Sobrecarga de operadores, listas

- mat-151

Problema:

Problema:

- Dadas 2 listas no vacias **`link lista1, lista2`**

Problema:

- Dadas 2 listas no vacias **`link lista1`**, **`lista2`**
- dado un apuntador a un nodo de la **`lista1`**, **`link t`**, que apunta a un nodo intermedio de dicha lista

Problema:

- Dadas 2 listas no vacias **`link lista1`**, **`lista2`**
- dado un apuntador a un nodo de la **`lista1`**, **`link t`**, que apunta a un nodo intermedio de dicha lista
- Esquematizar el algoritmo para insertar la **`lista2`** después del nodo al que apunta **`t`**. (3 minutos)

Problema:

- Dadas 2 listas no vacias **`link lista1`**, **`lista2`**
- dado un apuntador a un nodo de la **`lista1`**, **`link t`**, que apunta a un nodo intermedio de dicha lista
- Esquematizar el algoritmo para insertar la **`lista2`** después del nodo al que apunta **`t`**. (3 minutos)
- Para este ejercicio no es necesario crear nuevos nodos, sino usar los de la **`lista2`**.

Problema:

- Dadas 2 listas no vacias **`link lista1`**, **`lista2`**
- dado un apuntador a un nodo de la **`lista1`**, **`link t`**, que apunta a un nodo intermedio de dicha lista
- Esquematizar el algoritmo para insertar la **`lista2`** después del nodo al que apunta **`t`**. (3 minutos)
- Para este ejercicio no es necesario crear nuevos nodos, sino usar los de la **`lista2`**.

Problema:

- Dadas 2 listas no vacias **`link lista1`**, **`lista2`**
- dado un apuntador a un nodo de la **`lista1`**, **`link t`**, que apunta a un nodo intermedio de dicha lista
- Esquematizar el algoritmo para insertar la **`lista2`** después del nodo al que apunta **`t`**. (3 minutos)
- Para este ejercicio no es necesario crear nuevos nodos, sino usar los de la **`lista2`**.

Problema:

- Dadas 2 listas no vacias **`link lista1`**, **`lista2`**
- dado un apuntador a un nodo de la **`lista1`**, **`link t`**, que apunta a un nodo intermedio de dicha lista
- Esquematizar el algoritmo para insertar la **`lista2`** después del nodo al que apunta **`t`**. (3 minutos)
- Para este ejercicio no es necesario crear nuevos nodos, sino usar los de la **`lista2`**.
- Hacer el algoritmo en C++ (5 minutos)

ADTs

ADTs

- Un principio básico de la ingeniería de software:

ADTs

- Un principio básico de la ingeniería de software:

    - *Separar la **interfase** (**qué** podemos hacer) de la **implementación** (**cómo** lo vamos a hacer.)*

ADTs

- Un principio básico de la ingeniería de software:

    - *Separar la **interfase** (**qué** podemos hacer) de la **implementación** (**cómo** lo vamos a hacer.)*

- Un **tipo de datos abstracto** es una **interfase** a una colección de datos.

ADTs

- Un principio básico de la ingeniería de software:

    •*Separar la **interfase** (**qué** podemos hacer) de la **implementación** (**cómo** lo vamos a hacer.)*

- Un **tipo de datos abstracto** es una **interfase** a una colección de datos.

- Puede haber varias formas de implementar un **ADT**, cada una con diferentes características de desempeño.

ADTs

- Un principio básico de la ingeniería de software:

    •*Separar la **interfase** (**qué** podemos hacer) de la **implementación** (**cómo** lo vamos a hacer.)*

- Un **tipo de datos abstracto** es una **interfase** a una colección de datos.

- Puede haber varias formas de implementar un **ADT**, cada una con diferentes características de desempeño.

- Un **ADT** consta de:

ADTs

- Un principio básico de la ingeniería de software:

    •*Separar la **interfase** (**qué** podemos hacer) de la **implementación** (**cómo** lo vamos a hacer.)*

- Un **tipo de datos abstracto** es una **interfase** a una colección de datos.

- Puede haber varias formas de implementar un **ADT**, cada una con diferentes características de desempeño.

- Un **ADT** consta de:

    ➡ algunos **tipos**, necesarios para proporcionar operaciones, relaciones y satisfacer propiedades.

ADTs

- Un principio básico de la ingeniería de software:

    •*Separar la **interfase** (**qué** podemos hacer) de la **implementación** (**cómo** lo vamos a hacer.)*

- Un **tipo de datos abstracto** es una **interfase** a una colección de datos.

- Puede haber varias formas de implementar un **ADT**, cada una con diferentes características de desempeño.

- Un **ADT** consta de:

    ➡ algunos **tipos**, necesarios para proporcionar operaciones, relaciones y satisfacer propiedades.

    ➡ una interfase: las capacidades proporcionadas por la ADT

ADTs

Definición.

> Un tipo de datos abstracto **ADT** es un tipo de datos (conjunto de valores y colección de operaciones sobre estos valores) que es accesible **solamente** a través de una **interfase**.

- con un ADT los programas clientes no tienen acceso a ningún valor de los datos excepto a través de las operaciones que proporciona su **interfase**.
- La representación de los datos y las funciones que implementan las operaciones están en la **implementación**, que está completamente separada del cliente, por la interfase.

Ejemplo: representación de puntos

Ejemplo: representación de puntos

- la razón principal para crear un tipo de datos **punto**, es facilitar al programa ***cliente*** el acceso a las coordenadas individuales que necesite.

Ejemplo: representación de puntos

- la razón principal para crear un tipo de datos **punto**, es facilitar al programa ***cliente*** el acceso a las coordenadas individuales que necesite.

- definimos una **struct point**.

Ejemplo: representación de puntos

- la razón principal para crear un tipo de datos **punto**, es facilitar al programa ***cliente*** el acceso a las coordenadas individuales que necesite.

- definimos una **struct point**.

- sin cambiar tanto la **struct**, siendo el programa **cliente** no podemos modificar la representación de punto (coordenadas polares, tres dimensiones, el uso de diferentes tipos (int, float, etc) para las coordenadas, etc.)

Ejemplo: representación de puntos

- la razón principal para crear un tipo de datos **punto**, es facilitar al programa ***cliente*** el acceso a las coordenadas individuales que necesite.

- definimos una **struct point**.

- sin cambiar tanto la **struct**, siendo el programa **cliente** no podemos modificar la representación de punto (coordenadas polares, tres dimensiones, el uso de diferentes tipos (int, float, etc) para las coordenadas, etc.)

- una implementación diferente del tipo de datos abstracto ***point*** en un lenguaje orientado a objetos puede hacerse con una **clase**.

Ejemplo: representación de puntos

Ejemplo: representación de puntos

- Cuando escribimos una definición como ***int i*** en un programa, damos la instrucción al sistema de reservar espacio para los datos de un tipo ***int***, al que nos referimos por el nombre ***i***.

Ejemplo: representación de puntos

- Cuando escribimos una definición como ***int i*** en un programa, damos la instrucción al sistema de reservar espacio para los datos de un tipo ***int***, al que nos referimos por el nombre ***i***.
- En un lenguaje orientado a objetos, a este tipo de elemento le llamamos ***objeto***.

Ejemplo: representación de puntos

- Cuando escribimos una definición como ***int i*** en un programa, damos la instrucción al sistema de reservar espacio para los datos de un tipo ***int***, al que nos referimos por el nombre ***i***.
- En un lenguaje orientado a objetos, a este tipo de elemento le llamamos ***objeto***.
- Cuando escribimos una definición como ***point p*** en un programa, decimos que ***creamos*** un objeto de la ***clase*** point, al que nos referimos por el nombre ***p***.

Ejemplo: representación de puntos

- Cuando escribimos una definición como ***int i*** en un programa, damos la instrucción al sistema de reservar espacio para los datos de un tipo ***int***, al que nos referimos por el nombre ***i***.
- En un lenguaje orientado a objetos, a este tipo de elemento le llamamos ***objeto***.

- Cuando escribimos una definición como ***point p*** en un programa, decimos que ***creamos*** un objeto de la ***clase*** point, al que nos referimos por el nombre ***p***.
- Cada **objeto** de la clase ***p*** tiene dos elementos de datos llamados ***x*** y ***y***, que al igual que en las estructuras podemos escribir como: **p.x** y **p.y**

Ejemplo: representación de puntos

- Cuando escribimos una definición como ***int i*** en un programa, damos la instrucción al sistema de reservar espacio para los datos de un tipo ***int***, al que nos referimos por el nombre ***i***.
- En un lenguaje orientado a objetos, a este tipo de elemento le llamamos ***objeto***.

- Cuando escribimos una definición como ***point p*** en un programa, decimos que ***creamos*** un objeto de la ***clase*** point, al que nos referimos por el nombre ***p***.
- Cada **objeto** de la clase ***p*** tiene dos elementos de datos llamados ***x*** y ***y***, que al igual que en las estructuras podemos escribir como: **p.x** y **p.y**
- Nos referimos a los elementos ***x*** y ***y*** como **datos miembros** de la clase.

implementación de la clase point

point.cpp

```
#include <cmath>

class point
{
    private:
        float x,y;
    public:
        point() { // constructor
            x=1.0*rand()/RAND_MAX;
            y=1.0*rand()/RAND_MAX;
        }


        float distance( point a ){
            float dx = x-a.x;
            float dy = y-a.y;
            return( sqrt(dx*dx + dy*dy));
        }
}; // fin de la clase
```

ejemplo de cliente para la clase point (el número de puntos es un argumento)

```
#include <iostream>
#include "point.cpp"

int main( int argc, char *argv[]){
    float d=atof(argv[2]);
    int i, cnt=0, n=atoi(argv[1]);
    point *a = new point[n];
    for( i=0; i<n; i++ )
        for( int j=i+1; j<n; j++ )
            if( a[i].distance(a[j]) < d) cnt++;
    cout << cnt << "pares dentro" << d << endl;
}
```

ejemplo de interfase ADT para point

```
class point
{
    private:
        // codigo dependiente de la implementacion
    public:
        point();
        float distance(point) const;
};
```

ADTs

ADTs

- nos liberan de conocer su implementación.

ADTs

- nos liberan de conocer su implementación.
- muy conveniente para *modularizar* software.

ADTs

- nos liberan de conocer su implementación.
- muy conveniente para *modularizar* software.
- en cuanto al análisis del desempeño, nos interesa saber el costo de las operaciones básicas.

ADTs

- nos liberan de conocer su implementación.
- muy conveniente para *modularizar* software.
- en cuanto al análisis del desempeño, nos interesa saber el costo de las operaciones básicas.
- El mismo principio funciona para diferentes niveles de abstracción: Queremos identificar las **operaciones críticas** de nuestros **programas** y las **características críticas** de nuestros **datos.**

ADTs

ADTs

- Los conjuntos manipulados por algoritmos pueden crecer, decrecer, o cambiar en el tiempo. Estos se llaman conjuntos **dinámicos**.

ADTs

- Los conjuntos manipulados por algoritmos pueden crecer, decrecer, o cambiar en el tiempo. Estos se llaman conjuntos **dinámicos**.
- Hay algoritmos que pueden requerir diferentes tipos de **operaciones** en los **conjuntos**. Por ejemplo, insertar elementos, borrar elementos, probar si un elemento es parte de un conjunto, etc.

ADTs

- Los conjuntos manipulados por algoritmos pueden crecer, decrecer, o cambiar en el tiempo. Estos se llaman conjuntos **dinámicos**.
- Hay algoritmos que pueden requerir diferentes tipos de **operaciones** en los **conjuntos**. Por ejemplo, insertar elementos, borrar elementos, probar si un elemento es parte de un conjunto, etc.
- Un conjunto **dinámico** que soporta operaciones para insertar, borrar o cambiar nodos se llama **diccionario.**

ADTs

- Los conjuntos manipulados por algoritmos pueden crecer, decrecer, o cambiar en el tiempo. Estos se llaman conjuntos **dinámicos**.
- Hay algoritmos que pueden requerir diferentes tipos de **operaciones** en los **conjuntos**. Por ejemplo, insertar elementos, borrar elementos, probar si un elemento es parte de un conjunto, etc.
- Un conjunto **dinámico** que soporta operaciones para insertar, borrar o cambiar nodos se llama **diccionario.**
- En una implementación típica de un diccionario, cada elemento se representa por un objeto cuyos campos pueden ser examinados y manipulados si existe un **apuntador** al objeto.

ADTs

- Los ADTs son muy utilizados para comparar las características de desempeño de algoritmos y estructuras de datos.

- Las usaremos el resto del curso.

- Ejemplos: pilas, colas, grafos, árboles, etc.

Constructores que no reciben parámetros

```
class A{
public:
    int a,b;
    A(){

            a = 5;
            b = 8;
    }

};

void main (int argc, char * const argv[]) {
    A *var = new A[10];

    std::cout << "Datos miembro = " << var[3].a << " " << var[3].b << "\n";
}
```

Constructores que reciben parámetros (esto marca error de compilación en la linea del `new`)

```
class A{
public:
    int a,b;
    A(int _a , int _b){

            a = _a;
            b = _b;
    }

};


  void main (int argc, char * const argv[]) {
      A *var = new A[10];

      std::cout << "Datos miembro = " << var[3].a << " " << var[3].b << "\n";
  }
```

Constructores que reciben parámetros por default

```
class A{
public:
    int a,b;
    A(int _a = 5 , int _b = 8){

            a = _a;
            b = _b;
    }

};
```

```
void main (int argc, char * const argv[]) {
    A *var = new A[10];

    std::cout << "Datos miembro = " << var[3].a << " " << var[3].b << "\n";
}
```

Sobrecarga de operadores

```
class Vector2D{
    private:
        float x,y;
    public:
        void set_x(float val);
        void set_y(float val);
        float get_x(void) const { return x;}
        float get_y(void) const { return y;}
};
```

Sobrecarga de operadores

```
// binary operator as member function
Vector2D Vector2D::operator+(const Vector2D& right) {...}

// binary operator as non-member function
Vector2D operator+(const Vector2D& left, const Vector2D& right)
                   {...}

// unary operator as member function
Vector2D Vector2D::operator-() {...}

// unary operator as non-member function
Vector2D operator-(const Vector2D& vec) {...}
```

Sobrecarga de operadores

Ejemplo:

```
Vector2D operator+(const Vector2D& left, const Vector2D& right)
{
    Vector2D result;
    result.set_x(left.get_x() + right.get_x());
    result.set_y(left.get_y() + right.get_y());
    return result;
}
```

Sobrecarga de operadores

```
class Vector2D{
    private:
        float x,y;
    public:
        void set_x(float val);
        void set_y(float val);
        float get_x(void) const { return x;}
        float get_y(void) const { return y;}
        friend ostream& operator<<(ostream& out, const Vector2D& vec);
        friend istream& operator>>(istream& in, Vector2D& vec);
};
```

Sobrecargando flujo de entrada/salida

```
// output
friend ostream& operator<<(ostream& out, const Vector2D& vec)
{
    out << "(" << vec.x << ", " << vec.y << ")";
    return out;
}
```

```
friend istream& operator>>(istream& in, Vector2D& vec) // input
{
    double x, y;
    in >> x >> y;
    vec.x=x;
    vec.y=y;
    return in;
}
```

Funciones *friend* con 2 clases

```
// forward declaration
class B;
class A {
    private:
      int numA;
    public:
      A(): numA(12) { }
      // friend function declaration
      friend int add(A, B);
};

class B {
    private:
       int numB;
    public:
       B(): numB(1) { }
       // friend function declaration
       friend int add(A , B);
};
```

Funciones *friend* con 2 clases

```
// Function add() is the friend function of classes A and B
// that accesses the member variables numA and numB
int add(A objectA, B objectB)
{
    return (objectA.numA + objectB.numB);
}

int main()
{
    A objectA;
    B objectB;
    cout<<"Sum: "<< add(objectA, objectB);
    return 0;
}
```