, esté entre 0 y 59. Si algún parámetro no es correcto, no modifica la hora y devuelve `false`.

```
bool modificaHora(int h, int m, int s);
```

- Devuelve la hora completa en formato `hh:mm:ss`.

```
string getHoraCompleta();
```

Ejemplo

Especificación del TAD `Hora`

Operaciones:

- Devuelve la hora del día (entre 0 y 23)

```
int getHoras();
```

- Devuelve los minutos trasncurridos desde el comienzo de la última hora. Por ejemplo, `24` para las `13:24:06`

```
int getMinutos();
```

- Devuelve los segundos trasncurridos desde el comienzo del último minuto. Por ejemplo, `6` para las `13:24:06`

```
int getSegundos();
```

Ejemplo

Especificación del TAD `Hora`

Operaciones:

- Incrementa la hora en `s` segundos. Si la hora sobrepasa el final del día, vuelve a comenzar tantas veces como sea necesario. Si el parámetro `s` es negativo, la hora no se modifica y se devuelve `false`. En caso contrario, se devuelve `true`

```
bool incrementa(int s);
```

Ejemplo

Implementación del TAD `Hora`

- `Hora.h`:

```
class Hora{
private:
?????
public:
Hora();
Hora(const Hora&);
~Hora();
Hora& operator=(const Hora&)
bool modificaHora(int h, int m, int s);
string getHoraCompleta() const;
int getHoras() const;
int getMinutos() const;
int getSegundos() const;
bool incrementa(int);
};
```

Ejercicio

Implementa el TAD `Hora` eligiendo la mejor **representación**