Estructura de Datos

C++: Standard Template Library

John Corredor Franco, PhD
Departamento de Ingeniería de Sistemas

Enero, 2024

Agenda

- Abstract Data Types
- Generic Programming in C++
- Templates in C++
- Standard Template Library STL

Representación del objeto geométrico círculo

Estado:

- m_Radius: R
  - m_Radius >= 0
- m_CenterX: R
- m_CenterY: R

Interface:

- Circle()
  - Post: Círculo m_Radius = 0, m_CenterX = m_CenterY = 0;
- GetArea(): R
  - Pre: m_Radius >= 0
  - Post: area = π*m_Radius*m_Radius

TAD Punto

Datos mínimos:

- **coordenada**, arreglo de 3 reales, representa la coordenada tridimensional del punto.
- **color**, arreglo de 3 enteros cortos, representa el color con el que se dibuja el punto, en niveles de rojo, verde y azul.

Operaciones:

- **obtener_coordenada()**, retorna la coordenada tridimensional actual del punto.
- **obtener_color()**, retorna el color actual del punto.
- **fijar_coordenada(ncoord)**, fija la nueva coordenada tridimensional para el punto.
- **fijar_color(ncolor)**, fija el nuevo color para el punto.

Quiz
Diseñar TAD's para modelar Polígonos

- Triángulo.
- Cuadrado.
- Pentágono.
- Círculo.
  - Cálculo de áreas y perímetros.
  - Cambio de puntos.

a = b + c

En C++, ¿Qué problemas tiene la línea de código?

```
int b = 5;
int c = 10;
int a = b + c;
```

➢ Generic Programming

```cpp
#include <iostream>

using namespace std;

/*Declaración/implementación de funciones*/
int Suma(int a, int b){
    return a + b;
}

float Suma(float a, float b){
    return a + b;
}

string Suma(const string& a, const string& b){
    string s = a + b;
    return s;
}
```

```cpp
int main(){
    int i1 = 1; int i2 = -5;
    cout << Suma(i1, i2) << endl;
    float f1 = 0.02; float f2 = 5.98;
    cout << Suma(f1, f2) << endl;
    const char* c1 = "a"; const char *c2 = "z";
    cout << Suma(c1, c2) << endl;

    return 0;
}
```

➢ Templates in C++

```cpp
#include <iostream>

using namespace std;

/*Plantilla principal*/
template <class T> T Suma(const T &x, const T &y){
    return (x + y);
};

int main(){
    cout << Suma<int>(20,23) << endl;
    cout << Suma<float>(0.23,5.66) << endl;
    cout << Suma<long>(2,6) << endl;

    return 0;
}
```

➢ Templates in C++

```
1 #include <iostream>
2
3 using namespace std;
4
5 /*Plantilla principal*/
6 template <typename T> T myMax(T x, T y){
7     return (x > y)?x:y;
8 };
9
10 int main(){
11     cout << myMax<int>(20,23) << endl;
12     cout << myMax<char>('e','r') << endl;
13
14     return 0;
15 }
```

C++: Organización de una Plantilla

- Tipos en general
- Números enteros

```
template <class T, unsigned int N> class Vector{

        protected:
                T m_Data[N]

}
```

C++: Argumentos de una Plantilla

- Tipos en general
- Números enteros

```
template <class T, unsigned int N> class Vector{

        protected:
                T m_Data[N]

}
```

➢ Templates in C++

```cpp
#include <iostream>

using namespace std;

template <class T> class info{
    public:
        /*constructor del tipo template*/
        info(T A) {
            cout << "A = \t" << A << "\t size of data: " << sizeof(A) << "\t bytes\n";
        }
};

int main(){

    info<char>  p('x');
    info<int>   h(11);
    info<float> d(3.14159);

    return 0;
}
```

C++: Organización de una plantilla

- Encabezado (.h)
- Incluir archivo de código (.hxx)

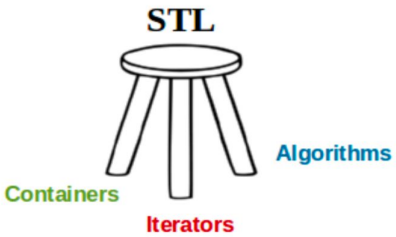
  ESTOS DOS ARCHIVOS NO SE COMPILAN.
- Se usan en un archivo compilable (.cxx, .cpp, .c++) donde se INSTANCIAN las clases genéricas.

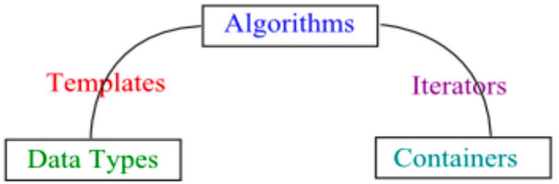
Preprocesado -> Compilación -> Enlazado

Standard Template Library STL

- Biblioteca de “cosas” genéricas cplusplus.com

  - Librería: conjunto de herramientas
    - ¿Caja de herramientas? (*toolkit*)
  - “Cosas”
    - Algoritmos y datos

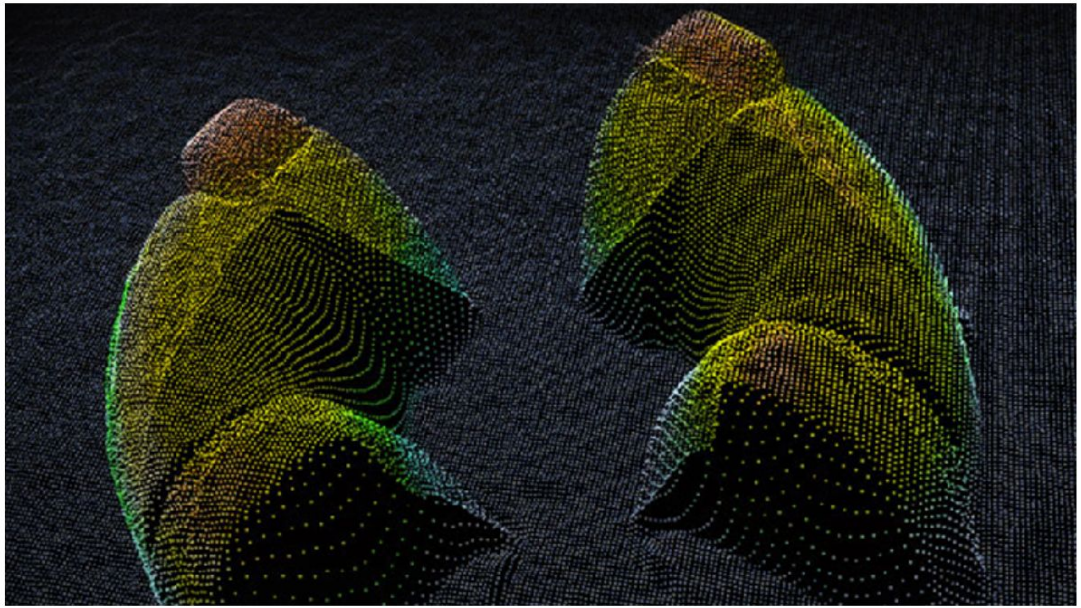




1. Templates
   make algorithms independent of the data types
2. Iterators
   make algorithms independent of the containters

Tomado de:
http://www.bogotobogo.com/cplusplus/

ADT: Tipos Abstractos de Datos (TAD) EJERCICIO

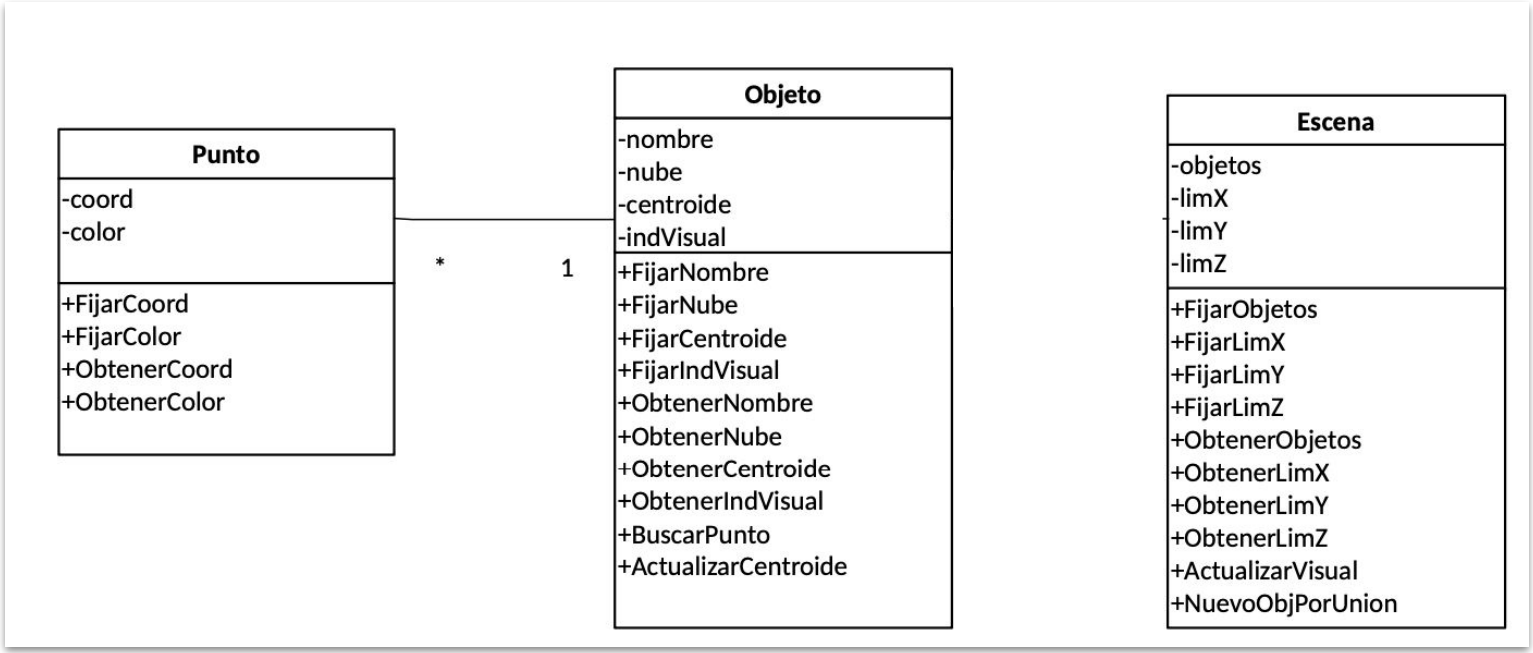


https://metrology.news/time-of-flight-camera-generates-3d-point-cloud/

Diseñe el sistema y el (los) TAD(s) solicitado(s). Utilice la plantilla de especificación de TADs vista en clase para el diseño. Recuerde que diseñar es un proceso previo a la implementación, por lo que no debería contener ninguna referencia a lenguajes de programación (es decir, si escribe encabezados o código fuente, el punto no será evaluado y tendrá una calificación de cero). Para simplicidad del diseño, no es necesario incluir los métodos obtener y fijar (get/set) del estado de cada TAD.

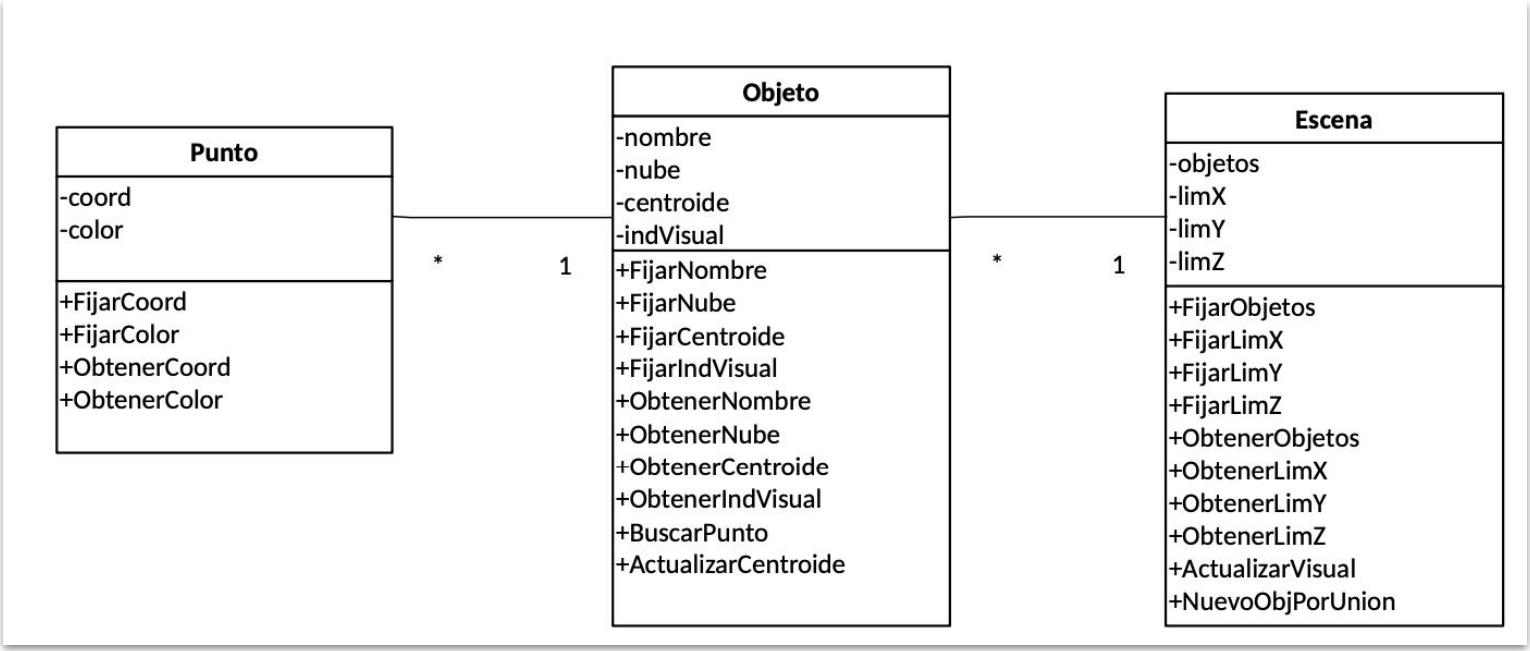
ADT: Tipos Abstractos de Datos (TAD) EJERCICIO

Diagrama de Relación



ADT: Tipos Abstractos de Datos (TAD) EJERCICIO

Diagrama de Relación



Actualizar indicador de visualización.

Dado el (los) TAD(s) ya diseñado(s), escriba la implementación en C++ del algoritmo que permite actualizar el indicador de visualización de un objeto dado. La implementación deberá tener en cuenta:

- la definición apropiada de los prototipos de los métodos/funciones (i.e. recibir/retornar los datos suficientes y necesarios para su correcta ejecución),
- el NO uso de salidas/entradas por pantalla/teclado (es decir, paso/retorno correcto de valores y/o objetos),
- el correcto uso del diseño definido en el punto anterior, y
- la escritura de todo el código que pueda llegar a necesitar que no esté incluido en la STL.

```
Pasos a seguir:
- Extraer los objetos de la estructura de la escena, cuidando, si es el caso, de
  restituirlos adecuadamente al terminar.
- Verificar objeto por objeto aquel que coincida con el nombre dado (o las
  características dadas).
- Al encontrar el objeto, recorrer su nube de puntos, verificando la situación de
  cada punto con respecto a los límites de la escena.
- De acuerdo a lo encontrado para todos los puntos, actualizar el indicador de
  visualización del objeto dado y restituírlo en la estructura de la escena.
```

```
bool Escena::ActualizarVisual (string nomObj) {
  std::stack<Objeto> aux;
  Objeto miObj;
  std::list<Punto> miNube;
  std::list<Punto>::iterator punIt;
  int visualTr, visualFl;
  float miCoor[3];
  bool encontrado = false;
  bool actualizado = false;

  while (!objetos.empty() && !encontrado) {
    miObj = objetos.top();
    objetos.pop();

    if (miObj.ObtenerNombre() == nomObj) {
      encontrado = true;
      miNube = miObj.ObtenerNube();
      visualTr = 0;
      visualFl = 0;

      for (punIt = miNube.begin(); punIt != miNube.end(); punIt++) {
        miCoor = punIt->ObtenerCoord();
        if (miCoor[0] >= limX[0] && miCoor[0] <= limX[1] &&
            miCoor[1] >= limY[0] && miCoor[1] <= limY[1] &&
            miCoor[2] >= limZ[0] && miCoor[2] <= limZ[1]) {
          visualTr++;
        } else {
          visualFl++;
        }
      }

      if (visualTr == miNube.size() && visualFl == 0) {
        miObj.FijarIndVisual("completa");
      } else if (visualTr == 0 && visualFl == miNube.size()) {
        miObj.FijarIndVisual("nula");
      } else {
        miObj.FijarIndVisual("parcial");
      }
      actualizado = true;
    }
    aux.push(miObj);
  }

  while (!aux.empty()) {
    miObj = aux.top();
    aux.pop();
    objetos.push(miObj);
  }

  return actualizado;
}
```

Pasos a seguir:

- Extraer los objetos de la estructura de la escena, cuidando, si es el caso, de restituirlos adecuadamente al terminar.
- Verificar objeto por objeto aquel que coincida con el nombre dado (o las características dadas).
- Al encontrar el objeto, recorrer su nube de puntos, verificando la situación de cada punto con respecto a los límites de la escena.
- De acuerdo a lo encontrado para todos los puntos, actualizar el indicador de visualización del objeto dado y restituírlo en la estructura de la escena.

Unión de dos objetos

Dado el (los) TAD(s) ya diseñado(s), escriba la implementación en C++ del algoritmo que permite crear un nuevo objeto a partir de la unión de las nubes de puntos de dos objetos dados en la escena. Así como en el punto anterior, la implementación deberá tener en cuenta:

- la definición apropiada de los prototipos de los métodos/funciones (i.e. recibir/retornar los datos suficientes y necesarios para su correcta ejecución),
- el NO uso de salidas/entradas por pantalla/teclado (es decir, paso/retorno correcto de valores y/o objetos),
- el correcto uso del diseño definido en el punto anterior, y
- la escritura de todo el código que pueda llegar a necesitar que no esté incluido en la STL.

```
Pasos a seguir:
- Extraer los objetos de la estructura de la escena, cuidando, si es el caso, de
  restituirlos adecuadamente al terminar.
- Verificar objeto por objeto hasta encontrar aquellos que coincidan con los
  nombres dados (o las características dadas).
- Al encontrar los objetos, ir pasando cada uno de los puntos al nuevo objeto,
  primero los del objeto más cercano, y luego los del otro objeto para no repetirlos.
- Calcular el centroide del nuevo objeto y asignarlo.
- Actualizar su indicador de visualización utilizando la función ya desarrollada.
- Agregar el nuevo objeto en el fondo de la estructura, para asegurar que sea el
  último en dibujarse en la escena.
```

```
bool Escena::NuevoObjPorUnion (string nomObj1, string nomObj2) {
  std::stack<Objeto> aux, objs;
  Objeto miObj, miObjU;

  while (!objetos.empty()) {
    miObj = objetos.top();
    objetos.pop();

    if (miObj.ObtenerNombre() == nomObj1 || miObj.ObtenerNombre() == nomObj2) {
      objs.push(miObj);
    }

  aux.push(miObj);
}

std::list<Punto> miNube, nubeUn;
std::list<Punto>::iterator punIt;

miObj = objs.top();
objs.pop();
miNube = miObj.ObtenerNube();
for (punIt = miNube.begin(); punIt != miNube.end(); punIt++) {
  nubeUn.push_back(*punIt);
}
miObj = objs.top();
objs.pop();
miNube = miObj.ObtenerNube();
for (punIt = miNube.begin(); punIt != miNube.end(); punIt++) {
  if (!miObj.BuscarPunto(*punIt))  nubeUn.push_back(*punIt);
}

miObjU.FijarNombre("Union_" + obj2.ObtenerNombre() + "_" + obj1.ObtenerNombre());
miObjU.FijarNube(nubeUn);
miObjU.ActualizarCentroide();

aux.push(miObjU);
while (!aux.empty()) {
  miObj = aux.top();
  aux.pop();
  objetos.push(miObj);
}

ActualizarVisual(miObjU);
}
```

```
bool Objeto::BuscarPunto (Point p) {
  bool enc = false;
  std::list<Punto>::iterator punIt;
  for (punIt = nube.begin(); punIt != nube.end() && !enc; punIt++) {
    if (*punIt == p)  enc = true;
  }
  return enc;
}

void Objeto::ActualizarCentroide () {
  float miCoor[3], cent[3];
  std::list<Punto>::iterator punIt;

  cent[0] = 0.0;
  cent[1] = 0.0;
  cent[2] = 0.0;

  for (punIt = nube.begin(); punIt != nube.end(); punIt++) {
    miCoor = punIt->ObtenerCoord();
    cent[0] += miCoor[0];
    cent[1] += miCoor[1];
    cent[2] += miCoor[2];
  }
  cent[0] /= nube.size();
  cent[1] /= nube.size();
  cent[2] /= nube.size();

  centroide = cent;
}
```

Pasos a seguir:

- Extraer los objetos de la estructura de la escena, cuidando, si es el caso, de restituirlos adecuadamente al terminar.
- Verificar objeto por objeto hasta encontrar aquellos que coincidan con los nombres dados (o las características dadas).
- Al encontrar los objetos, ir pasando cada uno de los puntos al nuevo objeto, primero los del objeto más cercano, y luego los del otro objeto para no repetirlos.
- Calcular el centroide del nuevo objeto y asignarlo.
- Actualizar su indicador de visualización utilizando la función ya desarrollada.
- Agregar el nuevo objeto en el fondo de la estructura, para asegurar que sea el último en dibujarse en la escena.