11/20/12

d-Pointer

Jonas Cordes

Division of Medical and Biological Informatics (MBI)
German Cancer Research Center Heidelberg

GERMAN
CANCER RESEARCH CENTER
IN THE HELMHOLTZ ASSOCIATION

Design Pattern: The d-pointer

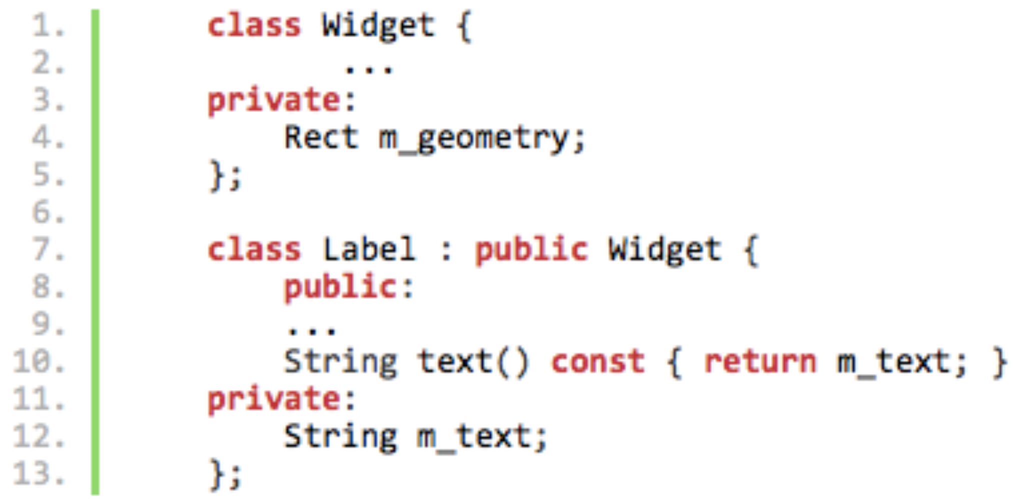
- **hide implementation details** for the user

- Changes can be made to a library without breaking **binary compatibility**

Binary compatibility, what is that?

Example:
your application *XYApp* is based on *WidgetLib* 1.0

WidgetLib
Version 1.0



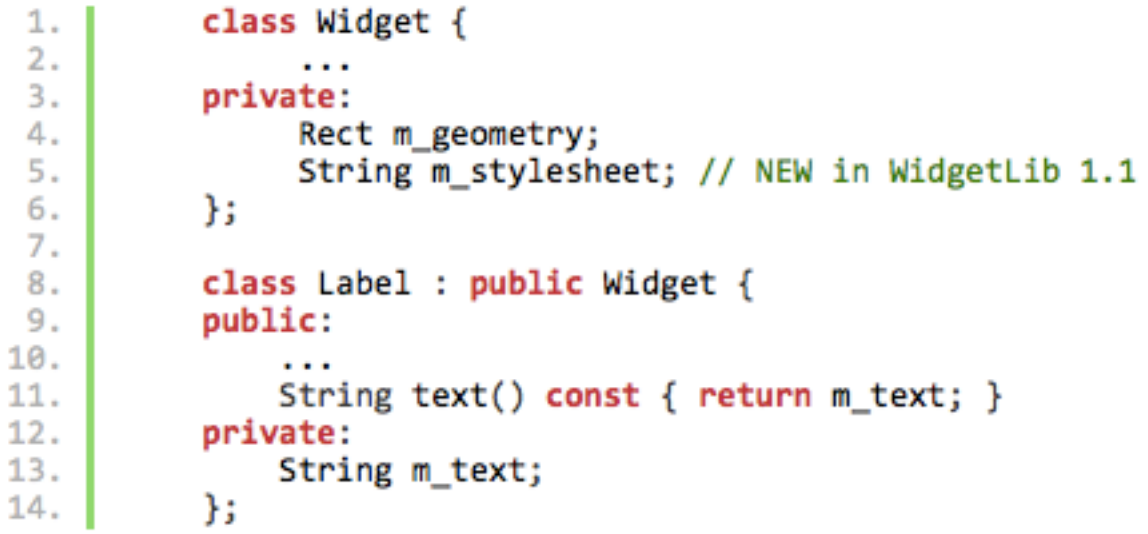
```
class Widget {
       ...
private:
    Rect m_geometry;
};

class Label : public Widget {
    public:
    ...
    String text() const { return m_text; }
private:
    String m_text;
};
```

The Application was compiled with *WidgetLib* 1.0

Binary compatibility, what is that? II

Oh! A new *WidgetLib* Version!

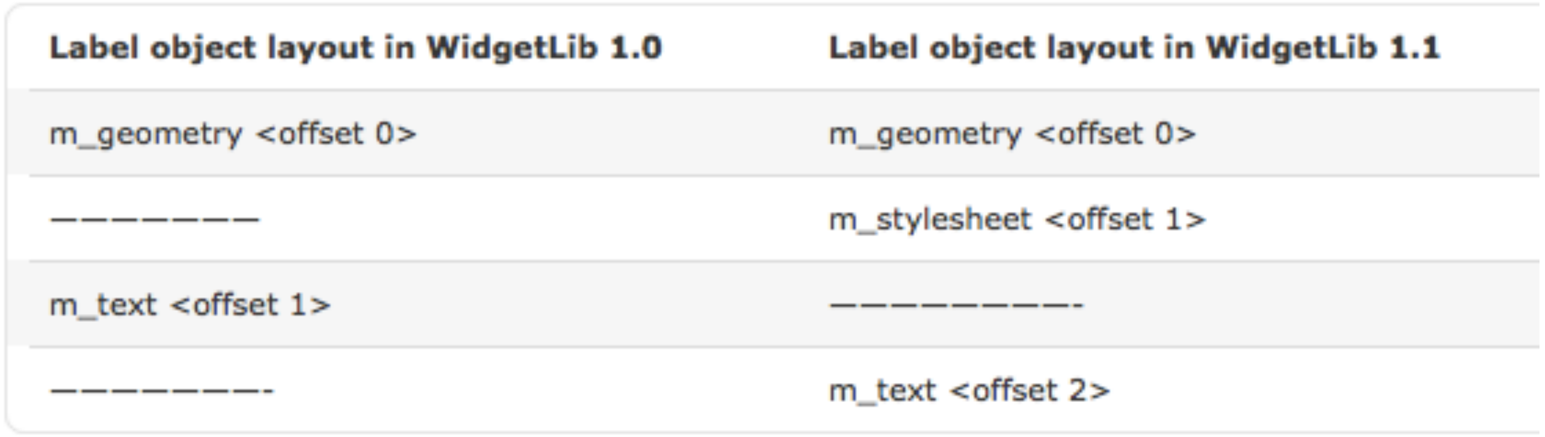
WidgetLib
Version 1.1



```
class Widget {
      ...
private:
      Rect m_geometry;
      String m_stylesheet; // NEW in WidgetLib 1.1
};

class Label : public Widget {
public:
    ...
    String text() const { return m_text; }
private:
    String m_text;
};
```

XYApp that was compiled and ran just fine with *WidgetLib* 1.0 crashes!

Why did it crash?



Label object layout in WidgetLib 1.0	Label object layout in WidgetLib 1.1
m_geometry <offset 0>	m_geometry <offset 0>
--------	m_stylesheet <offset 1>
m_text <offset 1>	----------
----------	m_text <offset 2>

adding new data member
→ changing the size of the objects

A library is **binary compatible**, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile

d-Pointer - Declaration

widget.h

```
class WidgetPrivate;

class Widget {
   ...
   Rect geometry() const;
   ...
private:
   WidgetPrivate *d_ptr;
};
```

widget_p.h, which is the private header file of the widget class

```
/* widget_p.h (_p means private) */
struct WidgetPrivate {
    Rect geometry;
    String stylesheet;
};
```

widget.cpp

```
// With this #include, we can access WidgetPrivate.
#include "widget_p.h"
Widget::Widget()
    : d_ptr(new WidgetPrivate) {
    // Creation of private data
}
```

d-Pointer – inheritance

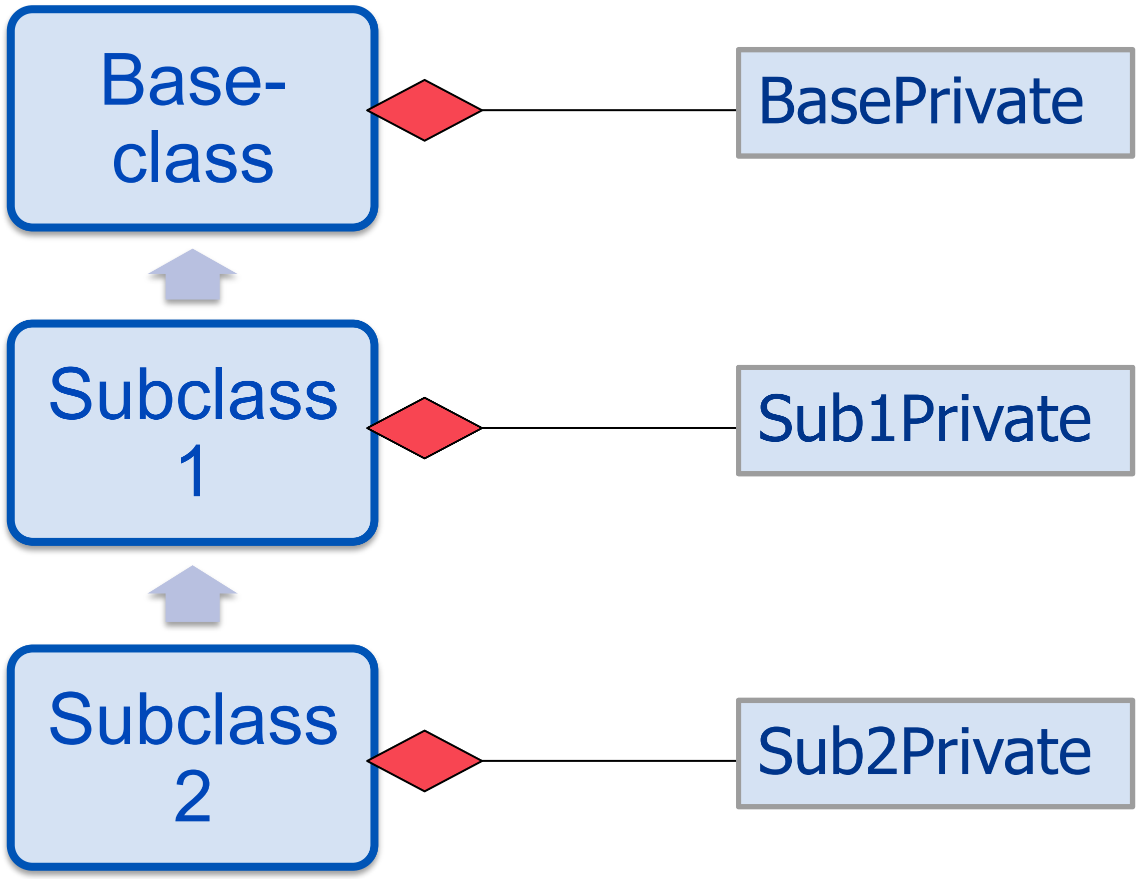
label.h

```cpp
class Label : public Widget {
   ...
   String text();
private:
   // Each class maintains its own d-pointer
   LabelPrivate *d_ptr;
};
```

label.cpp

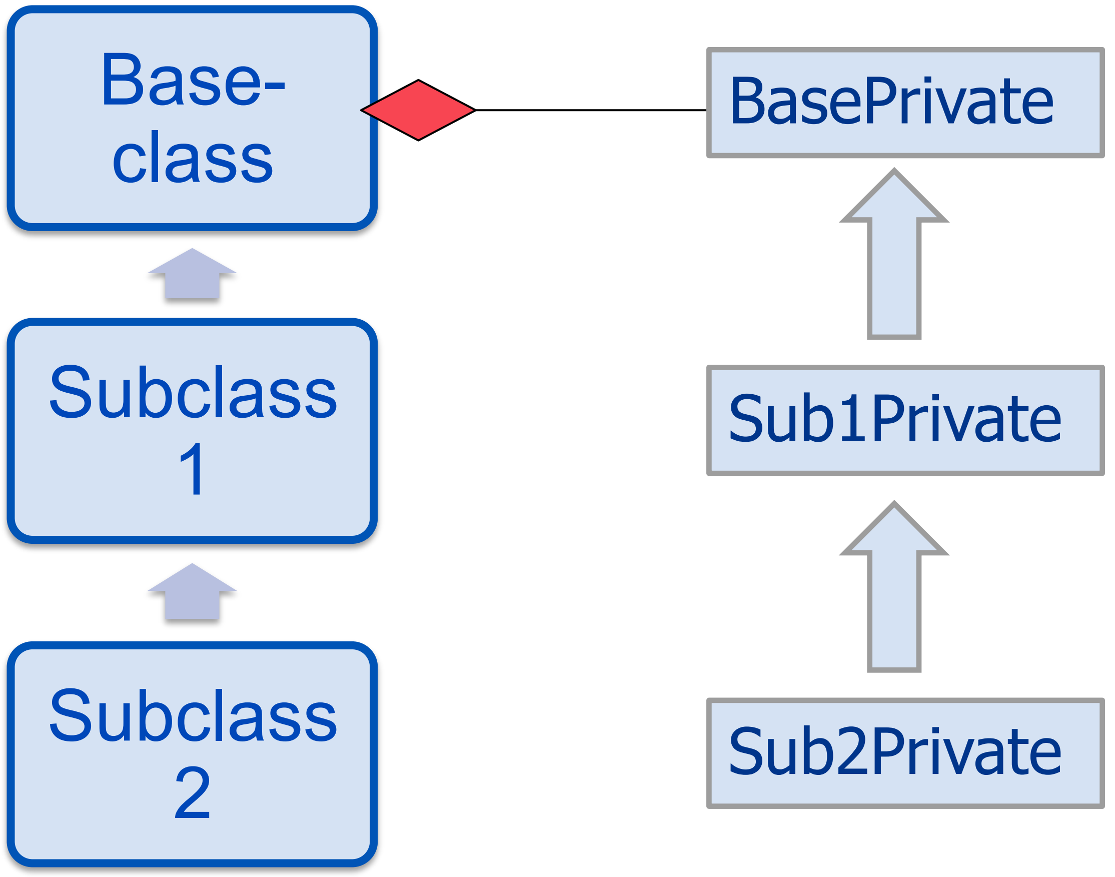
```cpp
// Unlike WidgetPrivate, the author decided LabelPrivate to be defined in the source file itself
struct LabelPrivate {
    String text;
};

Label::Label()
    : d_ptr(new LabelPrivate) {
}

String Label::text() {
    return d_ptr->text;
}
```

Inheritance problems



Level 3 deep inheritance
→ 3 memory allocations

Inheritance problems



Extensions to d-Pointer Pattern I
Inheriting d-Pointers

- Inheriting d-Pointers for optimization

```cpp
#include "widget_p.h"

class LabelPrivate : public WidgetPrivate {
public:
    String text;
};

Label::Label()
  : Widget(*new LabelPrivate) // initialize the d-pointer with our own Private
}

Label::Label(LabelPrivate &d)
  : Widget(d) {
}
```

- Won't work (d-Pointer is of type WidgetPrivate)

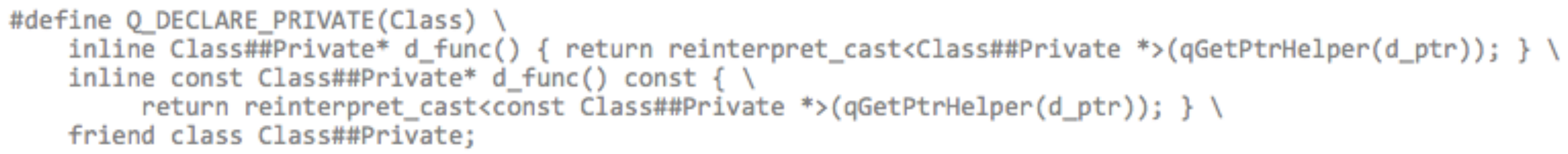
```cpp
void Label::setText(const String &text) {
    d_ptr->text = text;
}
```

- static_cast is necessary

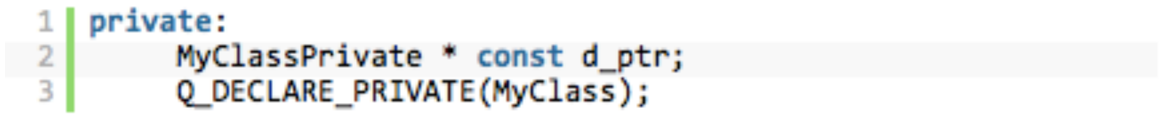
```cpp
void Label::setText(const String &text) {
    LabelPrivate *d = static_cast<LabelPrivate *>(d_ptr); // cast to our private type
    d->text = text;
}
```

d-Pointer in QT

- Create d-Pointer caster-functions

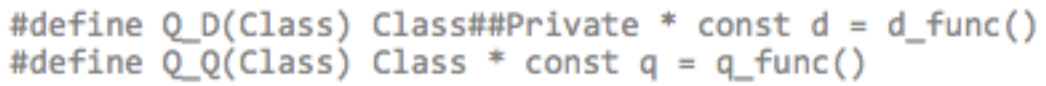


```
#define Q_DECLARE_PRIVATE(Class) \
    inline Class##Private* d_func() { return reinterpret_cast<Class##Private *>(qGetPtrHelper(d_ptr)); } \
    inline const Class##Private* d_func() const { \
         return reinterpret_cast<const Class##Private *>(qGetPtrHelper(d_ptr)); } \
    friend class Class##Private;
```



```
private:
     MyClassPrivate * const d_ptr;
     Q_DECLARE_PRIVATE(MyClass);
```

- Analogy: q-Pointer macro
- Auto-Access to casted d-Pointer or q-Pointer



```
#define Q_D(Class) Class##Private * const d = d_func()
#define Q_Q(Class) Class * const q = q_func()
```

Extensions to d-Pointer Pattern II
q-Pointer

- Implement **q-Pointer** (private-object gets access to the public object)

widget_p.h

```
struct WidgetPrivate {
    // Constructor that initializes the q-ptr
    WidgetPrivate(Widget *q) : q_ptr(q) { }
    Widget *q_ptr; // q-ptr points to the API class
    Rect geometry;
    String stylesheet;
};
```

widget.cpp

```
#include "widget_p.h"
// Create private data.
// Pass the 'this' pointer to initialize the q-ptr
Widget::Widget()
    : d_ptr(new WidgetPrivate(this)) {
}

Rect Widget::geometry() const {
    // the d-ptr is only accessed in the library code
    return d_ptr->geometry;
}
```

Benefits of d-Pointer

- Header file is clean of implementation details (can serve as the API reference)
- Forward-declarations speed up compiling process
- Binary compatibility

dkfz.

Questions?

Referneces

- http://qt-project.org/wiki/Dpointer
- http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++
- http://zchydem.enume.net/2010/01/19/qt-howto-private-classes-and-d-pointers/