Using types for profit and fun

Andreas Rein

January 24, 2016

This talk shall give a few examples on how to solve tricky problems using the (actually not so) dark powers lying within the C++ type system. You might find it especially beneficial if you have problems with a library or class hierarchy you cannot change, if you have a group of functions that all have similar overloads or if you have a general interest in templates.
I do a lot of Qt programming and as much as I cherish the library there are a few examples of, in my opinion, really poor design decisions that can make certain things unnecessarily hard to achieve. In theory, I could submit patches for those issues, but in reality most of them will never be incorporated because they might break a lot of code. Even though the following examples are all derived from solutions for some of these Qt specific issues, their goal is to serve as inspirations for writing less and better code in general.

1 Example 1

For the first example, let's assume that we have an application with a lot of item views (tables, trees) and want to make sure that they all have a consistent appearance and behaviour throughout the entire application. Doing this by hand in the ui file(s) or in code is cumbersome and error-prone. Thus we want to have a function that does this for us:

```
void polishView( QAbstractItemView* view )
{
  //Do some stuff like setting
  //alternatingRowColors, selectionBehavior
  //or selectionMode, etc.
}
```

1.1 The issue

Now, what if we want to set some attributes of the horizontal header as well?

```
void polishView( QAbstractItemView* view )
{
  //What if we want to access the horizontal
  //header to set stretchLastSection or
```

```
  //showSortIndicator?
}
```

QAbstractItemView, the base class of all item views, does not provide any method for accessing any header. *QTreeView* and *QTableView* both have horizontal headers and *QTableView* also has a vertical header. The access method in *QTableView* is *horizontalHeader*, while it is just *header* in *QTreeView*.

1.2 Possible solutions

1.2.1 Casting

The most obvious solution for this problem is to retrieve the header by casting the view to the appropriate type and calling the right method:

```
void polishView( QAbstractItemView* view )
{
  QHeaderView* header = nullptr;

  if( dynamic_cast< QTableView* >( view ) != nullptr )
  {
    header =
      static_cast< QTableView* >( view )->horizontalHeader();
  }
  else if( dynamic_cast< QTreeView* >( view ) != nullptr )
  {
    header = static_cast< QTreeView* >( view )->header();
  }
  else
  {
    Q_ASSERT_X( false, "polishView",
                "Unsupported view encountered" );
  }
}
```

Hopefully I do not have to explain why this is extremely ugly:

- Unsupported types won't be detected until runtime
- Small runtime performance impact
- To support custom types, we would have to change the library

1.3 Overloading

Another possible solution would be to overload the function:

```
void polishView( QTreeView* view )
{ polishViewImpl( view, view->header() ); }

void polishView( QTableView* view )
{ polishViewImpl( view, view->horizontalHeader() ); }

void polishViewImpl( QAbstractItemView* view,
                     QHeaderView* header )
{ /*do stuff...*/ }
```

Again, there are some major disadvantages:

- Unfeasable if you have a lot of functions that would need overloading
- Even worse when it comes to supporting custom types

1.4 A type safe solution

Luckily, there is a type safe solution to this problem:

```
struct ViewAdapter
{
  QAbstractItemView* view;
  QHeaderView* header;

  template< typename V >
  ViewAdapter( V* v );
};

template<>
ViewAdapter::ViewAdapter< QTreeView >( QTreeView* tree )
  : view( tree ), header( tree->header() ) {}

template<>
ViewAdapter::ViewAdapter< QTableView >( QTableView* tab )
  : view( tab ), header( tab->horizontalHeader() ) {}
```

How to use it:

```
void polishView( const ViewAdapter& viewAdapter )
{
  auto view = viewAdapter.view;
  auto header = viewAdapter.header;

  //Do stuff with view and header...
}

//...

polishView( ui->treeView );
polishView( ui->tableView );
```

It doesn't really get any better than this.

- Unsupported types will be reported at compile time.
- Virtually no runtime overhead.
- Easy to support custom types. Just add a constructor overload.

2 Example 1.5

A similar technique is useful in certain scenarios to reduce the number of function overloads:

```
void setColumnHidden( const ViewAdapter& viewAdapter,
                      int column )
{
  viewAdapter.header->setSectionHidden( column, true );
}
```

```
//Overload so we do not have to retrieve
//the index of the column ourselves every time
void setColumnHidden( const ViewAdapter& viewAdapter,
                      QSqlQueryModel* viewModel,
                      const QString& columnName )
{
  setColumnHidden( viewAdapter,
                   viewModel->record().indexOf( columnName ) );
}
```

Again, we can use a type to solve this:

```
struct ColumnIndex
{
  int col;

  ColumnIndex( int c ) : col( c ){}

  ColumnIndex( QSqlQueryModel* model,
               const QString& columnName )
  { col = model->record().indexOf( columnName ); }
};

void setColumnHidden( const ViewAdapter& viewAdapter,
                      const ColumnIndex& column )
{ viewAdapter.header->setSectionHidden( column.col, true ); }

//Usage
setColumnHidden( ui->view, 42 );
setColumnHidden( ui->view, { sqlModel, "column_name" } );
```

3 Example 2

```
void onButtonClicked()
{
  //Some blocking operation that is usually pretty fast.
  someOperation();
}
```

3.1 The issue

```
void onButtonClicked()
{
  //What if the operation must not
  //be run twice back-to-back
  //and the user clicks the button
  //while the operation is running?
  someOperation();
}
```

3.2 Possible solutions

```
void onButtonClicked()
{
  button->setEnabled( false );
  //...some other code
  someOperation();
  //...some other code
```

```
  button->setEnabled( true );
}
```

- What if the operation or some other code throws?
- What if the operation is encapsulated in some complicated if-else-forest and in many years somebady adds a return without resetting the state of the button.
- Etc.

```
void onButtonClicked()
{
  //RAII to the rescue!
  QSignalBlocker blocker(button);

  //...some other code
  someOperation();
  //...some other code
}
```

- Sledge hammer to crack a nut.
- What if the button emit's some other signals, that other components rely on?
- There is no visual feedback for the user that he should not click the button. We would actually like the button to be drawn disabled.

```
struct TmpDisabler
{
  TmpDisabler( QPushButton* button )
    : pb( button ), wasEnabled( button->isEnabled() )
    { pb->setEnabled( false ); }

  ~TmpDisabler(){ pb->setEnabled( wasEnabled ); }
private:
  QPushButton* pb; bool wasEnabled;
};

void onButtonClicked()
{
  TmpDisabler tmp(button);

  //...some other code
  someOperation();
  //...some other code
}
```

- ...what about QToolButton?
  No problem! We will just use the common base class QAbstractButton in TmpDisabler instead!
- ...what if I don't have a button but a QComboBox?
  No problem! We will just use the common base class QWidget in TmpDisabler instead!

- ...what if I have a QAction?
  Well... QAction has a setEnabled method but does not derive from QWidget... so...problem?

```
struct TmpDisabler
{
  template< typename T >
  TmpDisabler( T* t ) : b( new Impl< T >( t ) ){}
  ~TmpDisabler(){ delete b; }

private:
  struct Base{ virtual ~Base(){} };

  Base* b = nullptr;

  template< typename T > struct Impl : public Base
  {
    Impl( T* tt ) : t( tt ), wasEnabled( tt->isEnabled() ){}
    ~Impl(){ t->setEnabled( wasEnabled ); }

    T* t; bool wasEnabled;
  };
};
```

- Supports any type that has isEnabled/setEnabled getter/setter (including non-Qt).
- The small runtime overhead usually doesn't matter when it's within a user triggered action.
- Can be easily extended (compared to the bad and ugly solution) to provide more eye candy like setting/resetting Qt::WaitCursor.
- Watch out in threaded environments! (See Qt::QueuedConnection for more info)

4 Example 3

Let's assume your application shall provide a read-only state. For example because an underpriviliged user shall be able to see data but not to modify it, or because the user has to explicitly check out or lock a dataset before he is allowed to modify it. If you have a GUI with a lot of data visualising components like views or widgets it can be a lot of work to make sure to handle all of them correctly.
TODO