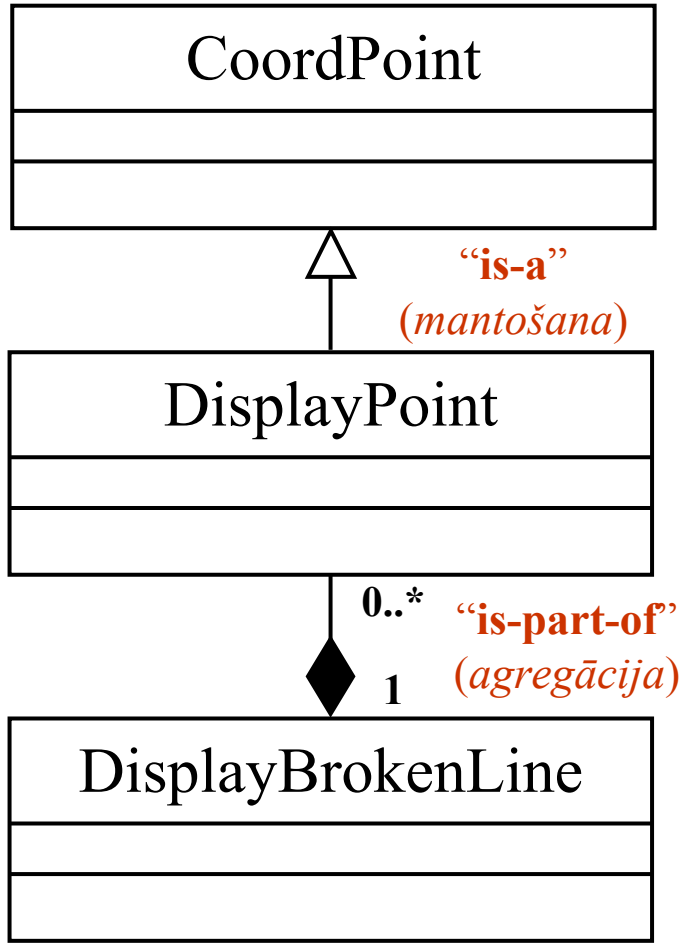
Agregācija. Statiskie locekļi. Izņēmumu apstrāde

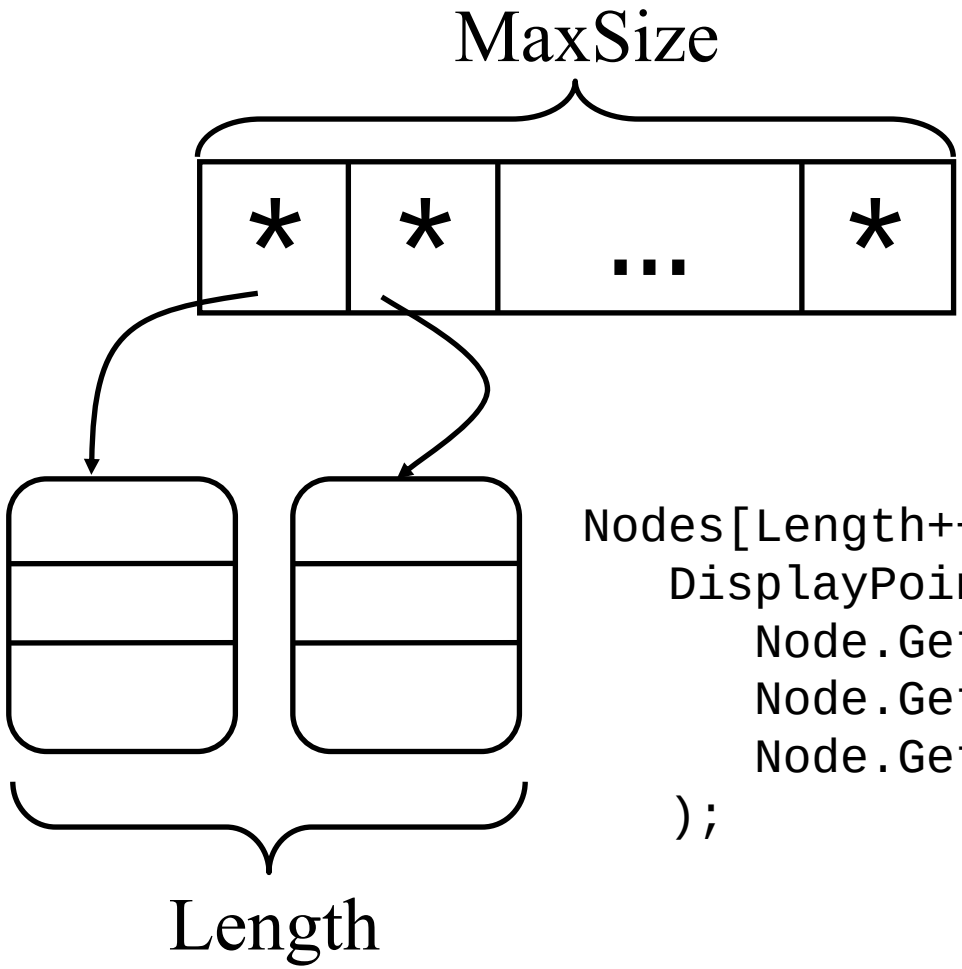
Lauzta līnija uz monitora ekrāna



Atribūtu sekcija klasē *DisplayBrokenLine*

```
class DisplayBrokenLine {
   private:
       // rādītājs uz rādītājiem
       typedef DisplayPoint* DPPointer;
       DPPointer *Nodes;

       // maksimālais mezglu daudzums "pēc noklusēšanas"  visās  lauztās līnijās
       static const unsigned int DEF_MAX_SIZE;

       // maksimālais mezglu daudzums  konkrētā  lauztā līnijā
       unsigned int MaxSize;

       // pašreizējais mezglu daudzums lauztā līnijā
       unsigned int Length;

       // līnijas nogriežņu krāsa
       unsigned int LineColor;
  public:
       ...
```

Atmiņas izdalīšana




```
Nodes =
new DPPointer[MaxSize];
```

```
Nodes[Length++] = new
   DisplayPoint(
      Node.GetX(),
      Node.GetY(),
      Node.GetColor()
   );
```

Statiskie klases locekļi

```
class DisplayBrokenLine {
   private:
      ...
      static const unsigned int DEF_MAX_SIZE;
   public:
      static unsigned int GetDefaultMaxSize() {
         return DEF_MAX_SIZE;
      }
      ...
};
const unsigned int DisplayBrokenLine::DEF_MAX_SIZE = 5;
```

```
DisplayBrokenLine *Line = new DisplayBrokenLine(3, 1);
...
cout << Line->GetDefaultMaxSize() << ".";
cout << DisplayBrokenLine::GetDefaultMaxSize() << ".";
```

Klases *DisplayBrokenLine* konstruktors "pēc noklusēšanas"

```
DisplayBrokenLine():MaxSize(DEF_MAX_SIZE),
   Length(0), LineColor(0) {
      Nodes = new DPPointer[MaxSize];
}
```

Jauna mezgla pievienošana

```
void DisplayBrokenLine::AddNode(const DisplayPoint& Node) {
   if (Length == MaxSize)
      throw OverflowException();
   else
      Nodes[Length++] = new DisplayPoint(
         Node.GetX(), Node.GetY(), Node.GetColor()
      );
}
```

`DisplayPoint&` - *norāde* uz objektu (netiks izveidota *parametra kopija*)

`throw OverflowException()` - *izņēmuma* ierosināšana

Izņēmumu klase *OverflowException()*

```
class OverflowException {
   public:
      OverflowException(){
         cout << endl << "Exception created!" << endl;
      }
      OverflowException(OverflowException&){
         cout << "Exception copied!" << endl;
      }
      ~OverflowException(){
         cout << "Exception finished!" << endl;
      }
};
```

C++ rezervētie vārdi izņēmumu apstrādei

try – *kontrolējamais bloks*

catch - *izņēmuma apstrādātājs*

throw - *izņēmuma ierosināšana*

Izņēmumu apstrāde programmā

```
try {
   Line->AddNode(D2);
   cout << "\nNew Node added successfully!" << endl;
}
   catch (OverflowException&) {
      cout << "Error: maximal size exceeded !" << endl;
   }
   catch (...) {
      cout << "Unknown Error !" << endl;
   }
```

Mezglu iznīcināšana destruktorā

```
DisplayBrokenLine::~DisplayBrokenLine() {
   for(int i=0; i<Length; i++)
      delete Nodes[i];
   delete [] Nodes;
}
```

Iegultās funkcijas ar parametriem-objektiem

```
class Info {
   private:
      string Key;
      …
   public:
      …
      void SetKey(string Key) { // netiks iegulta
         this->Key = Key;
      }
};
```

Rezultāts: brīdinājums `Functions taking class-by-value argument(s) are not expanded inline in function Info::SetKey(string)`

Iegultā funkcija: parametru nodošana *pēc norādes*

```
void SetKey(const string& Key) {
   this->Key = Key;
}
```