Quantum GIS

Coding and Compilation Guide

Version 1.1 *'Pan'*

Preamble

This document is the original Coding and Compilation Guide of the described software Quantum GIS. The software and hardware described in this document are in most cases registered trademarks and are therefore subject to the legal requirements. Quantum GIS is subject to the GNU General Public License. Find more information on the Quantum GIS Homepage http://qgis.osgeo.org.

The details, data, results etc. in this document have been written and verified to the best of knowledge and responsibility of the authors and editors. Nevertheless, mistakes concerning the content are possible.

Therefore, all data are not liable to any duties or guarantees. The authors, editors and publishers do not take any responsibility or liability for failures and their consequences. Your are always welcome to indicate possible mistakes.

This document has been typeset with LaTeX. It is available as LaTeX source code via subversion and online as PDF document via http://qgis.osgeo.org/documentation/manuals.html. Translated versions of this document can be downloaded via the documentation area of the QGIS project as well. For more information about contributing to this document and about translating it, please visit: http://www.qgis.org/wiki/index.php/Community_Ressources

Links in this Document

This document contains internal and external links. Clicking on an internal link moves within the document, while clicking on an external link opens an internet address. In PDF form, internal links are shown in blue, while external links are shown in red and are handled by the system browser. In HTML form, the browser displays and handles both identically.

Coding Compilation Guide Authors and Editors:

Tim Sutton	Marco Hugentobler	Gary E. Sherman
Tara Athan	Godofredo Contreras	Werner Macho
Carson J.Q. Farmer	Otto Dassau	Jürgen E. Fischer
Davis Wills	Magnus Homann	Martin Dobias

With thanks to Tisham Dhar for preparing the initial msys (MS Windows) environment documentation, to Tom Elwertowski and William Kyngesburye for help in the MAC OSX Installation Section and to Carlos Dávila. If we have neglected to mention any contributors, please accept our apologies for this oversight.

Contents

1 Introduction

DISCLAIMER: the Coding and Compilation Guide has not yet been updated for consistency with Quantum GIS v1.1. Read at your own risk!

Here we need an introduction :)

2 Writing a QGIS Plugin in C++

In this section we provide a beginner's tutorial for writing a simple QGIS C++ plugin. It is based on a workshop held by Dr. Marco Hugentobler.

QGIS C++ plugins are dynamically linked libraries (.so or .dll). They are linked to QGIS at runtime when requested in the Plugin Manager, and extend the functionality of QGIS via access to the QGIS GUI. In general, they can be devided into core and external plugins.

Technically the QGIS Plugin Manager looks in the lib/qgis directory for all .so files and loads them when it is started. When it is closed they are unloaded again, except the ones enabled by the user (See User Manual). For newly loaded plugins, the *classFactory* method creates an instance of the plugin class and the *initGui* method of the plugin is called to show the GUI elements in the plugin menu and toolbar. The *unload()* function of the plugin is used to remove the allocated GUI elements and the plugin class itself is removed using the class destructor. To list the plugins, each plugin must have a few external 'C' functions for description and of course the *classFactory* method.

2.1 Why C++ and what about licensing

QGIS itself is written in C++, so it makes sense to write plugins in C++ as well. It is an object-oriented programming (OOP) language that is prefered by many developers for creating large-scale applications.

QGIS C++ plugins take advantage of the functionalities provided by the libqgis*.so libraries. As these libraries licensed under the GNU GPL, QGIS C++ plugins must also be licenced under the GPL. This means that you may use your plugins for any purpose and you are not required to publish them. If you do publish them however, they must be published under the conditions of the GPL license.

2.2 Programming a QGIS C++ Plugin in four steps

The C++ plugin example covered in this manual is a point converter plugin and intentionally kept simple. The plugin searches the active vector layer in QGIS, converts all vertices of the layer's features to point features (keeping the attributes), and finally writes the point features to a delimited text file. The new layer can then be loaded into QGIS using the delimited text plugin (see User Manual).

Step 1: Make the plugin manager recognise the plugin

As a first step we create the `QgsPointConverter.h` and `QgsPointConverter.cpp` files. We then add virtual methods inherited from QgisPlugin (but leave them empty for now), create the necessary external 'C' methods, and a .pro file (which is a Qt mechanism to easily create Makefiles). Then we

compile the sources, move the compiled library into the plugin folder, and load it in the QGIS Plugin Manager.

a) Create new pointconverter.pro file and add:

```
#base directory of the qgis installation
QGIS_DIR = /home/marco/src/qgis

TEMPLATE = lib
CONFIG = qt
QT += xml qt3support
unix:LIBS += -L/$$QGIS_DIR/lib -lqgis_core -lqgis_gui
INCLUDEPATH += $$QGIS_DIR/src/ui $$QGIS_DIR/src/plugins  $$QGIS_DIR/src/gui \
       $$QGIS_DIR/src/raster $$QGIS_DIR/src/core $$QGIS_DIR
SOURCES = qgspointconverterplugin.cpp
HEADERS = qgspointconverterplugin.h
DEST = pointconverterplugin.so
DEFINES += GUI_EXPORT= CORE_EXPORT=
```

b) Create new qgspointconverterplugin.h file and add:

```
#ifndef QGSPOINTCONVERTERPLUGIN_H
#define QGSPOINTCONVERTERPLUGIN_H

#include "qgisplugin.h"

/**A plugin that converts vector layers to delimited text point files.
 The vertices of polygon/line type layers are converted to point features*/
class QgsPointConverterPlugin: public QgisPlugin
{
  public:
  QgsPointConverterPlugin(QgisInterface* iface);
  ~QgsPointConverterPlugin();
  void initGui();
  void unload();

  private:
  QgisInterface* mIface;
};
#endif
```

c) Create new qgspointconverterplugin.cpp file and add:

```
#include "qgspointconverterplugin.h"

#ifdef WIN32
#define QGISEXTERN extern "C" __declspec( dllexport )
#else
#define QGISEXTERN extern "C"
#endif

QgsPointConverterPlugin::QgsPointConverterPlugin(QgisInterface* iface): mIface(iface)
{
}

QgsPointConverterPlugin::~QgsPointConverterPlugin()
{
}

void QgsPointConverterPlugin::initGui()
{
}

void QgsPointConverterPlugin::unload()
{
}

QGISEXTERN QgisPlugin* classFactory(QgisInterface* iface)
{
  return new QgsPointConverterPlugin(iface);
}

QGISEXTERN QString name()
{
  return "point converter plugin";
}

QGISEXTERN QString description()
{
  return "A plugin that converts vector layers to delimited text point files";
}

QGISEXTERN QString version()
{
  return "0.00001";
```

```
}

// Return the type (either UI or MapLayer plugin)
QGISEXTERN int type()
{
  return QgisPlugin::UI;
}

// Delete ourself
QGISEXTERN void unload(QgisPlugin* theQgsPointConverterPluginPointer)
{
  delete theQgsPointConverterPluginPointer;
}
```

Step 2: Create an icon, a button and a menu for the plugin

This step includes adding a pointer to the QgisInterface object in the plugin class. Then we create a QAction and a callback function (slot), add it to the QGIS GUI using QgisInterface::addToolBarIcon() and QgisInterface::addPluginToMenu() and finally remove the QAction in the *unload()* method.

d) Open qgspointconverterplugin.h again and extend existing content to:

```
#ifndef QGSPOINTCONVERTERPLUGIN_H
#define QGSPOINTCONVERTERPLUGIN_H

#include "qgisplugin.h"
#include <QObject>

class QAction;

/**A plugin that converts vector layers to delimited text point files.
 The vertices of polygon/line type layers are converted to point features*/
class QgsPointConverterPlugin: public QObject, public QgisPlugin
{
  Q_OBJECT

 public:
  QgsPointConverterPlugin(QgisInterface* iface);
  ~QgsPointConverterPlugin();
  void initGui();
  void unload();
```

```
 private:
  QgisInterface* mIface;
  QAction* mAction;

   private slots:
   void convertToPoint();
};

#endif
```

e) Open qgspointconverterplugin.cpp again and extend existing content to:

```
#include "qgspointconverterplugin.h"
#include "qgisinterface.h"
#include <QAction>

#ifdef WIN32
#define QGISEXTERN extern "C" __declspec( dllexport )
#else
#define QGISEXTERN extern "C"
#endif

QgsPointConverterPlugin::QgsPointConverterPlugin(QgisInterface* iface): \
    mIface(iface), mAction(0)
{

}

QgsPointConverterPlugin::~QgsPointConverterPlugin()
{

}

void QgsPointConverterPlugin::initGui()
{
  mAction = new QAction(tr("&Convert to point"), this);
  connect(mAction, SIGNAL(activated()), this, SLOT(convertToPoint()));
  mIface->addToolBarIcon(mAction);
  mIface->addPluginToMenu(tr("&Convert to point"), mAction);
}
```

```
void QgsPointConverterPlugin::unload()
{
  mIface->removeToolBarIcon(mAction);
  mIface->removePluginMenu(tr("&Convert to point"), mAction);
  delete mAction;
}

void QgsPointConverterPlugin::convertToPoint()
{
  qWarning("in method convertToPoint");
}

QGISEXTERN QgisPlugin* classFactory(QgisInterface* iface)
{
  return new QgsPointConverterPlugin(iface);
}

QGISEXTERN QString name()
{
  return "point converter plugin";
}

QGISEXTERN QString description()
{
  return "A plugin that converts vector layers to delimited text point files";
}

QGISEXTERN QString version()
{
  return "0.00001";
}

// Return the type (either UI or MapLayer plugin)
QGISEXTERN int type()
{
  return QgisPlugin::UI;
}

// Delete ourself
QGISEXTERN void unload(QgisPlugin* theQgsPointConverterPluginPointer)
{
  delete theQgsPointConverterPluginPointer;
```

```
}
```

Step 3: Read point features from the active layer and write to text file

To read the point features from the active layer we need to query the current layer and the location for the new text file. Then we iterate through all features of the current layer, convert the geometries (vertices) to points, open a new file and use QTextStream to write the x- and y-coordinates into it.

f) Open qgspointconverterplugin.h again and extend existing content to

```
class QgsGeometry;
class QTextStream;

private:

void convertPoint(QgsGeometry* geom, const QString& attributeString, \
  QTextStream& stream) const;
void convertMultiPoint(QgsGeometry* geom, const QString& attributeString, \
  QTextStream& stream) const;
void convertLineString(QgsGeometry* geom, const QString& attributeString, \
  QTextStream& stream) const;
void convertMultiLineString(QgsGeometry* geom, const QString& attributeString, \
  QTextStream& stream) const;
void convertPolygon(QgsGeometry* geom, const QString& attributeString, \
  QTextStream& stream) const;
void convertMultiPolygon(QgsGeometry* geom, const QString& attributeString, \
  QTextStream& stream) const;
```

g) Open qgspointconverterplugin.cpp again and extend existing content to:

```
#include "qgsgeometry.h"
#include "qgsvectordataprovider.h"
#include "qgsvectorlayer.h"
#include <QFileDialog>
#include <QMessageBox>
#include <QTextStream>

void QgsPointConverterPlugin::convertToPoint()
{
  qWarning("in method convertToPoint");
  QgsMapLayer* theMapLayer = mIface->activeLayer();
```

```
  if(!theMapLayer)
    {
      QMessageBox::information(0, tr("no active layer"), \
      tr("this plugin needs an active point vector layer to make conversions \
          to points"), QMessageBox::Ok);
      return;
    }
  QgsVectorLayer* theVectorLayer = dynamic_cast<QgsVectorLayer*>(theMapLayer);
  if(!theVectorLayer)
    {
      QMessageBox::information(0, tr("no vector layer"), \
      tr("this plugin needs an active point vector layer to make conversions \
          to points"), QMessageBox::Ok);
      return;
    }

  QString fileName = QFileDialog::getSaveFileName();
  if(!fileName.isNull())
    {
      qWarning("The selected filename is: " + fileName);
      QFile f(fileName);
      if(!f.open(QIODevice::WriteOnly))
      {
QMessageBox::information(0, "error", "Could not open file", QMessageBox::Ok);
return;
      }
      QTextStream theTextStream(&f);
      theTextStream.setRealNumberNotation(QTextStream::FixedNotation);

      QgsFeature currentFeature;
      QgsGeometry* currentGeometry = 0;

      QgsVectorDataProvider* provider = theVectorLayer->dataProvider();
      if(!provider)
      {
          return;
      }

      theVectorLayer->select(provider->attributeIndexes(), \
      theVectorLayer->extent(), true, false);

      //write header
```

```
      theTextStream << "x,y";
      theTextStream << endl;

      while(theVectorLayer->nextFeature(currentFeature))
      {
 QString featureAttributesString;

        currentGeometry = currentFeature.geometry();
        if(!currentGeometry)
        {
            continue;
        }

        switch(currentGeometry->wkbType())
        {
            case QGis::WKBPoint:
            case QGis::WKBPoint25D:
                convertPoint(currentGeometry, featureAttributesString, \
theTextStream);
                break;

            case QGis::WKBMultiPoint:
            case QGis::WKBMultiPoint25D:
                convertMultiPoint(currentGeometry, featureAttributesString, \
theTextStream);
                break;

            case QGis::WKBLineString:
            case QGis::WKBLineString25D:
                convertLineString(currentGeometry, featureAttributesString, \
theTextStream);
                break;

            case QGis::WKBMultiLineString:
            case QGis::WKBMultiLineString25D:
                convertMultiLineString(currentGeometry, featureAttributesString \
theTextStream);
                break;

            case QGis::WKBPolygon:
            case QGis::WKBPolygon25D:
                convertPolygon(currentGeometry, featureAttributesString, \
```

```
theTextStream);
                break;

            case QGis::WKBMultiPolygon:
            case QGis::WKBMultiPolygon25D:
                convertMultiPolygon(currentGeometry, featureAttributesString, \
theTextStream);
                break;
        }
      }
    }
}

//geometry converter functions
void QgsPointConverterPlugin::convertPoint(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsPoint p = geom->asPoint();
    stream << p.x() << "," << p.y();
    stream << endl;
}

void QgsPointConverterPlugin::convertMultiPoint(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsMultiPoint mp = geom->asMultiPoint();
    QgsMultiPoint::const_iterator it = mp.constBegin();
    for(; it != mp.constEnd(); ++it)
    {
        stream << (*it).x() << "," << (*it).y();
        stream << endl;
    }
}

void QgsPointConverterPlugin::convertLineString(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsPolyline line = geom->asPolyline();
    QgsPolyline::const_iterator it = line.constBegin();
    for(; it != line.constEnd(); ++it)
    {
        stream << (*it).x() << "," << (*it).y();
```

```
        stream << endl;
    }
}

void QgsPointConverterPlugin::convertMultiLineString(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsMultiPolyline ml = geom->asMultiPolyline();
    QgsMultiPolyline::const_iterator lineIt = ml.constBegin();
    for(; lineIt != ml.constEnd(); ++lineIt)
    {
        QgsPolyline currentPolyline = *lineIt;
        QgsPolyline::const_iterator vertexIt = currentPolyline.constBegin();
        for(; vertexIt != currentPolyline.constEnd(); ++vertexIt)
        {
            stream << (*vertexIt).x() << "," << (*vertexIt).y();
            stream << endl;
        }
    }
}

void QgsPointConverterPlugin::convertPolygon(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsPolygon polygon = geom->asPolygon();
    QgsPolygon::const_iterator it = polygon.constBegin();
    for(; it != polygon.constEnd(); ++it)
    {
        QgsPolyline currentRing = *it;
        QgsPolyline::const_iterator vertexIt = currentRing.constBegin();
        for(; vertexIt != currentRing.constEnd(); ++vertexIt)
        {
            stream << (*vertexIt).x() << "," << (*vertexIt).y();
            stream << endl;
        }
    }
}

void QgsPointConverterPlugin::convertMultiPolygon(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsMultiPolygon mp = geom->asMultiPolygon();
```

```
    QgsMultiPolygon::const_iterator polyIt = mp.constBegin();
    for(; polyIt != mp.constEnd(); ++polyIt)
    {
        QgsPolygon currentPolygon = *polyIt;
        QgsPolygon::const_iterator ringIt = currentPolygon.constBegin();
        for(; ringIt != currentPolygon.constEnd(); ++ringIt)
        {
            QgsPolyline currentPolyline = *ringIt;
            QgsPolyline::const_iterator vertexIt = currentPolyline.constBegin();
            for(; vertexIt != currentPolyline.constEnd(); ++vertexIt)
            {
                stream << (*vertexIt).x() << "," << (*vertexIt).y();
                stream << endl;
            }
        }
    }
}
```

Step 4: Copy the feature attributes to the text file

At the end we extract the attributes from the active layer using QgsVectorDataProvider::fieldNameMap(). For each feature we extract the field values using QgsFeature::attributeMap() and add the contents (comma separated) behind the x- and y-coordinates for each new point feature. For this step there is no need for any furter change in `qgspointconverterplugin.h`

h) Open qgspointconverterplugin.cpp again and extend existing content to:

```
#include "qgspointconverterplugin.h"
#include "qgisinterface.h"
#include "qgsgeometry.h"
#include "qgsvectordataprovider.h"
#include "qgsvectorlayer.h"
#include <QAction>
#include <QFileDialog>
#include <QMessageBox>
#include <QTextStream>

#ifdef WIN32
#define QGISEXTERN extern "C" __declspec( dllexport )
#else
#define QGISEXTERN extern "C"
```

```
#endif

QgsPointConverterPlugin::QgsPointConverterPlugin(QgisInterface* iface): \
mIface(iface), mAction(0)
{

}

QgsPointConverterPlugin::~QgsPointConverterPlugin()
{

}

void QgsPointConverterPlugin::initGui()
{
  mAction = new QAction(tr("&Convert to point"), this);
  connect(mAction, SIGNAL(activated()), this, SLOT(convertToPoint()));
  mIface->addToolBarIcon(mAction);
  mIface->addPluginToMenu(tr("&Convert to point"), mAction);
}

void QgsPointConverterPlugin::unload()
{
  mIface->removeToolBarIcon(mAction);
  mIface->removePluginMenu(tr("&Convert to point"), mAction);
  delete mAction;
}

void QgsPointConverterPlugin::convertToPoint()
{
  qWarning("in method convertToPoint");
  QgsMapLayer* theMapLayer = mIface->activeLayer();
  if(!theMapLayer)
    {
      QMessageBox::information(0, tr("no active layer"), \
      tr("this plugin needs an active point vector layer to make conversions \
          to points"), QMessageBox::Ok);
      return;
    }
  QgsVectorLayer* theVectorLayer = dynamic_cast<QgsVectorLayer*>(theMapLayer);
  if(!theVectorLayer)
    {
```

```
      QMessageBox::information(0, tr("no vector layer"), \
      tr("this plugin needs an active point vector layer to make conversions \
          to points"), QMessageBox::Ok);
      return;
    }

  QString fileName = QFileDialog::getSaveFileName();
  if(!fileName.isNull())
    {
      qWarning("The selected filename is: " + fileName);
      QFile f(fileName);
      if(!f.open(QIODevice::WriteOnly))
      {
QMessageBox::information(0, "error", "Could not open file", QMessageBox::Ok);
return;
      }
      QTextStream theTextStream(&f);
      theTextStream.setRealNumberNotation(QTextStream::FixedNotation);

      QgsFeature currentFeature;
      QgsGeometry* currentGeometry = 0;

      QgsVectorDataProvider* provider = theVectorLayer->dataProvider();
      if(!provider)
      {
          return;
      }

      theVectorLayer->select(provider->attributeIndexes(), \
      theVectorLayer->extent(), true, false);

      //write header
      theTextStream << "x,y";
      QMap<QString, int> fieldMap = provider->fieldNameMap();
      //We need the attributes sorted by index.
      //Therefore we insert them in a second map where key / values are exchanged
      QMap<int, QString> sortedFieldMap;
      QMap<QString, int>::const_iterator fieldIt = fieldMap.constBegin();
      for(; fieldIt != fieldMap.constEnd(); ++fieldIt)
      {
        sortedFieldMap.insert(fieldIt.value(), fieldIt.key());
      }
```

```
      QMap<int, QString>::const_iterator sortedFieldIt = sortedFieldMap.constBegin();
      for(; sortedFieldIt != sortedFieldMap.constEnd(); ++sortedFieldIt)
      {
          theTextStream << "," << sortedFieldIt.value();
      }

      theTextStream << endl;

      while(theVectorLayer->nextFeature(currentFeature))
      {
        QString featureAttributesString;
         const QgsAttributeMap& map = currentFeature.attributeMap();
         QgsAttributeMap::const_iterator attributeIt = map.constBegin();
         for(; attributeIt != map.constEnd(); ++attributeIt)
         {
            featureAttributesString.append(",");
            featureAttributesString.append(attributeIt.value().toString());
         }


        currentGeometry = currentFeature.geometry();
        if(!currentGeometry)
        {
            continue;
        }

        switch(currentGeometry->wkbType())
        {
            case QGis::WKBPoint:
            case QGis::WKBPoint25D:
                convertPoint(currentGeometry, featureAttributesString, \
theTextStream);
                break;

            case QGis::WKBMultiPoint:
            case QGis::WKBMultiPoint25D:
                convertMultiPoint(currentGeometry, featureAttributesString, \
theTextStream);
                break;

            case QGis::WKBLineString:
```

```
            case QGis::WKBLineString25D:
                convertLineString(currentGeometry, featureAttributesString, \
theTextStream);
                break;

            case QGis::WKBMultiLineString:
            case QGis::WKBMultiLineString25D:
                convertMultiLineString(currentGeometry, featureAttributesString \
theTextStream);
                break;

            case QGis::WKBPolygon:
            case QGis::WKBPolygon25D:
                convertPolygon(currentGeometry, featureAttributesString, \
theTextStream);
                break;

            case QGis::WKBMultiPolygon:
            case QGis::WKBMultiPolygon25D:
                convertMultiPolygon(currentGeometry, featureAttributesString, \
theTextStream);
                break;
        }
      }
    }
}

//geometry converter functions
void QgsPointConverterPlugin::convertPoint(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsPoint p = geom->asPoint();
    stream << p.x() << "," << p.y();
    stream << attributeString;
    stream << endl;
}

void QgsPointConverterPlugin::convertMultiPoint(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsMultiPoint mp = geom->asMultiPoint();
    QgsMultiPoint::const_iterator it = mp.constBegin();
```

```
    for(; it != mp.constEnd(); ++it)
    {
        stream << (*it).x() << "," << (*it).y();
        stream << attributeString;
        stream << endl;
    }
}

void QgsPointConverterPlugin::convertLineString(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsPolyline line = geom->asPolyline();
    QgsPolyline::const_iterator it = line.constBegin();
    for(; it != line.constEnd(); ++it)
    {
        stream << (*it).x() << "," << (*it).y();
        stream << attributeString;
        stream << endl;
    }
}

void QgsPointConverterPlugin::convertMultiLineString(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsMultiPolyline ml = geom->asMultiPolyline();
    QgsMultiPolyline::const_iterator lineIt = ml.constBegin();
    for(; lineIt != ml.constEnd(); ++lineIt)
    {
        QgsPolyline currentPolyline = *lineIt;
        QgsPolyline::const_iterator vertexIt = currentPolyline.constBegin();
        for(; vertexIt != currentPolyline.constEnd(); ++vertexIt)
        {
            stream << (*vertexIt).x() << "," << (*vertexIt).y();
            stream << attributeString;
            stream << endl;
        }
    }
}

void QgsPointConverterPlugin::convertPolygon(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
```

```
    QgsPolygon polygon = geom->asPolygon();
    QgsPolygon::const_iterator it = polygon.constBegin();
    for(; it != polygon.constEnd(); ++it)
    {
        QgsPolyline currentRing = *it;
        QgsPolyline::const_iterator vertexIt = currentRing.constBegin();
        for(; vertexIt != currentRing.constEnd(); ++vertexIt)
        {
            stream << (*vertexIt).x() << "," << (*vertexIt).y();
            stream << attributeString;
            stream << endl;
        }
    }
}

void QgsPointConverterPlugin::convertMultiPolygon(QgsGeometry* geom, const QString& \
attributeString, QTextStream& stream) const
{
    QgsMultiPolygon mp = geom->asMultiPolygon();
    QgsMultiPolygon::const_iterator polyIt = mp.constBegin();
    for(; polyIt != mp.constEnd(); ++polyIt)
    {
        QgsPolygon currentPolygon = *polyIt;
        QgsPolygon::const_iterator ringIt = currentPolygon.constBegin();
        for(; ringIt != currentPolygon.constEnd(); ++ringIt)
        {
            QgsPolyline currentPolyline = *ringIt;
            QgsPolyline::const_iterator vertexIt = currentPolyline.constBegin();
            for(; vertexIt != currentPolyline.constEnd(); ++vertexIt)
            {
                stream << (*vertexIt).x() << "," << (*vertexIt).y();
                stream << attributeString;
                stream << endl;
            }
        }
    }
}

QGISEXTERN QgisPlugin* classFactory(QgisInterface* iface)
{
  return new QgsPointConverterPlugin(iface);
}
```

```
QGISEXTERN QString name()
{
  return "point converter plugin";
}

QGISEXTERN QString description()
{
  return "A plugin that converts vector layers to delimited text point files";
}

QGISEXTERN QString version()
{
  return "0.00001";
}

// Return the type (either UI or MapLayer plugin)
QGISEXTERN int type()
{
  return QgisPlugin::UI;
}

// Delete ourself
QGISEXTERN void unload(QgisPlugin* theQgsPointConverterPluginPointer)
{
  delete theQgsPointConverterPluginPointer;
}
```

2.3 Further information

As you can see, you need information from many different sources to write QGIS C++ plugins. Plugin writers need to know C++, the QGIS plugin interface as well as Qt4 classes and tools. At the beginning it is best to learn from examples and copy the mechanism of existing plugins.

There is a a collection of online documentation that may be usefull for QGIS C++ programers:

- QGIS Plugin Debugging: http://wiki.qgis.org/qgiswiki/DebuggingPlugins
- QGIS API Documentation: http://svn.qgis.org/api_doc/html/
- Qt documentation: http://doc.trolltech.com/4.3/index.html

3 Creating C++ Applications

Not everyone wants a full blown GIS desktop application. Sometimes you want to just have a widget inside your application that displays a map while the main goal of the application lies elsewhere. Perhaps a database frontend with a map display? This Section provides two simple code examples by Tim Sutton, based on earlier work by Francis Bolduc. They are available in the qgis subversion repository together with more interesting tutorials. Check out the whole repository from: `https://svn.osgeo.org/qgis/trunk/code_examples/`

3.1 Creating a simple mapping widget

With this tutorial we will create a simple mapping widget. It won't do anything much - just load a shape file and display it in a random colour. This should give you an idea of the potential for using QGIS as an embedded mapping component.

We start by adding the neccessary includes for our app:

```
//
// QGIS Includes
//
#include <qgsapplication.h>
#include <qgsproviderregistry.h>
#include <qgssinglesymbolrenderer.h>
#include <qgsmaplayerregistry.h>
#include <qgsvectorlayer.h>
#include <qgsmapcanvas.h>
//
// Qt Includes
//
#include <QString>
#include <QApplication>
#include <QWidget>
```

We use QgsApplication instead of Qt's QApplication, to take advantage of various static methods that can be used to locate library paths and so on.

The provider registry is a singleton that keeps track of vector data provider plugins. It does all the work for you of loading the plugins and so on. The single symbol renderer is the most basic symbology class. It renders points, lines or polygons in a single colour which is chosen at random by default (though you can set it yourself). Every vector layer must have a symbology associated with it.

The map layer registry keeps track of all the layers you are using. The vector layer class inherits from maplayer and extends it to include specialist functionality for vector data.

Finally, the mapcanvas is our main map area. Its the drawable widget that our map will be dispalyed on.

Now we can move on to initialising our application....

```
int main(int argc, char ** argv)
{
  // Start the Application
  QgsApplication app(argc, argv, true);

  QString myPluginsDir        = "/home/timlinux/apps/lib/qgis";
  QString myLayerPath         = "/home/timlinux/gisdata/brazil/BR_Cidades/";
  QString myLayerBaseName     = "Brasil_Cap";
  QString myProviderName      = "ogr";
```

We now have a qgsapplication and we have defined several variables. Since this tutorial was initially tested on Ubuntu Linux 8.10, we have specified the location of the vector provider plugins as being inside our development install directory. It would probaby make more sense in general to keep the QGIS libs in one of the standard library search paths on your system (e.g. /usr/lib) but this way will do for now.

The next two variables defined here point to the shapefile that is going to be used (though you will likely want to substitute your own data here).

The provider name is important - it tells qgis which data provider to use to load the file. Typically you will use 'ogr' or 'postgres'.

Now we can go on to actually create our layer object.

```
  // Instantiate Provider Registry
  QgsProviderRegistry::instance(myPluginsDir);
```

First we get the provider registry initialised. Its a singleton class so we use the static instance call and pass it the provider lib search path. As it initialises it will scan this path for provider libs.

Now we go on to create a layer...

```
  QgsVectorLayer * mypLayer =
      new QgsVectorLayer(myLayerPath, myLayerBaseName, myProviderName);
```

```
  QgsSingleSymbolRenderer *mypRenderer = new
QgsSingleSymbolRenderer(mypLayer->geometryType());
  QList <QgsMapCanvasLayer> myLayerSet;

  mypLayer->setRenderer(mypRenderer);
  if (mypLayer->isValid())
  {
    qDebug("Layer is valid");
  }
  else
  {
    qDebug("Layer is NOT valid");
  }

  // Add the Vector Layer to the Layer Registry
  QgsMapLayerRegistry::instance()->addMapLayer(mypLayer, TRUE);
  // Add the Layer to the Layer Set
  myLayerSet.append(QgsMapCanvasLayer(mypLayer, TRUE));
```

The code is fairly self explanatory here. We create a layer using the variables we defined earlier. Then we assign the layer a renderer. When we create a renderer, we need to specify the geometry type, which we do by asking the vector layer for its geometry type. Next we add the layer to a layerset (which is used by the QgsMapCanvas to keep track of which layers to render and in what order) and to the maplayer registry. Finally we make sure the layer will be visible.

Now we create a map canvas on to which we can draw the layer.

```
  // Create the Map Canvas
  QgsMapCanvas * mypMapCanvas = new QgsMapCanvas(0, 0);
  mypMapCanvas->setExtent(mypLayer->extent());
  mypMapCanvas->enableAntiAliasing(true);
  mypMapCanvas->setCanvasColor(QColor(255, 255, 255));
  mypMapCanvas->freeze(false);
  // Set the Map Canvas Layer Set
  mypMapCanvas->setLayerSet(myLayerSet);
  mypMapCanvas->setVisible(true);
  mypMapCanvas->refresh();
```

Once again there is nothing particularly tricky here. We create the canvas and then we set its extents to those of our layer. Next we tweak the canvas a bit to draw antialiased vectors. Next we set the

background colour, unfreeze the canvas, make it visible and then refresh it.

```
  // Start the Application Event Loop
  return app.exec();
}
```

In the last step we simply start the Qt event loop and we are done. You can check out, compile and run this example using cmake like this:

```
svn co
https://svn.osgeo.org/qgis/trunk/code_examples/1_hello_world_qgis_style
cd 1_hello_world_qgis_style
mkdir build
#optionally specify where your QGIS is installed (should work on all
platforms)
#if your QGIS is installed to /usr or /usr/local you can leave this next step
out
export LIB_DIR=/home/timlinux/apps
cmake ..
make
./timtut1
```

When we compile and run it here is what the running app looks like:

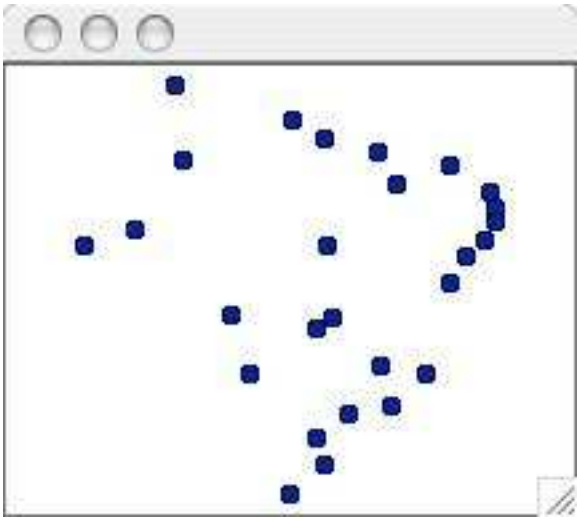
3.2 Working with QgsMapCanvas

In the previous Section (Section 3.1) we showed you how to use the QgsMapCanvas API to create a simple application that loads a shapefile and displays the points in it. But what good is a map that you can't interact with?

In this second tutorial we will extend the previous tutorial by making it a QMainWindow application with a menu, toolbar and canvas area. We show you how to use QgsMapTool - the base class for all tools that are used to interact with the map canvas. The project will provide 4 toolbar icons for

- loading a map layer (layer name is hard coded in the application
- zooming in
- zooming out
- panning

Figure 1: Simple C++ Application



In the working directory for the tutorial code you will find a number of files including c++ sources, icons and a simple data file under data. There is also the .ui file for the main window.

Note: You will need to edit the .pro file in the above svn directory to match your system.

Since much of the code is the same as the previous tutorial, we will focus on the MapTool specifics - the rest of the implementation details can be investigated by checking out the project form SVN. A QgsMapTool is a class that interacts with the MapCanvas using the mouse pointer. QGIS has a number of QgsMapTools implemented, and you can subclass QgsMapTool to create your own. In mainwindow.cpp you will see we have included the headers for the QgsMapTools near the start of the file:

```
//
// QGIS Map tools
//
#include "qgsmaptoolpan.h"
#include "qgsmaptoolzoom.h"
//
// These are the other headers for available map tools
// (not used in this example)
//
//#include "qgsmaptoolcapture.h"
//#include "qgsmaptoolidentify.h"
//#include "qgsmaptoolselect.h"
//#include "qgsmaptoolvertexedit.h"
//#include "qgsmeasure.h"
```

As you can see, I am only using two types of MapTool subclasses for this tutorial, but there are more available in the QGIS library. Hooking up our MapTools to the canvas is very easy using the normal Qt4 signal/slot mechanism:

```
//create the action behaviours
connect(mActionPan, SIGNAL(triggered()), this, SLOT(panMode()));
connect(mActionZoomIn, SIGNAL(triggered()), this, SLOT(zoomInMode()));
connect(mActionZoomOut, SIGNAL(triggered()), this, SLOT(zoomOutMode()));
connect(mActionAddLayer, SIGNAL(triggered()), this, SLOT(addLayer()));
```

Next we make a small toolbar to hold our toolbuttons. Note that the mpAction* actions were created in designer.

```
//create a little toolbar
mpMapToolBar = addToolBar(tr("File"));
mpMapToolBar->addAction(mpActionAddLayer);
mpMapToolBar->addAction(mpActionZoomIn);
mpMapToolBar->addAction(mpActionZoomOut);
mpMapToolBar->addAction(mpActionPan);
```

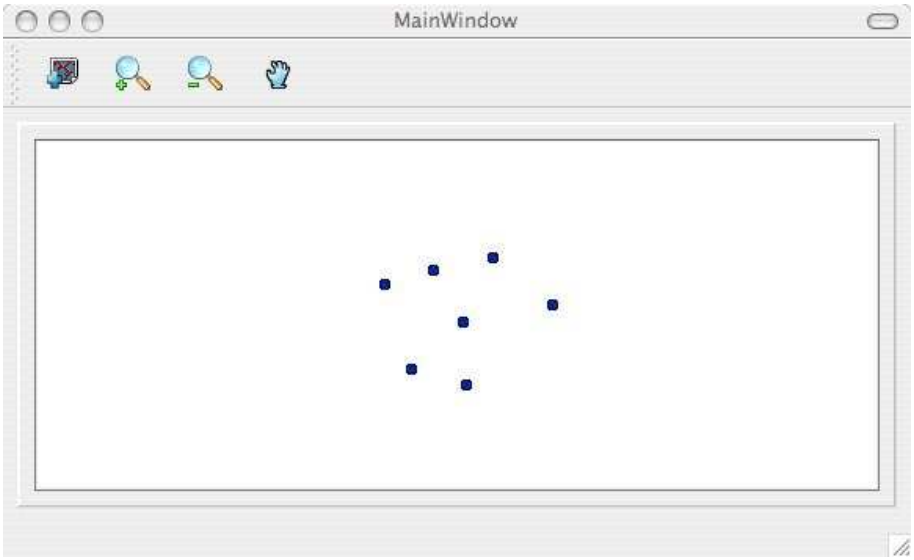
Now we create our three map tools:

```
//create the maptools
mpPanTool = new QgsMapToolPan(mpMapCanvas);
mpPanTool->setAction(mpActionPan);
mpZoomInTool = new QgsMapToolZoom(mpMapCanvas, FALSE); // false = in
mpZoomInTool->setAction(mpActionZoomIn);
mpZoomOutTool = new QgsMapToolZoom(mpMapCanvas, TRUE ); //true = out
mpZoomOutTool->setAction(mpActionZoomOut);
```

Again nothing here is very complicated - we are creating tool instances, each of which is associated with the same mapcanvas, and a different QAction. When the user selects one of the toolbar icons, the active MapTool for the canvas is set. For example when the pan icon is clicked, we do this:

```
void MainWindow::panMode()
{
   mpMapCanvas->setMapTool(mpPanTool);
}
```

Figure 2: QMainWindow application with a menu, toolbar and canvas area



Conclusion

As you can see extending our previous example into something more functional using MapTools is really easy and only requires a few lines of code for each MapTool you want to provide.

You can check out and build this tutorial using SVN and CMake using the following steps:

```
svn co https://svn.osgeo.org/qgis/trunk/code_examples/2_basic_main_window
cd 2_basic_main_window
mkdir build
#optionally specify where your QGIS is installed (should work on all platforms)
#if your QGIS is installed to /usr or /usr/local you can leave this next step out
export LIB_DIR=/home/timlinux/apps
cmake ..
make
./timtut2
```

4 Writing a QGIS Plugin in Python

In this section we provide a beginner's tutorial for writing a simple QGIS Python plugin. It is based on the workshop "Extending the Functionality of QGIS with Python Plugins" held at FOSS4G 2008 by Dr. Marco Hugentobler, Dr. Horst Düster and Tim Sutton.

Apart from writing a QGIS Python plugin, it is also possible to use PyQGIS from a python command line console which is useful for debugging or writing standalone applications in Python, with their own user interfaces based on the functionality of the QGIS core library.

4.1 Why Python and what about licensing

Python is a scripting language that was designed with the goal of being easy to program. It has a mechanism for automatically releasing memory that is no longer used (garbagge collector). A further advantage is that many programs that are written in C++ or Java offer the possibility to write extensions in Python, e.g. OpenOffice or Gimp. Therefore it is a good investment of time to learn the Python language.

PyQGIS plugins take advantage of the functionality of libqgis_core.so and libqgis_gui.so. As both libqgis_core.so and libqgis_gui.so are licensed under GNU GPL, QGIS Python plugins must also be licenced under the GPL. This means you may use your plugins for any purpose, and you are not forced to publish them. If you do publish them however, they must be published under the conditions of the GPL license.

4.2 What needs to be installed to get started

You will need the following libraries and programs to create QGIS python plugins yourself:

- QGIS
- Python >= 2.5
- Qt
- PyQT
- PyQt development tools

If you use Linux, there are binary packages for all major distributions. For Windows, the PyQt installer contains Qt, PyQt and the PyQt development tools.

4.3 Programming a simple PyQGIS Plugin in four steps

The example plugin demonstrated here is intentionally kept simple. It adds a button to the menu bar of QGIS. When the button is clicked, a file dialog appears where the user may load a shape file.

For each python plugin, a dedicated folder that contains the plugin files needs to be created. By default, QGIS looks for plugins in two locations: $QGIS_DIR/share/qgis/python/plugins and $HOME/.qgis/python/plugins. Note that plugins installed in the latter location are only visible for one user.

Step 1: Make the plugin manager recognise the plugin

Each Python plugin is contained in its own directory. When QGIS starts up it will scan each OS specific subdirectory and initialize any plugins it finds.

- Linux and other unices:
  ./share/qgis/python/plugins
  /home/$USERNAME/.qgis/python/plugins
- Mac OS X:
  ./Contents/MacOS/share/qgis/python/plugins
  /Users/$USERNAME/.qgis/python/plugins
- Windows:
  C:\Program Files\QGIS\python\plugins
  C:\Documents and Settings\$USERNAME\.qgis\python\plugins

Once that is done, the plugin will show up in the 

To provide the neccessary information for QGIS, the plugin needs to implement the methods *name()*, *description()*, *version()*, *qgisMinimumVersion()* and *authorName()* which return descriptive strings. The *qgisMinimumVersion()* should return a simple form, for example “1.0“. A plugin also needs a method *classFactory(QgisInterface)* which is called by the plugin manager to create an instance of the plugin. The argument of type QGisInterface is used by the plugin to access functions of the QGIS instance. We are going to work with this object in step 2.

Note that in contrast to other programing languages, indention is very important. The Python interpreter throws an error if it is not correct.

For our plugin we create the plugin folder ’foss4g_plugin’ in `$HOME/.qgis/python/plugins`. Then we add two new textfiles into this folder, `foss4gplugin.py` and `__init__.py`.

The file `foss4gplugin.py` contains the plugin class:

```
# -*- coding: utf-8 -*-
```

```
# Import the PyQt and QGIS libraries
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *
# Initialize Qt resources from file resources.py
import resources

class FOSS4GPlugin:

def __init__(self, iface):
# Save reference to the QGIS interface
  self.iface = iface

def initGui(self):
  print 'Initialising GUI'

def unload(self):
  print 'Unloading plugin'
```

The file `__init__.py` contains the methods *name()*, *description()*, *version()*, *qgisMinimumVersion()* and *authorName()* and *classFactory*. As we are creating a new instance of the plugin class, we need to import the code of this class:

```
# -*- coding: utf-8 -*-
from foss4gplugin import FOSS4GPlugin
def name():
  return "FOSS4G example"
def description():
  return "A simple example plugin to load shapefiles"
def version():
  return "0.1"
def qgisMinimumVersion():
  return "1.0"
def authorName():
  return "John Developer"
def classFactory(iface):
  return FOSS4GPlugin(iface)
```

At this point the plugin already has the neccessary infrastructure to appear in the QGIS 

to be loaded or unloaded.

Step 2: Create an Icon for the plugin

To make the icon graphic available for our program, we need a so-called resource file. In the resource file, the graphic is contained in hexadecimal notation. Fortunately, we don't need to worry about its representation because we use the pyrcc compiler, a tool that reads the file `resources.qrc` and creates a resource file.

The file `foss4g.png` and the `resources.qrc` we use in this workshop can be downloaded from http://karlinapp.ethz.ch/python_foss4g, you can also use your own icon if you prefer, you just need to make sure it is named `foss4g.png`. Move these 2 files into the directory of the example plugin `$HOME/.qgis/python/plugins/foss4g_plugin` and enter: `pyrcc4 -o resources.py resources.qrc`.

Step 3: Add a button and a menu

In this section, we implement the content of the methods *initGui()* and *unload()*. We need an instance of the class **QAction** that executes the *run()* method of the plugin. With the action object, we are then able to generate the menu entry and the button:

```
import resources

  def initGui(self):
    # Create action that will start plugin configuration
    self.action = QAction(QIcon(":/plugins/foss4g_plugin/foss4g.png"), "FOSS4G plugin",
self.iface.getMainWindow())
    # connect the action to the run method
    QObject.connect(self.action, SIGNAL("activated()"), self.run)

    # Add toolbar button and menu item
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginMenu("FOSS-GIS plugin...", self.action)

    def unload(self):
    # Remove the plugin menu item and icon
    self.iface.removePluginMenu("FOSSGIS Plugin...", self.action)
    self.iface.removeToolBarIcon(self.action)
```

Step 4: Load a layer from a shape file

In this step we implement the real functionality of the plugin in the *run()* method. The Qt4 method *QFileDialog::getOpenFileName* opens a file dialog and returns the path to the chosen file. If the user cancels the dialog, the path is a null object, which we test for. We then call the method *addVectorLayer*

of the interface object which loads the layer. The method only needs three arguments: the file path, the name of the layer that will be shown in the legend and the data provider name. For shapefiles, this is 'ogr' because QGIS internally uses the OGR library to access shapefiles:

```
    def run(self):
    fileName = QFileDialog.getOpenFileName(None,QString.fromLocal8Bit("Select a file:"),
"", "*.shp *.gml")
    if fileName.isNull():
      QMessageBox.information(None, "Cancel", "File selection canceled")
      else:
      vlayer = self.iface.addVectorLayer(fileName, "myLayer", "ogr")
```

4.4 Uploading the plugin to the repository

If you have written a plugin you consider to be useful and you want to share with other users you are welcome to upload it to the QGIS User-Contributed Repository.

- Prepare a plugin directory containing only the necessary files (ensure that there is no compiled .pyc files, Subversion .svn directories etc).
- Make a zip archive of it, including the directory. Be sure the zip file name is exactly the same as the directory inside (except the .zip extension of course), if not the Plugin Installer will be unable to relate the available plugin with its locally installed instance.
- Upload it to the repository: http://pyqgis.org/admin/contributed (you will need to register at first time). Please pay attention when filling the form. The Version Number field is especially important, and if filled out incorrectly it may confuse the Plugin Installer and cause false notifications of available updates.

4.5 Further information

As you can see, you need information from many different sources to a write PyQGIS plugin. Plugin authors need to know Python, the QGIS plugin interface, as well as the Qt4 classes and tools. In the beginning, it is best to learn from examples and copy the mechanism of existing plugins. Using the QGIS plugin installer, which itself is a Python plugin, it is possible to download many existing Python plugins and to study their behaviour. It is also possible to use the on-line Python plugin generator to create a base plugin to work off of. This on-line tool will help you to build a minimal plugin that you can use as a starting point in your development. The result is a ready to install QGIS 1.0 plugin that implements an empty dialog with Ok and Close buttons. It is available here: http://www.pyqgis.org/builder/plugin_builder.py

There is a a collection of on-line documentation that may be useful for PyQGIS programmers:

- QGIS wiki: http://wiki.qgis.org/qgiswiki/PythonBindings
- QGIS API documentation: http://doc.qgis.org/index.html
- Qt documentation: http://doc.trolltech.com/4.3/index.html
- PyQt: http://www.riverbankcomputing.co.uk/pyqt/
- Python tutorial: http://docs.python.org/
- A book about desktop GIS and QGIS. It contains a chapter about PyQGIS plugin programing: http://www.pragprog.com/titles/gsdgis/desktop-gis

You can also write plugins for QGIS in C++. See Section 2 for more information about that.

5 Creating PyQGIS Applications

One of the goals of QGIS is to provide not only an application, but a set of libraries that can be used to create new applications. This goal has been realized with the refactoring of libraries that took place after the release of 0.8. Since the release of 0.9, development of standalone applications using either C++ or Python is possible. We recommend you use QGIS 1.0.0 or greater as the basis for your python applications because since this version we now provide a stable consistent API.

In this chapter we'll take a brief look at the process of creating a standalone Python application. The QGIS blog has several examples for creating PyQGIS[1] applications. We'll use one of them as a starting point to get a look at how to create an application.

The features we want in the application are:

- Load a vector layer
- Pan
- Zoom in and out
- Zoom to the full extent of the layer
- Set custom colors when the layer is loaded

This is a pretty minimal feature set. Let's start by designing the GUI using Qt Designer.

5.1 Designing the GUI

Since we are creating a minimalistic application, we'll take the same approach with the GUI. Using Qt Designer, we create a simple MainWindow with no menu or toolbars. This gives us a blank slate to work with. To create the MainWindow:

1. Create a directory for developing the application and change to it
2. Run Qt Designer
3. The New Form dialog should appear. If it doesn't, choose New Form... from the File menu.
4. Choose Main Window from the templates/forms list
5. Click Create
6. Resize the new window to something manageable
7. Find the Frame widget in the list (under Containers) and drag it to the main window you just created
8. Click outside the frame to select the main window area

[1] An application created using Python and the QGIS bindings

9. Click on the Lay Out in a Grid tool. When you do, the frame will expand to fill your entire main window
10. Save the form as `mainwindow.ui`
11. Exit Qt Designer

Now compile the form using the PyQt interface compiler:

```
pyuic4 -o mainwindow_ui.py mainwindow.ui
```

This creates the Python source for the main window GUI. Next we need to create the application code to fill the blank slate with some tools we can use.

5.2 Creating the MainWindow

Now we are ready to write the **MainWindow** class that will do the real work. Since it takes up quite a few lines, we'll look at it in chunks, starting with the import section and environment setup:

```
1 # Loosely based on:
2 #   Original C++ Tutorial 2 by Tim Sutton
3 #   ported to Python by Martin Dobias
4 #   with enhancements by Gary Sherman for FOSS4G2007
5 # Licensed under the terms of GNU GPL 2
6
7 from PyQt4.QtCore import *
8 from PyQt4.QtGui import *
9 from qgis.core import *
10 from qgis.gui import *
11 import sys
12 import os
13 # Import our GUI
14 from mainwindow_ui import Ui_MainWindow
15
16 # Environment variable QGISHOME must be set to the 1.0 install directory
17 # before running this application
18 qgis_prefix = os.getenv("QGISHOME")
```

Some of this should look familiar from our plugin, especially the PyQt4 and QGIS imports. Some specific things to note are the import of our GUI in line 14 and the import of our CORE library on line 9.

Our application needs to know where to find the QGIS installation. Because of this, we set the `QGISHOME` environment variable to point to the install directory of QGIS 1.x In line 20 we store this value from the environment for later use.

Next we need to create our **MainWindow** class which will contain all the logic of our application.

```
21 class MainWindow(QMainWindow, Ui_MainWindow):
22
23   def __init__(self):
24     QMainWindow.__init__(self)
25
26     # Required by Qt4 to initialize the UI
27     self.setupUi(self)
28
29     # Set the title for the app
30     self.setWindowTitle("QGIS Demo App")
31
32     # Create the map canvas
33     self.canvas = QgsMapCanvas()
34     # Set the background color to light blue something
35     self.canvas.setCanvasColor(QColor(200,200,255))
36     self.canvas.enableAntiAliasing(True)
37     self.canvas.useQImageToRender(False)
38     self.canvas.show()
39
40     # Lay our widgets out in the main window using a
41     # vertical box layout
42     self.layout = QVBoxLayout(self.frame)
43     self.layout.addWidget(self.canvas)
44
45     # Create the actions for our tools and connect each to the appropriate
46     # method
47     self.actionAddLayer = QAction(QIcon("(qgis_prefix + "/share/qgis/themes/classic/mActionA
48     \
49         "Add Layer", self.frame)
50     self.connect(self.actionAddLayer, SIGNAL("activated()"), self.addLayer)
51     self.actionZoomIn = QAction(QIcon("(qgis_prefix + "/share/qgis/themes/classic/mActionZoo
52         "Zoom In", self.frame)
53     self.connect(self.actionZoomIn, SIGNAL("activated()"), self.zoomIn)
54     self.actionZoomOut = QAction(QIcon("(qgis_prefix + "/share/qgis/themes/classic/mActionZo
55         "Zoom Out", self.frame)
56     self.connect(self.actionZoomOut, SIGNAL("activated()"), self.zoomOut)
```

```
57      self.actionPan = QAction(QIcon("(qgis_prefix + "/share/qgis/themes/classic/mActionPan.p
58          "Pan", self.frame)
59      self.connect(self.actionPan, SIGNAL("activated()"), self.pan)
60      self.actionZoomFull = QAction(QIcon("(qgis_prefix + "/share/qgis/themes/classic/mAction
61          "Zoom Full Extent", self.frame)
62      self.connect(self.actionZoomFull, SIGNAL("activated()"),
63      self.zoomFull)
64
65      # Create a toolbar
66      self.toolbar = self.addToolBar("Map")
67      # Add the actions to the toolbar
68      self.toolbar.addAction(self.actionAddLayer)
69      self.toolbar.addAction(self.actionZoomIn)
70      self.toolbar.addAction(self.actionZoomOut);
71      self.toolbar.addAction(self.actionPan);
72      self.toolbar.addAction(self.actionZoomFull);
73
74      # Create the map tools
75      self.toolPan = QgsMapToolPan(self.canvas)
76      self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
77      self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
```

Lines 21 through 27 are the basic declaration and initialization of the **MainWindow** and the set up of the user interface using the *setupUi* method. This is required for all applications.

Next we set the title for the application so it says something more interesting than `MainWindow` (line 30). Once that is complete, we are ready to complete the user interface. When we created it in Designer, we left it very sparse—just a main window and a frame. You could have added a menu and the toolbar using Designer, however we'll do it with Python.

In lines 33 through 38 we set up the map canvas, set the background color to a light blue, and enable antialiasing. We also tell it not to use a **QImage** for rendering (trust me on this one) and then set the canvas to visible by calling the *show* method.

Next we set the layer to use a vertical box layout within the frame and add the map canvas to it in line 43.

Lines 48 to 63 set up the actions and connections for the tools in our toolbar. For each tool, we create a **QAction** using the icon we defined in the QGIS classic theme. Then we connect up the `activated` signal from the tool to the method in our class that will handle the action. This is similar to how we set things up in the plugin example.

Once we have the actions and connections, we need to add them to the toolbar. In lines 66 through 72 we create the toolbar and add each tool to it.

Lastly we create the three map tools for the application (lines 75 through 77). We'll use the map tools in a moment when we define the methods to make our application functional. Let's look at the methods for the map tools.

```
78   # Set the map tool to zoom in
79   def zoomIn(self):
80     self.canvas.setMapTool(self.toolZoomIn)
81
82   # Set the map tool to zoom out
83   def zoomOut(self):
84     self.canvas.setMapTool(self.toolZoomOut)
85
86   # Set the map tool to
87   def pan(self):
88    self.canvas.setMapTool(self.toolPan)
89
90   # Zoom to full extent of layer
91   def zoomFull(self):
92     self.canvas.zoomFullExtent()
```

For each map tool, we need a method that corresponds to the connection we made for each action. In lines 79 through 88 we set up a method for each of the three tools that interact with the map. When a tool is activated by clicking on it in the toolbar, the corresponding method is called that "tells" the map canvas it is the active tool. The active tool governs what happens when the mouse is clicked on the canvas.

The `zoom to full extent` tool isn't a map tool—it does its job without requiring a click on the map. When it is activated, we call the *zoomFullExtent* method of the map canvas (line 92). This completes the implementation of all our tools except one—the `Add Layer` tool. Let's look at it next:

```
93   # Add an OGR layer to the map
94   def addLayer(self):
95     file = QFileDialog.getOpenFileName(self, "Open Shapefile", ".", "Shapefiles
96     (*.shp)")
97     fileInfo = QFileInfo(file)
98
99     # Add the layer
100     layer = QgsVectorLayer(file, fileInfo.fileName(), "ogr")
101
102    if not layer.isValid():
103      return
104
```

```
105     # Change the color of the layer to gray
106     symbols = layer.renderer().symbols()
107     symbol = symbols[0]
108     symbol.setFillColor(QColor.fromRgb(192,192,192))
109
110     # Add layer to the registry
111     QgsMapLayerRegistry.instance().addMapLayer(layer);
112
113     # Set extent to the extent of our layer
114     self.canvas.setExtent(layer.extent())
115
116     # Set up the map canvas layer set
117     cl = QgsMapCanvasLayer(layer)
118     layers = [cl]
119     self.canvas.setLayerSet(layers)
```

In the *addLayer* method we use a **QFileDialog** to get the name of the shapefile to load. This is done in line 96. Notice that we specify a "filter" so the dialog will only show files of type `.shp`.

Next in line 97 we create a **QFileInfo** object from the shapefile path. Now the layer is ready to be created in line 100. Using the **QFileInfo** object to get the file name from the path we specify it for the name of the layer when it is created. To make sure that the layer is valid and won't cause any problems when loading, we check it in line 102. If it's bad, we bail out and don't add it to the map canvas.

Normally layers are added with a random color. Here we want to tweak the colors for the layer to make a more pleasing display. Plus we know we are going to add the `world_borders` layer to the map and this will make it look nice on our blue background. To change the color, we need to get the symbol used for rendering and use it to set a new fill color. This is done in lines 106 through 108.

All that's left is to actually add the layer to the registry and a few other housekeeping items (lines 111 through 119). This stuff is standard for adding a layer and the end result is the world borders on a light blue background. The only thing you may not want to do is set the extent to the layer, if you are going to be adding more than one layer in your application.

That's the heart of the application and completes the **MainWindow** class.

5.3 Finishing Up

The remainder of the code shown below creates the *QgsApplication* object, sets the path to the QGIS install, sets up the *main* method and then starts the application. The only other thing to note is that we move the application window to the upper left of the display. We could get fancy and use the Qt

API to center it on the screen.

```
120 def main(argv):
121   # create Qt application
122   app = QApplication(argv)
123
124   # Initialize qgis libraries
125   QgsApplication.setPrefixPath(qgis_prefix, True)
126   QgsApplication.initQgis()
127
128   # create main window
129   wnd = MainWindow()
130   # Move the app window to upper left
131   wnd.move(100,100)
132   wnd.show()
133
134   # run!
135   retval = app.exec_()
136
137   # exit
138   QgsApplication.exitQgis()
139   sys.exit(retval)
140
141
142 if __name__ == "__main__":
143   main(sys.argv)
```

5.4 Running the Application

Now we can run the application and see what happens. Of course if you are like most developers, you've been testing it out as you went along.

Before we can run the application, we need to set some environment variables.

```
export LD_LIBRARY_PATH=$HOME/qgis/lib%$
export PYTHONPATH=$HOME/qgis/share/qgis/python
export QGISHOME=$HOME/qgis%$
```

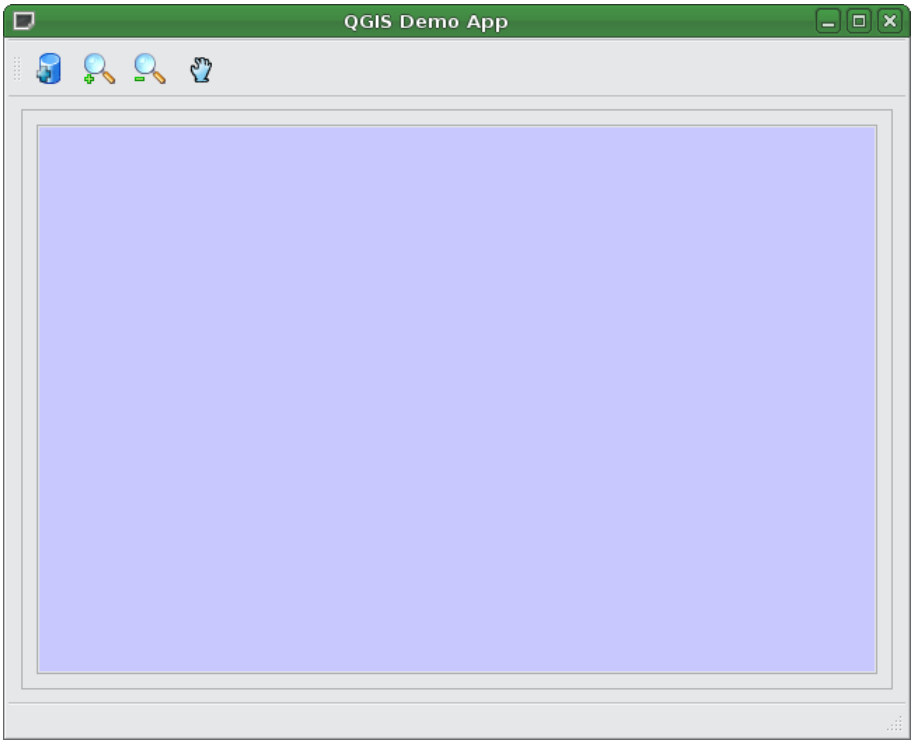
```
set PATH=C:\qgis;%PATH%
set PYTHONPATH=C:\qgis\python
set QGISHOME=C:\qgis
```

We assume

- QGIS is installed in your home directory in `qgis`.
- QGIS is installed in `C:\qgis`.

When the application starts up, it looks like this:

Figure 3: Starting the new demo application



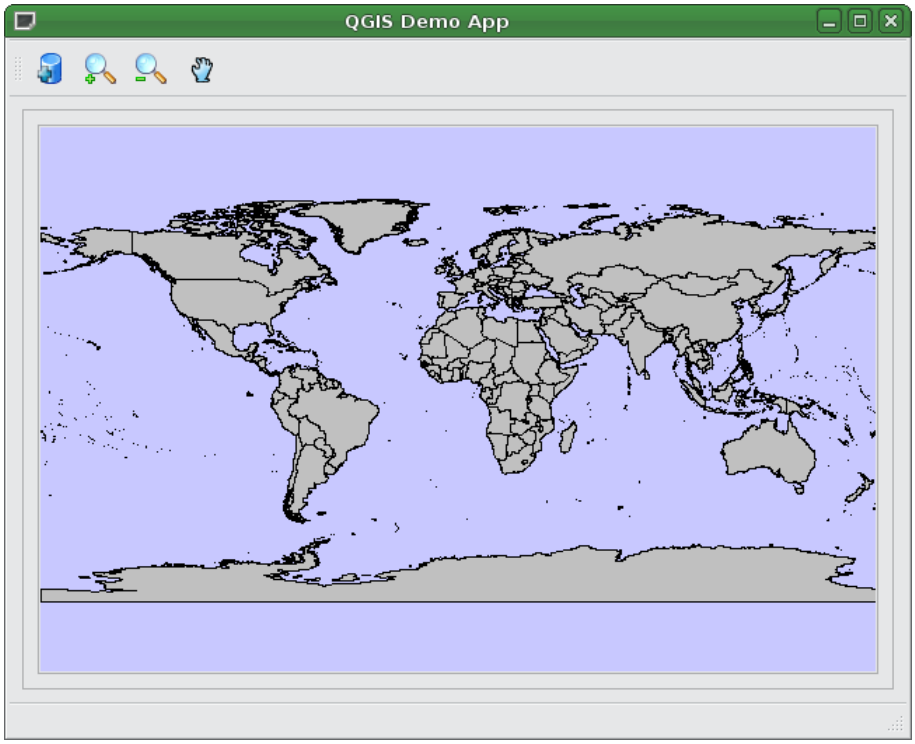
To add the `world_borders` layer, click on the `Add Layer` tool and navigate to the data directory. Select the shapefile and click Open to add it to the map. Our custom fill color is applied and the result is shown in Figure 4.

Creating a PyQGIS application is really pretty simple. In less than 150 lines of code we have an application that can load a shapefile and navigate the map. If you play around with the map, you'll notice that some of the built-in features of the canvas also work, including mouse wheel scrolling and panning by holding down the Space bar and moving the mouse.

Some sophisticated applications have been created with PyQGIS and more are in the works. This is

pretty impressive, considering that this development has taken place even before the official release of QGIS 1.0.

Figure 4: Adding a layer the demo application



6 Installation Guide

The following chapters provide build and installation information for QGIS Version 1.1. This document corresponds almost to a LaTeX conversion of the INSTALL.t2t file coming with the QGIS sources from April, 24th 2009.

A current version is also available at the wiki, see: http://wiki.qgis.org/qgiswiki/BuildingFromSource

General Build Notes

Since version 0.8.1 QGIS no longer uses the autotools for building. QGIS, like a number of major projects (eg. KDE 4.0), now uses CMake (http://www.cmake.org) for building from source. The configure script in this directory simply checks for the existence of cmake and provides some clues to build QGIS.

For complete information, see the wiki at: http://wiki.qgis.org/qgiswiki/Building_with_CMake

6.1 An overview of the dependencies required for building

Required build deps:

- CMake >= 2.4.3
- Flex, Bison

Required runtime deps:

- Qt >= 4.3.0
- Proj >= ? (known to work with 4.4.x)
- GEOS >= 2.2 (3.0 is supported, maybe 2.1.x works too)
- Sqlite3 >= ? (probably 3.0.0)
- GDAL/OGR >= 1.4.x

Optional dependencies:

- for GRASS plugin - GRASS >= 6.0.0
- for georeferencer - GSL >= ? (works with 1.8)
- for postgis support and SPIT plugin - PostgreSQL >= 8.0.x
- for gps plugin - expat >= ? (1.95 is OK)
- for mapserver export and PyQGIS - Python >= 2.3 (2.5+ preferred)
- for PyQGIS - SIP >= 4.5, PyQt >= must match Qt version

Recommended runtime deps:

- for gps plugin - gpsbabel

7 Building under windows using msys

Note: For a detailed account of building all the dependencies yourself you can visit Marco Pasetti's website here:

http://www.webalice.it/marco.pasetti/qgis+grass/BuildFromSource.html

Read on to use the simplified approach with pre-built libraries...

7.1 MSYS:

MSYS provides a unix style build environment under windows. We have created a zip archive that contains just about all dependencies.

Get this:

http://download.osgeo.org/qgis/win32/msys.zip

and unpack to c:\msys

If you wish to prepare your msys environment yourself rather than using our pre-made one, detailed instructions are provided elsewhere in this document.

7.2 Qt4.3

Download qt4.3 opensource precompiled edition exe and install (including the download and install of mingw) from here:

http://www.trolltech.com/developer/downloads/qt/windows

When the installer will ask for MinGW, you don't need to download and install it, just point the installer to c:\msys\mingw

When Qt installation is complete:

Edit C:\Qt\4.3.0\bin\qtvars.bat and add the following lines:

```
set PATH=%PATH%;C:\msys\local\bin;c:\msys\local\lib
set PATH=%PATH%;"C:\Program Files\Subversion\bin"
```

I suggest you also add C:\Qt\4.3.0\bin\ to your Environment Variables Path in the windows system preferences.

If you plan to do some debugging, you'll need to compile debug version of Qt: C:\Qt\4.3.0\bin\qtvars.bat compile_debug

Note: there is a problem when compiling debug version of Qt 4.3, the script ends with this message "mingw32-make: *** No rule to make target 'debug'. Stop.". To compile the debug version you have to go out of src directory and execute the following command:

```
c:\Qt\4.3.0 make
```

7.3 Flex and Bison

*** Note I think this section can be removed as it should be installed int the msys image already. TS

```
Get Flex
\URL{http://sourceforge.net/project/showfiles.php?group_id=23617&package_id=16424}
(the zip bin) and extract it into c:\msys\mingw\bin
```

7.4 Python stuff: (optional)

Follow this section in case you would like to use Python bindings for QGIS. To be able to compile bindings, you need to compile SIP and PyQt4 from sources as their installer doesn't include some development files which are necessary.

7.4.1 Download and install Python - use Windows installer

(It doesn't matter to what folder you'll install it)

http://python.org/download/

7.4.2 Download SIP and PyQt4 sources

- http://www.riverbankcomputing.com/software/sip/download
- http://www.riverbankcomputing.com/software/pyqt/download

Extract each of the above zip files in a temporary directory. Make sure to get versions that match your current Qt installed version.

7.4.3 Compile SIP

```
c:\Qt\4.3.0\bin\qtvars.bat
python configure.py -p win32-g++
make
make install
```

7.4.4 Compile PyQt

```
c:\Qt\4.3.0\bin\qtvars.bat
python configure.py
make
make install
```

7.4.5 Final python notes

/!\ You can delete the directories with unpacked SIP and PyQt4 sources after a successfull install, they're not needed anymore.

7.5 Subversion:

In order to check out QGIS sources from the repository, you need Subversion client. This installer should work fine:

http://subversion.tigris.org/files/documents/15/36797/svn-1.4.3-setup.exe

7.6 CMake:

CMake is build system used by Quantum GIS. Download it from here:

http://www.cmake.org/files/v2.4/cmake-2.4.6-win32-x86.exe

7.7 QGIS:

Start a cmd.exe window (Start -> Run -> cmd.exe) Create development directory and move into it

```
md c:\dev\cpp
cd c:\dev\cpp
```

Check out sources from SVN For svn head:

```
svn co https://svn.osgeo.org/qgis/trunk/qgis
```

For svn 0.8 branch

```
svn co https://svn.osgeo.org/qgis/branches/Release-0_8_0 qgis0.8
```

7.8 Compiling:

As a background read the generic building with CMake notes at the end of this document.

Start a cmd.exe window (Start -> Run -> cmd.exe) if you don't have one already. Add paths to compiler and our MSYS environment:

```
c:\Qt\4.3.0\bin\qtvars.bat
```

For ease of use add c:\Qt\4.3.0\bin\ to your system path in system properties so you can just type qtvars.bat when you open the cmd console. Create build directory and set it as current directory:

```
cd c:\dev\cpp\qgis
md build
cd build
```

7.9 Configuration

```
cmakesetup ..
```

/!\ NOTE: You must include the '..' above.

Click 'Configure' button. When asked, you should choose 'MinGW Makefiles' as generator.

There's a problem with MinGW Makefiles on Win2K. If you're compiling on this platform, use 'MSYS Makefiles' generator instead.

All dependencies should be picked up automatically, if you have set up the Paths correctly. The only thing you need to change is the installation destination (CMAKE_INSTALL_PREFIX) and/or set 'Debug'.

For compatibility with NSIS packaging cripts I recommend to leave the install prefix to its default c:\program files\

When configuration is done, click 'OK' to exit the setup utility.

7.10 Compilation and installation

```
make make install
```

7.11 Run qgis.exe from the directory where it's installed (CMAKE_INSTALL_PREFIX)

Make sure to copy all .dll:s needed to the same directory as the qgis.exe binary is installed to, if not already done so, otherwise QGIS will complain about missing libraries when started.

The best way to do this is to download both the QGIS current release installer package from http://qgis.org/uploadfiles/testbuilds/ and install it. Now copy the installation dir from C:\Program Files\Quantum GIS into c:\Program Files\qgis-0.8.1 (or whatever the current version is. The name should strictly match the version no.) After making this copy you can uninstall the release version of QGIS from your c:\Program Files directory using the provided uninstaller. Double check that the Quantum GIS dir is completely gone under program files afterwards.

Another possibility is to run qgis.exe when your path contains c:\msys\local\bin and c:\msys\local\lib directories, so the DLLs will be used from that place.

7.12 Create the installation package: (optional)

Downlad and install NSIS from (http://nsis.sourceforge.net/Main_Page)

Now using windows explorer, enter the win_build directory in your QGIS source tree. Read the READMEfile there and follow the instructions. Next right click on qgis.nsi and choose the option 'Compile NSIS Script'.

8 Building on Mac OSX using frameworks and cmake (QGIS > 0.8)

In this approach I will try to avoid as much as possible building dependencies from source and rather use frameworks wherever possible.

Included are a few notes for building on Mac OS X 10.5 (Leopard).

8.1 Install XCODE

I recommend to get the latest xcode dmg from the Apple XDC Web site. Install XCODE after the ˜941mb download is complete.

/!\ Note: It may be that you need to create some symlinks after installing the XCODE SDK (in particular if you are using XCODE 2.5 on tiger):

```
cd /Developer/SDKs/MacOSX10.4u.sdk/usr/
sudo mv local/ local_
sudo ln -s /usr/local/ local
```

8.2 Install Qt4 from .dmg

You need a minimum of Qt4.3.0. I suggest getting the latest (at time of writing).

```
ftp://ftp.trolltech.com/qt/source/qt-mac-opensource-4.3.2.dmg
```

If you want debug libs, Qt also provide a dmg with these:

```
ftp://ftp.trolltech.com/qt/source/qt-mac-opensource-4.3.2-debug-libs.dmg
```

I am going to proceed using only release libs at this stage as the download for the debug dmg is substantially bigger. If you plan to do any debugging though you probably want to get the debug libs dmg. Once downloaded open the dmg and run the installer. Note you need admin access to install.

After installing you need to make two small changes:

First edit `/Library/Frameworks/QtCore.framework/Headers/qconfig.h` and change

/!\ Note this doesnt seem to be needed since version 4.2.3

`QT_EDITION_Unknown` to `QT_EDITION_OPENSOURCE`

Second change the default mkspec symlink so that it points to macx-g++:

```
cd /usr/local/Qt4.3/mkspecs/
sudo rm default
sudo ln -sf macx-g++ default
```

8.3 Install development frameworks for QGIS dependencies

Download William Kyngesburye's excellent all in one framework that includes proj, gdal, sqlite3 etc

```
http://www.kyngchaos.com/wiki/software:frameworks
```

Once downloaded, open and install the frameworks.

William provides an additional installer package for Postgresql/PostGIS. Its available here:

```
http://www.kyngchaos.com/wiki/software:postgres
```

There are some additional dependencies that at the time of writing are not provided as frameworks so we will need to build these from source.

8.3.1 Additional Dependencies : GSL

Retrieve the Gnu Scientific Library from

```
curl -O ftp://ftp.gnu.org/gnu/gsl/gsl-1.8.tar.gz
```

Then extract it and build it to a prefix of /usr/local:

```
tar xvfz gsl-1.8.tar.gz
cd gsl-1.8
./configure --prefix=/usr/local
make
sudo make install
cd ..
```

8.3.2 Additional Dependencies : Expat

Get the expat sources:

```
http://sourceforge.net/project/showfiles.php?group_id=10127
```

```
tar xvfz expat-2.0.0.tar.gz
cd expat-2.0.0
./configure --prefix=/usr/local
make
sudo make install
cd ..
```

8.3.3 Additional Dependencies : SIP

Make sure you have the latest Python fom

```
http://www.python.org/download/mac/
```

Leopard note: Leopard includes a usable Python 2.5. Though you can install Python from python.org if preferred.

Retrieve the python bindings toolkit SIP from

```
http://www.riverbankcomputing.com/software/sip/download
```

Then extract and build it (this installs by default into the Python framework):

```
tar xvfz sip-<version number>.tar.gz
cd sip-<version number>
python configure.py
make
sudo make install
cd ..
```

Leopard notes

If building on Leopard, using Leopard's bundled Python, SIP wants to install in the system path – this is not a good idea. Use this configure command instead of the basic configure above:

```
python configure.py -d /Library/Python/2.5/site-packages -b /usr/local/bin \
-e /usr/local/include -v /usr/local/share/sip
```

8.3.4 Additional Dependencies : PyQt

If you encounter problems compiling PyQt using the instructions below you can also try adding python from your frameworks dir explicitly to your path e.g.

```
export PATH=/Library/Frameworks/Python.framework/Versions/Current/bin:$PATH$
```

Retrieve the python bindings toolkit for Qt from

```
http://www.riverbankcomputing.com/software/pyqt/download
```

Then extract and build it (this installs by default into the Python framework):

```
tar xvfz PyQt-mac<version number here>
cd PyQt-mac<version number here>
export QTDIR=/Developer/Applications/Qt
python configure.py
yes
make
sudo make install
cd ..
```

Leopard notes

If building on Leopard, using Leopard's bundled Python, PyQt wants to install in the system path – this is not a good idea. Use this configure command instead of the basic configure above:

```
python configure.py -d /Library/Python/2.5/site-packages -b /usr/local/bin
```

There may be a problem with undefined symbols in QtOpenGL on Leopard. Edit QtOpenGL/makefile and add -undefined dynamic_lookup to LFLAGS.

8.3.5 Additional Dependencies : Bison

Leopard note: Leopard includes Bison 2.3, so this step can be skipped on Leopard.

The version of bison available by default on Mac OSX is too old so you need to get a more recent one on your system. Download if from:

```
curl -O http://ftp.gnu.org/gnu/bison/bison-2.3.tar.gz
```

Now build and install it to a prefix of /usr/local :

```
tar xvfz bison-2.3.tar.gz
cd bison-2.3
./configure --prefix=/usr/local
make
sudo make install
cd ..
```

8.4 Install CMAKE for OSX

Get the latest release from here:

```
http://www.cmake.org/HTML/Download.html
```

At the time of writing the file I grabbed was:

```
curl -O http://www.cmake.org/files/v2.4/cmake-2.4.6-Darwin-universal.dmg
```

Once downloaded open the dmg and run the installer

8.5 Install subversion for OSX

Leopard note: Leopard includes SVN, so this step can be skipped on Leopard.

The http://sourceforge.net/projects/macsvn/ project has a downloadable build of svn. If you are a GUI inclined person you may want to grab their gui client too. Get the command line client here:

```
curl -O http://ufpr.dl.sourceforge.net/sourceforge/macsvn/Subversion_1.4.2.zip
```

Once downloaded open the zip file and run the installer.

You also need to install BerkleyDB available from the same http://sourceforge.net/projects/macsvn/. At the time of writing the file was here:

```
curl -O http://ufpr.dl.sourceforge.net/sourceforge/macsvn/Berkeley_DB_4.5.20.zip
```

Once again unzip this and run the installer therein.

Lastly we need to ensure that the svn commandline executeable is in the path. Add the following line to the end of /etc/bashrc using sudo:

```
sudo vim /etc/bashrc
```

And add this line to the bottom before saving and quiting:

```
export PATH=/usr/local/bin:$PATH:/usr/local/pgsql/bin
```

/usr/local/bin needs to be first in the path so that the newer bison (that will be built from source further down) is found before the bison (which is very old) that is installed by MacOSX

Now close and reopen your shell to get the updated vars.

8.6 Check out QGIS from SVN

Now we are going to check out the sources for QGIS. First we will create a directory for working in:

```
mkdir -p ~/dev/cpp cd ~/dev/cpp
```

Now we check out the sources:

Trunk:

```
svn co https://svn.osgeo.org/qgis/trunk/qgis qgis
```

For svn 0.8 branch

```
svn co https://svn.osgeo.org/qgis/branches/Release-0_8_0 qgis0.8
```

For svn 0.9 branch

```
svn co https://svn.qgis.org/qgis/branches/Release-0_9_0 qgis0.9
```

The first time you check out QGIS sources you will probably get a message like this:

```
Error validating server certificate for 'https://svn.qgis.org:443':
- The certificate is not issued by a trusted authority. Use the fingerprint to
  validate the certificate manually!  Certificate information:
- Hostname: svn.qgis.org
- Valid: from Apr  1 00:30:47 2006 GMT until Mar 21 00:30:47 2008 GMT
- Issuer: Developer Team, Quantum GIS, Anchorage, Alaska, US
- Fingerprint: 2f:cd:f1:5a:c7:64:da:2b:d1:34:a5:20:c6:15:67:28:33:ea:7a:9b
  (R)eject, accept (t)emporarily or accept (p)ermanently?
```

I suggest you press 'p' to accept the key permanently.

8.7 Configure the build

CMake supports out of source build so we will create a 'build' dir for the build process. By convention I build my software into a dir called 'apps' in my home directory. If you have the correct permissions you may want to build straight into your /Applications folder. The instructions below assume you are building into a pre-existing ${HOME}/apps directory ...

```
cd qgis
mkdir build
cd build
cmake -D CMAKE_INSTALL_PREFIX=$HOME/apps/ -D CMAKE_BUILD_TYPE=Release ..
```

Leopard note: To find the custom install of SIP on Leopard, add ""- D SIP_BINARY_-PATH=/usr/local/bin/sip"" to the cmake command above, before the .. at the end, ie:

```
cmake -D CMAKE_INSTALL_PREFIX=$HOME/apps/ -D CMAKE_BUILD_TYPE=Release -
D SIP_BINARY_PATH=/usr/local/bin/sip ..
```

To use the application build of GRASS on OSX, you can optionally use the following cmake invocation (minimum GRASS 6.3 required, substitute the GRASS version as required):

```
cmake -D CMAKE_INSTALL_PREFIX=${HOME}/apps/ \
      -D GRASS_INCLUDE_DIR=/Applications/GRASS-6.3.app/Contents/MacOS/
      include \
      -D GRASS_PREFIX=/Applications/GRASS-6.3.app/Contents/MacOS \
      -D CMAKE_BUILD_TYPE=Release \
      ..
```

Or, to use a Unix-style build of GRASS, use the following cmake invocation (minimum GRASS version as stated in the Qgis requirements, substitute the GRASS path and version as required):

```
cmake -D CMAKE_INSTALL_PREFIX=${HOME}/apps/ \
  -D GRASS_INCLUDE_DIR=/user/local/grass-6.3.0/include \
  -D GRASS_PREFIX=/user/local/grass-6.3.0 \
  -D CMAKE_BUILD_TYPE=Release \
  ..
```

8.8 Building

Now we can start the build process:

```
make
```

If all built without errors you can then install it:

```
make install
```

9 Building on GNU/Linux

9.1 Building QGIS with Qt4.x

Requires: Ubuntu Hardy / Debian derived distro

These notes are current for Ubuntu 7.10 - other versions and Debian derived distros may require slight variations in package names.

These notes are for if you want to build QGIS from source. One of the major aims here is to show how this can be done using binary packages for ***all*** dependencies - building only the core QGIS stuff from source. I prefer this approach because it means we can leave the business of managing system packages to apt and only concern ourselves with coding QGIS!

This document assumes you have made a fresh install and have a 'clean' system. These instructions should work fine if this is a system that has already been in use for a while, you may need to just skip those steps which are irrelevant to you.

9.2 Prepare apt

The packages qgis depends on to build are available in the "universe" component of Ubuntu. This is not activated by default, so you need to activate it:

1. Edit your /etc/apt/sources.list file. 2. Uncomment the all the lines starting with "deb"

Also you will need to be running (K)Ubuntu 'edgy' or higher in order for all dependencies to be met.

Now update your local sources database:

```
sudo apt-get update
```

9.3 Install Qt4

```
sudo apt-get install libqt4-core libqt4-debug  \
libqt4-dev libqt4-gui libqt4-qt3support libqt4-sql lsb-qt4 qt4-designer \
qt4-dev-tools qt4-doc qt4-qtconfig uim-qt gcc libapt-pkg-perl resolvconf
```

/!\ ***A Special Note:*** If you are following this set of instructions on a system where you already have Qt3 development tools installed, there will be a conflict between Qt3 tools and Qt4 tools. For example, qmake will point to the Qt3 version not the Qt4. Ubuntu Qt4 and Qt3 packages are designed to live alongside each other. This means that for example if you have them both installed you will have three qmake exe's:

```
/usr/bin/qmake -> /etc/alternatives/qmake
/usr/bin/qmake-qt3
/usr/bin/qmake-qt4
```

The same applies to all other Qt binaries. You will notice above that the canonical 'qmake' is managed by apt alternatives, so before we start to build QGIS, we need to make Qt4 the default. To return Qt3 to default later you can use this same process.

You can use apt alternatives to correct this so that the Qt4 version of applications is used in all cases:

```
sudo update-alternatives --config qmake
sudo update-alternatives --config uic
sudo update-alternatives --config designer
sudo update-alternatives --config assistant
sudo update-alternatives --config qtconfig
sudo update-alternatives --config moc
```

```
sudo update-alternatives --config lupdate
sudo update-alternatives --config lrelease
sudo update-alternatives --config linguist
```

Use the simple command line dialog that appears after running each of the above commands to select the Qt4 version of the relevant applications.

9.4 Install additional software dependencies required by QGIS

```
sudo apt-get install gdal-bin libgdal1-dev libgeos-dev proj \
libgdal-doc libhdf4g-dev libhdf4g-run python-dev \
libgsl0-dev g++ libjasper-dev libtiff4-dev subversion \
libsqlite3-dev sqlite3 ccache make libpq-dev flex bison cmake txt2tags \
python-qt4 python-qt4-dev python-sip4 sip4 python-sip4-dev
```

/!\ Debian users should use libgdal-dev above rather

/!\ ***Note:*** For python language bindings SIP >= 4.5 and PyQt4 >= 4.1 is required! Some stable GNU/Linux distributions (e.g. Debian or SuSE) only provide SIP < 4.5 and PyQt4 < 4.1. To include support for python language bindings you may need to build and install those packages from source.

If you do not have cmake installed already:

```
sudo apt-get install cmake
```

9.5 GRASS Specific Steps

/!\ ***Note:*** If you don't need to build with GRASS support, you can skip this section.

Now you can install grass from dapper:

```
sudo apt-get install grass libgrass-dev libgdal1-1.4.0-grass
```

/!\ You may need to explicitly state your grass version e.g. libgdal1-1.3.2-grass

9.6 Setup ccache (Optional)

You should also setup ccache to speed up compile times:

```
cd /usr/local/bin
sudo ln -s /usr/bin/ccache gcc
sudo ln -s /usr/bin/ccache g++
```

9.7 Prepare your development environment

As a convention I do all my development work in $HOME/dev/<language>, so in this case we will create a work environment for C++ development work like this:

```
mkdir -p ${HOME}/dev/cpp
cd ${HOME}/dev/cpp
```

This directory path will be assumed for all instructions that follow.

9.8 Check out the QGIS Source Code

There are two ways the source can be checked out. Use the anonymous method if you do not have edit privaleges for the QGIS source repository, or use the developer checkout if you have permissions to commit source code changes.

1. Anonymous Checkout

```
cd ${HOME}/dev/cpp
svn co https://svn.osgeo.org/qgis/trunk/qgis qgis
```

2. Developer Checkout

```
cd ${HOME}/dev/cpp
svn co --username <yourusername> https://svn.osgeo.org/qgis/trunk/qgis qgis
```

The first time you check out the source you will be prompted to accept the qgis.org certificate. Press 'p' to accept it permanently:

```
Error validating server certificate for 'https://svn.qgis.org:443':
   - The certificate is not issued by a trusted authority. Use the
     fingerprint to validate the certificate manually!  Certificate
     information:
   - Hostname: svn.qgis.org
```

```
- Valid: from Apr  1 00:30:47 2006 GMT until Mar 21 00:30:47 2008 GMT
- Issuer: Developer Team, Quantum GIS, Anchorage, Alaska, US
- Fingerprint:
  2f:cd:f1:5a:c7:64:da:2b:d1:34:a5:20:c6:15:67:28:33:ea:7a:9b (R)eject,
  accept (t)emporarily or accept (p)ermanently?
```

9.9 Starting the compile

/!\ ***Note:*** The next section describes howto build debian packages

I compile my development version of QGIS into my ~/apps directory to avoid conflicts with Ubuntu packages that may be under /usr. This way for example you can use the binary packages of QGIS on your system along side with your development version. I suggest you do something similar:

```
mkdir -p ${HOME}/apps
```

Now we create a build directory and run ccmake:

```
cd qgis
mkdir build
cd build
ccmake ..
```

When you run ccmake (note the .. is required!), a menu will appear where you can configure various aspects of the build. If you do not have root access or do not want to overwrite existing QGIS installs (by your packagemanager for example), set the CMAKE_BUILD_PREFIX to somewhere you have write access to (I usually use /home/timlinux/apps). Now press 'c' to configure, 'e' to dismiss any error messages that may appear. and 'g' to generate the make files. Note that sometimes 'c' needs to be pressed several times before the 'g' option becomes available. After the 'g' generation is complete, press 'q' to exit the ccmake interactive dialog.

Now on with the build:

```
make
make install
```

It may take a little while to build depending on your platform.

9.10 Building Debian packages

Instead of creating a personal installation as in the previous step you can also create debian package. This is done from the qgis root directory, where you'll find a debian directory.

First you need to install the debian packaging tools once:

```
apt-get install build-essential
```

The QGIS packages will be created with:

```
dpkg-buildpackage -us -us -b
```

/!\ ***Note:*** If `dpkg-buildpackage` complains about unmet build dependencies you can install them using `apt-get` and re-run the command.

/!\ ***Note:*** If you have `libqgis1-dev` installed, you need to remove it first using `dpkg -r libqgis1-dev`. Otherwise `dpkg-buildpackage` will complain about a build conflict.

The the packages are created in the parent directory (ie. one level up). Install them using dpkg. E.g.:

```
sudo dpkg -i \
../qgis_1.0preview16_amd64.deb \
../libqgis-gui1_1.0preview16_amd64.deb \
../libqgis-core1_1.0preview16_amd64.deb \
../qgis-plugin-grass_1.0preview16_amd64.deb \
../python-qgis_1.0preview16_amd64.deb
```

9.11 Running QGIS

Now you can try to run QGIS:

```
$HOME/apps/bin/qgis
```

If all has worked properly the QGIS application should start up and appear on your screen.

10 Creation of MSYS environment for compilation of Quantum GIS

10.1 Initial setup

10.1.1 MSYS

This is the environment that supplies many utilities from UNIX world in Windows and is needed by many dependencies to be able to compile.

Download from here:

http://puzzle.dl.sourceforge.net/sourceforge/mingw/MSYS-1.0.11-2004.04.30-1.exe

Install to `c:\msys`

All stuff we're going to compile is going to get to this directory (resp. its subdirs).

10.1.2 MinGW

Download from here:

http://puzzle.dl.sourceforge.net/sourceforge/mingw/MinGW-5.1.3.exe

Install to `c:\msys\mingw`

It suffices to download and install only `g++` and `mingw-make` components.

10.1.3 Flex and Bison

Flex and Bison are tools for generation of parsers, they're needed for GRASS and also QGIS compilation.

Download the following packages:

http://gnuwin32.sourceforge.net/downlinks/flex-bin-zip.php

http://gnuwin32.sourceforge.net/downlinks/bison-bin-zip.php

http://gnuwin32.sourceforge.net/downlinks/bison-dep-zip.php

Unpack them all to `c:\msys\local`

10.2 Installing dependencies

10.2.1 Getting ready

Paul Kelly did a great job and prepared a package of precompiled libraries for GRASS. The package currently includes:

- zlib-1.2.3
- libpng-1.2.16-noconfig
- xdr-4.0-mingw2
- freetype-2.3.4
- fftw-2.1.5
- PDCurses-3.1
- proj-4.5.0
- gdal-1.4.1

It's available for download here:

http://www.stjohnspoint.co.uk/grass/wingrass-extralibs.tar.gz

Moreover he also left the notes how to compile it (for those interested):

http://www.stjohnspoint.co.uk/grass/README.extralibs

Unpack the whole package to `c:\msys\local`

10.2.2 GDAL level one

Since Quantum GIS needs GDAL with GRASS support, we need to compile GDAL from source - Paul Kelly's package doesn't include GRASS support in GDAL. The idea is following:

1. compile GDAL without GRASS
2. compile GRASS
3. compile GDAL with GRASS

So, start with downloading GDAL sources:

http://download.osgeo.org/gdal/gdal141.zip

Unpack it to some directory, preferably `c:\msys\local\src`.

Start MSYS console, go to gdal-1.4.1 directory and run the commands below. You can put them all to a script, e.g. build-gdal.sh and run them at once. The recipe is taken from Paul Kelly's instructions - basically they just make sure that the library will be created as DLL and the utility programs will be dynamically linked to it...

```
CFLAGS="-O2 -s" CXXFLAGS="-O2 -s" LDFLAGS=-s ./configure --without-libtool \
--prefix=/usr/local --enable-shared --disable-static --with-libz=/usr/local \
--with-png=/usr/local
make
make install
rm /usr/local/lib/libgdal.a
g++ -s -shared -o ./libgdal.dll -L/usr/local/lib -lz -lpng ./frmts/o/*.o \
./gcore/*.o ./port/*.o ./alg/*.o ./ogr/ogrsf_frmts/o/*.o \
./ogr/ogrgeometryfactory.o ./ogr/ogrpoint.o ./ogr/ogrcurve.o \
./ogr/ogrlinestring.o ./ogr/ogrlinearring.o ./ogr/ogrpolygon.o \
./ogr/ogrutils.o ./ogr/ogrgeometry.o ./ogr/ogrgeometrycollection.o \
./ogr/ogrmultipolygon.o ./ogr/ogrsurface.o ./ogr/ogrmultipoint.o \
./ogr/ogrmultilinestring.o ./ogr/ogr_api.o ./ogr/ogrfeature.o \
./ogr/ogrfeaturedefn.o ./ogr/ogrfeaturequery.o ./ogr/ogrfeaturestyle.o \
./ogr/ogrfielddefn.o ./ogr/ogrspatialreference.o \
./ogr/ogr_srsnode.o ./ogr/ogr_srs_proj4.o ./ogr/ogr_fromepsg.o ./ogr/ogrct.o \
./ogr/ogr_opt.o ./ogr/ogr_srs_esri.o ./ogr/ogr_srs_pci.o ./ogr/ogr_srs_usgs.o \
./ogr/ogr_srs_dict.o ./ogr/ogr_srs_panorama.o ./ogr/swq.o \
./ogr/ogr_srs_validate.o ./ogr/ogr_srs_xml.o ./ogr/ograssemblepolygon.o \
./ogr/ogr2gmlgeometry.o ./ogr/gml2ogrgeometry.o
install libgdal.dll /usr/local/lib
cd ogr
g++ -s ogrinfo.o -o ogrinfo.exe -L/usr/local/lib -lpng -lz -lgdal
g++ -s ogr2ogr.o -o ogr2ogr.exe -lgdal -L/usr/local/lib -lpng -lz -lgdal
g++ -s ogrtindex.o -o ogrtindex.exe -lgdal -L/usr/local/lib -lpng -lz -lgdal
install ogrinfo.exe ogr2ogr.exe ogrtindex.exe /usr/local/bin
cd ../apps
g++ -s gdalinfo.o -o gdalinfo.exe -L/usr/local/lib -lpng -lz -lgdal
g++ -s gdal_translate.o -o gdal_translate.exe -L/usr/local/lib -lpng -lz -lgdal
g++ -s gdaladdo.o -o gdaladdo.exe -L/usr/local/lib -lpng -lz -lgdal
g++ -s gdalwarp.o -o gdalwarp.exe -L/usr/local/lib -lpng -lz -lgdal
g++ -s gdal_contour.o -o gdal_contour.exe -L/usr/local/lib -lpng -lz -lgdal
g++ -s gdaltindex.o -o gdaltindex.exe -L/usr/local/lib -lpng -lz -lgdal
g++ -s gdal_rasterize.o -o gdal_rasterize.exe -L/usr/local/lib -lpng -lz -lgdal
install gdalinfo.exe gdal_translate.exe gdaladdo.exe gdalwarp.exe gdal_contour.exe \
gdaltindex.exe gdal_rasterize.exe /usr/local/bin
```

Finally, manually edit `gdal-config` in `c:\msys\local\bin` to replace the static library reference with -lgdal:

```
CONFIG_LIBS="-L/usr/local/lib -lpng -lz -lgdal"
```

GDAL build procedure can be greatly simplified to use libtool with a libtool line patch: configure gdal as below: ./configure –with-ngpython –with-xerces=/local/ –with-jasper=/local/ –with-grass=/local/grass-6.3.cvs/ –with-pg=/local/pgsql/bin/pg_config.exe

Then fix libtool with: mv libtool libtool.orig cat libtool.orig | sed 's/max_cmd_len=8192/max_cmd_-len=32768/g' > libtool

Libtool on windows assumes a line length limit of 8192 for some reason and tries to page the linking and fails miserably. This is a work around.

Make and make install should be hassle free after this.

10.2.3 GRASS

Grab sources from CVS or use a weekly snapshot, see:

http://grass.itc.it/devel/cvs.php

In MSYS console go to the directory where you've unpacked or checked out sources (e.g. `c:\msys\local\src\grass-6.3.cvs`)

Run these commands:

```
export PATH="/usr/local/bin:/usr/local/lib:$PATH"
./configure --prefix=/usr/local --bindir=/usr/local --with-includes=/usr/local/include \
--with-libs=/usr/local/lib --with-cxx --without-jpeg --without-tiff \
--with-postgres=yes \ --with-postgres-includes=/local/pgsql/include \
--with-pgsql-libs=/local/pgsql/lib \
--with-opengl=windows --with-fftw --with-freetype \
--with-freetype-includes=/mingw/include/freetype2 --without-x --without-tcltk \
--enable-x11=no --enable-shared=yes --with-proj-share=/usr/local/share/proj
make
make install
```

It should get installed to `c:\msys\local\grass-6.3.cvs`

By the way, these pages might be useful:

- http://grass.osgeo.org/wiki/WinGRASS_Current_Status
- http://geni.ath.cx/grass.html

10.2.4 GDAL level two

At this stage, we'll use GDAL sources we've used before, only the compilation will be a bit different.

But first in order to be able to compile GDAL sources with current GRASS CVS, you need to patch them, here's what you need to change:

http://trac.osgeo.org/gdal/attachment/ticket/1587/plugin_patch_grass63.diff

(you can patch it by hand or use patch.exe in `c:\msys\bin`)

Now in MSYS console go to the GDAL sources directory and run the same commands as in level one, only with these differences:

1. when running `./configure` add this argument: `-with-grass=/usr/local/grass-6.3.cvs`
2. when calling `g++` on line 5 (which creates libgdal.dll), add these arguments: `-L/usr/local/grass-6.3.cvs/lib -lgrass_vect -lgrass_dig2 -lgrass_dgl -lgrass_rtree -lgrass_linkm -lgrass_dbmiclient -lgrass_dbmibase -lgrass_I -lgrass_gproj -lgrass_vask -lgrass_gmath -lgrass_gis -lgrass_datetime`

Then again, edit `gdal-config` and change line with CONFIG_LIBS

```
CONFIG_LIBS="-L/usr/local/lib -lpng -L/usr/local/grass-6.3.cvs/lib -lgrass_vect \
-lgrass_dig2 -lgrass_dgl -lgrass_rtree -lgrass_linkm -lgrass_dbmiclient \
-lgrass_dbmibase -lgrass_I -lgrass_gproj -lgrass_vask -lgrass_gmath -lgrass_gis \
-lgrass_datetime -lz -L/usr/local/lib -lgdal"
```

Now, GDAL should be able to work also with GRASS raster layers.

10.2.5 GEOS

Download the sources:

http://geos.refractions.net/geos-2.2.3.tar.bz2

Unpack to e.g. `c:\msys\local\src`

To compile, I had to patch the sources: in file `source/headers/timeval.h` line 13. Change it from:

```
#ifdef _WIN32
```

to:

```
#if defined(_WIN32) && defined(_MSC_VER)
```

Now, in MSYS console, go to the source directory and run:

```
./configure --prefix=/usr/local
make
make install
```

10.2.6 SQLITE

You can use precompiled DLL, no need to compile from source:

Download this archive:

http://www.sqlite.org/sqlitedll-3_3_17.zip

and copy sqlite3.dll from it to `c:\msys\local\lib`

Then download this archive:

http://www.sqlite.org/sqlite-source-3_3_17.zip

and copy sqlite3.h to `c:\msys\local\include`

10.2.7 GSL

Download sources:

ftp://ftp.gnu.org/gnu/gsl/gsl-1.9.tar.gz

Unpack to `c:\msys\local\src`

Run from MSYS console in the source directory:

```
./configure
make
make install
```

10.2.8 EXPAT

Download sources:

http://dfn.dl.sourceforge.net/sourceforge/expat/expat-2.0.0.tar.gz

Unpack to `c:\msys\local\src`

Run from MSYS console in the source directory:

```
./configure
make
make install
```

10.2.9 POSTGRES

We're going to use precompiled binaries. Use the link below for download:

`http://wwwmaster.postgresql.org/download/mirrors-ftp?file=%2Fbinary%2Fv8.2.4%2Fwin32%2\`
`Fpostgresql-8.2.4-1-binaries-no-installer.zip}`

copy contents of pgsql directory from the archive to `c:\msys\local`

10.3 Cleanup

We're done with preparation of MSYS environment. Now you can delete all stuff in `c:\msys\local\src` - it takes quite a lot of space and it's not necessary at all.

11 Building with MS Visual Studio

/!\ This section describes a process where you build all dependencies yourself. See the section after this for a simpler procedure where we have all the dependencies you need pre-packaged and we focus just on getting Visual Studio Express set up and building QGIS.

Note that this does not currently include GRASS or Python plugins.

11.1 Setup Visual Studio

This section describes the setup required to allow Visual Studio to be used to build QGIS.

11.1.1 Express Edition

The free Express Edition lacks the platform SDK which contains headers and so on that are needed when building QGIS. The platform SDK can be installed as described here:

http://msdn.microsoft.com/vstudio/express/visualc/usingpsdk/

Once this is done, you will need to edit the <vsinstalldir>\Common7\Tools\vsvars file as follows:

Add `%PlatformSDKDir%\Include\atl` and `%PlatformSDKDir%\Include\mfc` to the `@set INCLUDE` entry.

This will add more headers to the system INCLUDE path. Note that this will only work when you use the Visual Studio command prompt when building. Most of the dependencies will be built with this. You will also need to perform the edits described here to remove the need for a library that Visual Studio Express lacks:

http://www.codeproject.com/wtl/WTLExpress.asp

11.1.2 All Editions

You will need stdint.h and unistd.h. unistd.h comes with GnuWin32 version of flex & bison binaries (see later). stdint.h can be found here:

http://www.azillionmonkeys.com/qed/pstdint.h.

Copy both of these to <vsinstalldir>\VC\include.

11.2 Download/Install Dependencies

This section describes the downloading and installation of the various QGIS dependencies.

11.2.1 Flex and Bison

Flex and Bison are tools for generation of parsers, they're needed for GRASS and also QGIS compilation.

Download the following packages and run the installers:

http://gnuwin32.sourceforge.net/downlinks/flex.php

http://gnuwin32.sourceforge.net/downlinks/bison.php

11.2.2 To include PostgreSQL support in Qt

If you want to build Qt with PostgreSQL support you need to download PostgreSQL, install it and create a library you can later link with Qt.

Download from .../binary/v8.2.5/win32/postgresql-8.2.5-1.zip from an PostgreSQL.org Mirror and install.

PostgreSQL is currently build with MinGW and comes with headers and libraries for MinGW. The headers can be used with Visual C++ out of the box, but the library is only shipped in DLL and archive (.a) form and therefore cannot be used with Visual C++ directly.

To create a library copy following sed script to the file mkdef.sed in PostgreSQL lib directory:

```
/Dump of file / {
s/Dump of file \([^  ]*\)$/LIBRARY \1/p
a\
EXPORTS
}
/[  ]*ordinal hint/,/^[ ]*Summary/ {
 /^[  ]\+[0-9]\+/ {
   s/^[  ]\+[0-9]\+[  ]\+[0-9A-Fa-f]\+[  ]\+[0-9A-Fa-f]\+[  ]\+\([^  =]\+\).*$/ \1/p
 }
}
```

and process execute in the Visual Studio C++ command line (from Programs menu):

```
cd c:\Program Files\PostgreSQL\8.2\bin
dumpbin /exports ..\bin\libpq.dll | sed -nf ../lib/mkdef.sed >..\lib\libpq.def
cd ..\lib
lib /def:libpq.def /machine:x86
```

You'll need an sed for that to work in your path (e.g. from cygwin or msys).

That's almost it. You only need to the include and lib path to INCLUDE and LIB in vcvars.bat respectively.

11.2.3 Qt

Build Qt following the instructions here:

```
http://wiki.qgis.org/qgiswiki/Building\_QT\_4\_with\_Visual\_C%2B%2B\_2005}
```

11.2.4 Proj.4

Get proj.4 source from here:

http://proj.maptools.org/

Using the Visual Studio command prompt (ensures the environment is setup properly), run the following in the src directory:

```
nmake -f makefile.vc
```

Install by running the following in the top level directory setting PROJ_DIR as appropriate:

```
set PROJ_DIR=c:\lib\proj

mkdir %PROJ_DIR%\bin
mkdir %PROJ_DIR%\include
mkdir %PROJ_DIR%\lib

copy src\*.dll %PROJ_DIR%\bin
copy src\*.exe %PROJ_DIR%\bin
copy src\*.h %PROJ_DIR%\include
copy src\*.lib %PROJ_DIR%\lib
```

This can also be added to a batch file.

11.2.5 GSL

Get gsl source from here:

http://david.geldreich.free.fr/downloads/gsl-1.9-windows-sources.zip

Build using the gsl.sln file

11.2.6 GEOS

Get geos from svn (svn checkout http://svn.refractions.net/geos/trunk geos). Edit geos\source\makefile.vc as follows:

Uncomment lines 333 and 334 to allow the copying of version.h.vc to version.h.

Uncomment lines 338 and 339.

Rename geos_c.h.vc to geos_c.h.in on lines 338 and 339 to allow the copying of geos_c.h.in to geos_c.h.

Using the Visual Studio command prompt (ensures the environment is setup properly), run the following in the top level directory:

```
nmake -f makefile.vc
```

Run the following in top level directory, setting GEOS_DIR as appropriate:

```
set GEOS_DIR="c:\lib\geos"

mkdir %GEOS_DIR%\include
mkdir %GEOS_DIR%\lib
mkdir %GEOS_DIR%\bin

xcopy /S/Y source\headers\*.h %GEOS_DIR%\include
copy /Y capi\*.h %GEOS_DIR%\include
copy /Y source\*.lib %GEOS_DIR%\lib
copy /Y source\*.dll %GEOS_DIR%\bin
```

This can also be added to a batch file.

11.2.7 GDAL

Get gdal from svn (svn checkout https://svn.osgeo.org/gdal/branches/1.4/gdal gdal).

Edit nmake.opt to suit, it's pretty well commented.

Using the Visual Studio command prompt (ensures the environment is setup properly), run the following in the top level directory:

```
nmake -f makefile.vc
```

and

```
nmake -f makefile.vc devinstall
```

11.2.8 PostGIS

Get PostGIS and the Windows version of PostgreSQL from here:

http://postgis.refractions.net/download/

Note the warning about not installing the version of PostGIS that comes with the PostgreSQL installer. Simply run the installers.

11.2.9 Expat

Get expat from here:

http://sourceforge.net/project/showfiles.php?group_id=10127

You'll need expat-win32bin-2.0.1.exe.

Simply run the executable to install expat.

11.2.10 CMake

Get CMake from here:

http://www.cmake.org/HTML/Download.html

You'll need cmake-<version>-win32-x86.exe. Simply run this to install CMake.

11.3 Building QGIS with CMAKE

Get QGIS source from svn (svn co https://svn.osgeo.org/qgis/trunk/qgis qgis).

Create a 'Build' directory in the top level QGIS directory. This will be where all the build output will be generated.

Run Start–>All Programs–>CMake–>CMake.

In the 'Where is the source code:' box, browse to the top level QGIS directory.

In the 'Where to build the binaries:' box, browse to the 'Build' directory you created in the top level QGIS directory.

Fill in the various *_INCLUDE_DIR and *_LIBRARY entries in the 'Cache Values' list.

Click the Configure button. You will be prompted for the type of makefile that will be generated. Select Visual Studio 8 2005 and click OK.

All being well, configuration should complete without errors. If there are errors, it is usually due to an incorrect path to a header or library directory. Failed items will be shown in red in the list.

Once configuration completes without error, click OK to generate the solution and project files.

With Visual Studio 2005, open the qgis.sln file that will have been created in the Build directory you created earlier.

Build the ALL_BUILD project. This will build all the QGIS binaries along with all the plugins.

Install QGIS by building the INSTALL project. By default this will install to c:\Program Files\qgis<version> (this can be changed by changing the CMAKE_INSTALL_PREFIX variable in CMake).

You will also either need to add all the dependency dlls to the QGIS install directory or add their respective directories to your PATH.

12 Building under Windows using MSVC Express

/!\ Note: Building under MSVC is still a work in progress. In particular the following dont work yet: python, grass, postgis connections.

/!\ This section of the document is in draft form and is not ready to be used yet.

Tim Sutton, 2007

12.1 System preparation

I started with a clean XP install with Service Pack 2 and all patches applied. I have already compiled all the dependencies you need for gdal, expat etc, so this tutorial wont cover compiling those from source too. Since compiling these dependencies was a somewhat painful task I hope my precompiled libs will be adequate. If not I suggest you consult the individual projects for specific build documentation and support. Lets go over the process in a nutshell before we begin:

* Install XP (I used a Parallels virtual machine) * Install the premade libraries archive I have made for you * Install Visual Studio Express 2005 sp1 * Install the Microsoft Platform SDK * Install command line subversion client * Install library dependencies bundle * Install Qt 4.3.2 * Check out QGIS sources * Compile QGIS * Create setup.exe installer for QGIS

12.2 Install the libraries archive

Half of the point of this section of the MSVC setup procedure is to make things as simple as possible for you. To that end I have prepared an archive that includes all dependencies needed to build QGIS except Qt (which we will build further down). Fetch the archive from:

```
http://qgis.org/uploadfiles/msvc/qgis_msvc_deps_except_qt4.zip
```

Create the following directory structure:

```
c:\dev\cpp\
```

And then extract the libraries archive into a subdirectory of the above directory so that you end up with:

```
c:\dev\cpp\qgislibs-release
```

/!\ Note that you are not obliged to use this directory layout, but you should adjust any instructions that follow if you plan to do things differently.

12.3 Install Visual Studio Express 2005

First thing we need to get is MSVC Express from here:

http://msdn2.microsoft.com/en-us/express/aa975050.aspx

The page is really confusing so dont feel bad if you cant actually find the download at first! There are six coloured blocks on the page for the various studio family members (vb / c# / j# etc). Simply choose your language under the 'select your language' combo under the yellow C++ block, and your download will begin. Under internet explorer I had to disable popup blocking for the download to be able to commence.

Once the setup commences you will be prompted with various options. Here is what I chose :

* Send useage information to Microsoft (No) * Install options: * Graphical IDE (Yes) * Microsoft MSDN Express Edition (No) * Microsoft SQL Server Express Edition (No) * Install to folder: C:\Program Files\Microsoft Visual Studio 8\ (default)

It will need to download around 90mb of installation files and reports that the install will consume 554mb of disk space.

12.4 Install Microsoft Platform SDK2

Go to this page:

http://msdn2.microsoft.com/en-us/express/aa700755.aspx

Start by using the link provided on the above page to download and install the platform SDK2.

The actual SDK download page is once again a bit confusing since the links for downloading are hidden amongst a bunch of other links. Basically look for these three links with their associated 'Download' buttons and choose the correct link for your platform:

```
PSDK-amd64.exe  1.2 MB  Download
PSDK-ia64.exe   1.3 MB  Download
PSDK-x86.exe    1.2 MB  Download
```

When you install make sure to choose 'custom install'. These instructions assume you are installing into the default path of:

```
C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\
```

We will go for the minimal install that will give us a working environment, so on the custom installation screen I made the following choices:

```
Configuration Options
  + Register Environmental Variables            (Yes)
Microsoft Windows Core SDK
  + Tools                                       (Yes)
    + Tools (AMD 64 Bit)                        (No unless this applies)
    + Tools (Intel 64 Bit)                      (No unless this applies)
  + Build Environment
    + Build Environment (AMD 64 Bit)            (No unless this applies)
    + Build Environment (Intel 64 Bit)          (No unless this applies)
    + Build Environment (x86 32 Bit)            (Yes)
  + Documentation                               (No)
  + Redistributable Components                  (Yes)
  + Sample Code                                 (No)
  + Source Code                                 (No)
    + AMD 64 Source                             (No)
    + Intel 64 Source                           (No)
Microsoft Web Workshop                          (Yes) (needed for shlwapi.h)
```

```
  + Build Environment                          (Yes)
  + Documentation                              (No)
  + Sample Code                                (No)
  + Tools                                      (No)
Microsoft Internet Information Server (IIS) SDK (No)
Microsoft Data Access Services (MDAC) SDK      (Yes) (needed by GDAL for odbc)
  + Tools
    + Tools (AMD 64 Bit)                       (No)
    + Tools (AMD 64 Bit)                       (No)
    + Tools (x86 32 Bit)                       (Yes)
  + Build Environment
    + Tools (AMD 64 Bit)                       (No)
    + Tools (AMD 64 Bit)                       (No)
    + Tools (x86 32 Bit)                       (Yes)
  + Documentation                              (No)
  + Sample Code                                (No)
Microsodt Installer SDK                        (No)
Microsoft Table PC SDK                         (No)
Microsoft Windows Management Instrumentation   (No)
Microsoft DirectShow SDK                       (No)
Microsoft Media Services SDK                   (No)
Debuggin Tools for Windows                     (Yes)
```

/!\ Note that you can always come back later to add extra bits if you like.

/!\ Note that installing the SDK requires validation with the Microsoft Genuine Advantage application. Some people have a philosophical objection to installing this software on their computers. If you are one of them you should probably consider using the MINGW build instructions described elsewhere in this document.

The SDK installs a directory called

```
C:\Office10
```

Which you can safely remove.

After the SDK is installed, follow the remaining notes on the page link above to get your MSVC Express environment configured correctly. For your convenience, these are summarised again below, and I have added a couple more paths that I discovered were needed:

1) open Visual Studio Express IDE

2) Tools -> Options -> Projects and Solutions -> VC++ Directories

3) Add:

```
Executable files:
  C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Bin

Include files:
  C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Include
  C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Include\atl
  C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Include\mfc
Library files: C:\Program Files\Microsoft Platform SDK for Windows Server 2003 R2\Lib
```

4) Close MSVC Express IDE

5) Open the following file with notepad:

```
C:\Program Files\Microsoft Visual Studio 8\VC\VCProjectDefaults\corewin_express.vsprops
```

and change the property:

```
AdditionalDependencies="kernel32.lib"
```

To read:

```
AdditionalDependencies="kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib
                        advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib"
```

The notes go on to show how to build a mswin32 application which you can try if you like - I'm not going to recover that here.

12.5 Edit your vsvars

Backup your vsvars32.bat file in

```
C:\Program Files\Microsoft Visual Studio 8\Common7\Tools
```

and replace it with this one:

```
@SET VSINSTALLDIR=C:\Program Files\Microsoft Visual Studio 8
@SET VCINSTALLDIR=C:\Program Files\Microsoft Visual Studio 8\VC
@SET FrameworkDir=C:\WINDOWS\Microsoft.NET\Framework
@SET FrameworkVersion=v2.0.50727
@SET FrameworkSDKDir=C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0
@if "%VSINSTALLDIR%"=="" goto error_no_VSINSTALLDIR
@if "%VCINSTALLDIR%"=="" goto error_no_VCINSTALLDIR

@echo Setting environment for using Microsoft Visual Studio 2005 x86 tools.

@rem
@rem Root of Visual Studio IDE installed files.
@rem
@set DevEnvDir=C:\Program Files\Microsoft Visual Studio 8\Common7\IDE

@set PATH=C:\Program Files\Microsoft Visual Studio 8\Common7\IDE;C:\Program \
Files\Microsoft Visual Studio 8\VC\BIN;C:\Program Files\Microsoft \
Visual Studio 8\Common7\Tools;C:\Program Files\Microsoft Visual Studio \
8\SDK\v2.0\bin;C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727;C:\Program \
Files\Microsoft Visual Studio 8\VC\VCPackages;%PATH%
@rem added by Tim
@set PATH=C:\Program Files\Microsoft Platform SDK for Windows Server 2003 \
R2\Bin;%PATH%
@set INCLUDE=C:\Program Files\Microsoft Visual Studio 8\VC\INCLUDE;%INCLUDE%
@rem added by Tim
@set INCLUDE=C:\Program Files\Microsoft Platform SDK for Windows Server 2003 \
R2\Include;%INCLUDE%
@set INCLUDE=C:\Program Files\Microsoft Platform SDK for Windows Server 2003 \
R2\Include\mfc;%INCLUDE%
@set INCLUDE=%INCLUDE%;C:\dev\cpp\qgislibs-release\include\postgresql
@set LIB=C:\Program Files\Microsoft Visual Studio 8\VC\LIB;C:\Program \
Files\Microsoft Visual Studio 8\SDK\v2.0\lib;%LIB%
@rem added by Tim
@set LIB=C:\Program Files\Microsoft Platform SDK for Windows Server 2003 \
R2\Lib;%LIB%
@set LIB=%LIB%;C:\dev\cpp\qgislibs-release\lib
@set LIBPATH=C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727

@goto end

:error_no_VSINSTALLDIR
@echo ERROR: VSINSTALLDIR variable is not set.
```

```
@goto end

:error_no_VCINSTALLDIR
@echo ERROR: VCINSTALLDIR variable is not set.
@goto end

:end
```

12.6 Environment Variables

Right click on 'My computer' then select the 'Advanced' tab. Click environment variables and create or augment the following "'System"' variables (if they dont already exist):

```
Variable Name:     Value:
-------------------------------------------------------------------------
EDITOR             vim
INCLUDE            C:\Program Files\Microsoft Platform SDK for Windows Server \
                   2003 R2\Include\.
LIB                C:\Program Files\Microsoft Platform SDK for Windows Server \
                   2003 R2\Lib\.
LIB_DIR            C:\dev\cpp\qgislibs-release
PATH               C:\Program Files\CMake 2.4\bin;
                   %SystemRoot%\system32;
                   %SystemRoot%;
                   %SystemRoot%\System32\Wbem;
                   C:\Program Files\Microsoft Platform SDK for Windows Server \
                   2003 R2\Bin\.;
                   C:\Program Files\Microsoft Platform SDK for Windows Server \
                   2003 R2\Bin\WinNT\;
                   C:\Program Files\svn\bin;C:\Program Files\Microsoft Visual \
                   Studio 8\VC\bin;
                   C:\Program Files\Microsoft Visual Studio 8\Common7\IDE;
                   "c:\Program Files\Microsoft Visual Studio 8\Common7\Tools";
                   c:\Qt\4.3.2\bin;
                   "C:\Program Files\PuTTY"
QTDIR              c:\Qt\4.3.2
SVN_SSH            "C:\\Program Files\\PuTTY\\plink.exe"
```

12.7 Building Qt4.3.2

You need a minimum of Qt 4.3.2 here since this is the first version to officially support building the open source version of Qt for windows under MSVC.

Download Qt 4.x.x source for windows from

```
http:\\www.trolltech.com
```

Unpack the source to

```
c:\Qt\4.x.x\
```

12.7.1 Compile Qt

Open the Visual Studio C++ command line and cd to c:\Qt\4.x.x where you extracted the source and enter:

```
configure -platform win32-msvc2005
nmake
nmake install
```

Add -qt-sql-odbc -qt-sql-psql to the configure line if your want odbc and PostgreSQL support build into Qt.

/!\ Note: For me in some cases I got a build error on qscreenshot.pro. If you are only interested in having the libraries needed for building Qt apps, you can probably ignore that. Just check in c:\Qt\4.3.2\bin to check all dlls and helper apps (assistant etc) have been made.

12.7.2 Configure Visual C++ to use Qt

After building configure the Visual Studio Express IDE to use Qt:

1) open Visual Studio Express IDE

2) Tools -> Options -> Projects and Solutions -> VC++ Directories

3) Add:

```
Executable files:
```

```
  $(QTDIR)\bin

Include files:
  $(QTDIR)\include
  $(QTDIR)\include\Qt
  $(QTDIR)\include\QtCore
  $(QTDIR)\include\QtGui
  $(QTDIR)\include\QtNetwork
  $(QTDIR)\include\QtSvg
  $(QTDIR)\include\QtXml
  $(QTDIR)\include\Qt3Support
  $(LIB_DIR)\include   (needed during qgis compile to find stdint.h and unistd.h)

Library files:
  $(QTDIR)\lib

Source Files:
  $(QTDIR)\src
```

Hint: You can also add

```
QString = t=<d->data, su>, size=<d->size, i>
```

to AutoExp.DAT in C:\Program Files\Microsoft Visual Studio 8\Common7\Packages\Debugger before

```
[Visualizer]
```

That way the Debugger will show the contents of QString when you point at or watch a variable in the debugger. There are probably much more additions - feel free to add some - I just needed QString and took the first hit in google I could find.

12.8 Install Python

Download http://python.org/ftp/python/2.5.1/python-2.5.1.msi and install it.

12.9 Install SIP

Download http://www.riverbankcomputing.com/Downloads/sip4/sip-4.7.1.zip and extract it into your c:\dev\cpp directory. From a Visual C++ command line cd to the directory where you extract SIP and

run:

```
c:\python25\python configure.py -p win32-msvc2005
nmake
nmake install
```

12.10 Install PyQt4

Download http://www.riverbankcomputing.com/Downloads/PyQt4/GPL/PyQt-win-gpl-4.3.1.zip and extract it into your c:\dev\cpp directory. From a Visual C++ command line cd to the directory where you extracted PyQt4 and run:

```
c:\python25\python configure.py -p win32-msvc2005
nmake
nmake install
```

12.11 Install CMake

Download and install cmake 2.4.7 or better, making sure to enable the option

```
Update path for all users
```

12.12 Install Subversion

You "'must"' install the command line version if you want the CMake svn scripts to work. Its a bit tricky to find the correct version on the subversion download site as they have som misleadingly named similar downloads. Easiest is to just get this file:

http://subversion.tigris.org/downloads/1.4.5-win32/apache-2.2/svn-win32-1.4.5.zip

Extract the zip file to

```
C:\Program Files\svn
```

And then add

```
C:\Program Files\svn\bin
```

To your path.

12.13 Initial SVN Check out

Open a cmd.exe window and do:

```
cd \
cd dev
cd cpp
svn co https://svn.osgeo.org/qgis/trunk/qgis
```

At this point you will probably get a message like this:

```
C:\dev\cpp>svn co https://svn.osgeo.org/qgis/trunk/qgis
Error validating server certificate for 'https://svn.qgis.org:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: svn.qgis.org
 - Valid: from Sat, 01 Apr 2006 03:30:47 GMT until Fri, 21 Mar 2008 03:30:47 GMT
 - Issuer: Developer Team, Quantum GIS, Anchorage, Alaska, US
 - Fingerprint: 2f:cd:f1:5a:c7:64:da:2b:d1:34:a5:20:c6:15:67:28:33:ea:7a:9b
(R)eject, accept (t)emporarily or accept (p)ermanently?
```

Press 'p' to accept and the svn checkout will commence.

12.14 Create Makefiles using cmakesetup.exe

I wont be giving a detailed description of the build process, because the process is explained in the first section (where you manually build all dependencies) of the windows build notes in this document. Just skip past the parts where you need to build GDAL etc, since this simplified install process does all the dependency provisioning for you.

```
cd qgis
mkdir build
cd build
cmakesetup ..
```

Cmakesetup should find all dependencies for you automatically (it uses the LIB_DIR environment to find them all in c:\dev\cpp\qgislibs-release). Press configure again after the cmakesetup gui appears

and when all the red fields are gone, and you have made any personalisations to the setup, press ok to close the cmake gui.

Now open Visual Studio Express and do:

File -> Open -> Project / Solution

Now open the cmake generated QGIS solution which should be in :

```
c:\dev\cpp\qgis\build\qgisX.X.X.sln
```

Where X.X.X represents the current version number of QGIS. Currently I have only made release built dependencies for QGIS (debug versions will follow in future), so you need to be sure to select 'Release' from the solution configurations toolbar.

Next right click on ALL_BUILD in the solution browser, and then choose build.

Once the build completes right click on INSTALL in the solution browser and choose build. This will by default install qgis into c:\program files\qgisX.X.X.

12.15 Running and packaging

To run QGIS you need to at the minimum copy the dlls from c:\dev\cpp\qgislibs-release\bin into the c:\program files\qgisX.X.X directory.

13 QGIS Coding Standards

The following chapters provide coding information for QGIS Version 1.1. This document corresponds almost to a LaTeX conversion of the CODING.t2t file coming with the QGIS sources from April, 24th 2009.

These standards should be followed by all QGIS developers. Current information about QGIS Coding Standards are also available from wiki at:

http://wiki.qgis.org/qgiswiki/CodingGuidelines
http://wiki.qgis.org/qgiswiki/CodingStandards
http://wiki.qgis.org/qgiswiki/UsingSubversion
http://wiki.qgis.org/qgiswiki/DebuggingPlugins
http://wiki.qgis.org/qgiswiki/DevelopmentInBranches
http://wiki.qgis.org/qgiswiki/SubmittingPatchesAndSvnAccess

13.1 Classes

13.1.1 Names

Class in QGIS begin with Qgs and are formed using mixed case.

```
Examples:
QgsPoint
QgsMapCanvas
QgsRasterLayer
```

13.1.2 Members

Class member names begin with a lower case *m* and are formed using mixed case.

```
mMapCanvas
mCurrentExtent
```

All class members should be private. **Public class members are STRONGLY discouraged**

13.1.3 Accessor Functions

Class member values should be obtained through accesssor functions. The function should be named without a *get* prefix. Accessor functions for the two private members above would be:

```
mapCanvas()
currentExtent()
```

13.1.4 Functions

Function names begin with a lowercase letter and are formed using mixed case. The function name should convey something about the purpose of the function.

```
updateMapExtent()
setUserOptions()
```

13.2 Qt Designer

13.2.1 Generated Classes

QGIS classes that are generated from Qt Designer (ui) files should have a *Base* suffix. This identifies the class as a generated base class.

```
Examples:
QgsPluginMangerBase
QgsUserOptionsBase
```

13.2.2 Dialogs

All dialogs should implement the following: * Tooltip help for all toolbar icons and other relevant widgets * WhatsThis help for **all** widgets on the dialog * An optional (though highly recommended) context sensitive *Help* button that directs the user to the appropriate help page by launching their web browser

13.3 C++ Files

13.3.1 Names

C++ implementation and header files should be have a .cpp and .h extension respectively. Filename should be all lowercase and, in the case of classes, match the class name.

```
Example:
Class QgsFeatureAttribute source files are
qgsfeatureattribute.cpp and qgsfeatureattribute.h
```

13.3.2 Standard Header and License

Each source file should contain a header section patterned after the following example:

```
/***************************************************************************
    qgsfield.cpp - Describes a field in a layer or table
     --------------------------------------
    Date                 : 01-Jan-2004
    Copyright            : (C) 2004 by Gary E.Sherman
    Email                : sherman at mrcc.com
/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
```

13.3.3 CVS Keyword

Each source file should contain the Id keyword. This will be expanded by CVS to contain useful information about the file, revision, last committer, and date/time of last checkin.

Place the keyword right after the standard header/license that is found at the top of each source file:

```
/* $Id$ */
```

13.4 Variable Names

Variable names begin with a lower case letter and are formed using mixed case.

```
Examples:
mapCanvas
currentExtent
```

13.5 Enumerated Types

Enumerated types should be named in CamelCase with a leading capital e.g.:

```
    enum UnitType
    {
      Meters,
      Feet,
      Degrees,
      UnknownUnit
    } ;
```

Do not use generic type names that will conflict with other types. e.g. use "UnkownUnit" rather than "Unknown"

13.6 Global Constants

Global constants should be written in upper case underscore separated e.g.:

```
const long GEOCRS_ID = 3344;
```

13.7 Editing

Any text editor/IDE can be used to edit QGIS code, providing the following requirements are met.

13.7.1 Tabs

Set your editor to emulate tabs with spaces. Tab spacing should be set to 2 spaces.

13.7.2 Indentation

Source code should be indented to improve readability. There is a .indent.pro file in the QGIS src directory that contains the switches to be used when indenting code using the GNU indent program. If you don't use GNU indent, you should emulate these settings.

13.7.3 Braces

Braces should start on the line following the expression:

```
if(foo == 1)
{
  // do stuff
  ...
  }else
{
  // do something else
  ...
}
```

13.8 API Compatibility

From QGIS 1.0 we will provide a stable, backwards compatible API. This will provide a stable basis for people to develop against, knowing their code will work against any of the 1.x QGIS releases (although recompiling may be required).Cleanups to the API should be done in a manner similar to the Trolltech developers e.g.

```
class Foo
{
  public:
    /** This method will be deprecated, you are encouraged to use
      doSomethingBetter() rather.
      @see doSomethingBetter()
     */
    bool doSomething();

    /** Does something a better way.
      @note This method was introduced in QGIS version 1.1
     */
```

```
    bool doSomethingBetter();

}
```

13.9 Coding Style

Here are described some programming hints and tips that will hopefully reduce errors, development time, and maintenance.

13.9.1 Where-ever Possible Generalize Code

`If you are cut-n-pasting code, or otherwise writing the same thing more than once, consider consolidating the code into a single function.`

This will:

- allow changes to be made in one location instead of in multiple places
- help prevent code bloat
- make it more difficult for multiple copies to evolve differences over time, thus making it harder to understand and maintain for others

13.9.2 Prefer Having Constants First in Predicates

Prefer to put constants first in predicates.

```
"0 == value" instead of "value == 0"
```

This will help prevent programmers from accidentally using "=" when they meant to use "==", which can introduce very subtle logic bugs. The compiler will generate an error if you accidentally use "=" instead of "==" for comparisons since constants inherently cannot be assigned values.

13.9.3 Whitespace Can Be Your Friend

Adding spaces between operators, statements, and functions makes it easier for humans to parse code.

Which is easier to read, this:

```
if (!a&&b)
```

or this:

```
if ( ! a && b )
```

13.9.4 Add Trailing Identifying Comments

Adding comments at the end of function, struct and class implementations makes it easier to find them later.

Consider that you're at the bottom of a source file and need to find a very long function – without these kinds of trailing comments you will have to page up past the body of the function to find its name. Of course this is ok if you wanted to find the beginning of the function; but what if you were interested at code near its end? You'd have to page up and then back down again to the desired part.

E.g.,

```
void foo::bar()
{
    // ... imagine a lot of code here
 } // foo::bar()
```

13.9.5 Use Braces Even for Single Line Statements

Using braces for code in if/then blocks or similar code structures even for single line statements means that adding another statement is less likely to generate broken code.

Consider:

```
  if (foo)
    bar();
 else
    baz();
```

Adding code after bar() or baz() without adding enclosing braces would create broken code. Though most programmers would naturally do that, some may forget to do so in haste.

So, prefer this:

```
if (foo)
{
  bar();
}
else
{
   baz();
}
```

13.9.6 Book recommendations

- Effective C++, Scott Meyers
- More Effective C++, Scott Meyers
- Effective STL, Scott Meyers
- Design Patterns, GoF

You should also really read this article from Qt Quarterly on http://doc.trolltech.com/qq/qq13-apis.html designing Qt style

14 SVN Access

This page describes how to get started using the QGIS Subversion repository

14.1 Accessing the Repository

To check out QGIS HEAD:

```
svn --username [your user name] co https://svn.osgeo.org/qgis/trunk/qgis
```

14.2 Anonymous Access

You can use the following commands to perform an anonymous checkout from the QGIS Subversion repository. Note we recommend checking out the trunk (unless you are a developer or really HAVE to have the latest changes and dont mind lots of crashing!).

You must have a subversion client installed prior to checking out the code. See the Subversion website for more information. The Links page contains a good selection of SVN clients for various platforms.

To check out a branch

```
svn co https://svn.osgeo.org/qgis/branches/<branch name>
```

To check out SVN stable trunk:

```
svn co https://svn.osgeo.org/qgis/trunk/qgis qgis_trunk
```

Note: If you are behind a proxy server, edit your ~/subversion/servers file to specify your proxy settings first!

Note: In QGIS we keep our most stable code in trunk. Periodically we will tag a release off trunk, and then continue stabilisation and selective incorporation of new features into trunk.

See the INSTALL file in the source tree for specific instructions on building development versions.

14.3 QGIS documentation sources

If you're interested in checking out Quantum GIS documentation sources:

```
svn co https://svn.osgeo.org/qgis/docs/trunk qgis_docs
```

You can also take a look at DocumentationWritersCorner for more information.

14.4 Documentation

The repository is organized as follows:

http://wiki.qgis.org/images/repo.png

See the Subversion book http://svnbook.red-bean.com for information on becoming a SVN master.

14.5 Development in branches

14.5.1 Purpose

The complexity of the QGIS source code has increased considerably during the last years. Therefore it is hard to anticipate the side effects that the addition of a feature will have. In the past, the QGIS

project had very long release cycles because it was a lot of work to reetablish the stability of the software system after new features were added. To overcome these problems, QGIS switched to a development model where new features are coded in svn branches first and merged to trunk (the main branch) when they are finished and stable. This section describes the procedure for branching and merging in the QGIS project.

14.5.2 Procedure

- Initial announcement on mailing list Before starting, make an announcement on the developer mailing list to see if another developer is already working on the same feature. Also contact the technical advisor of the project steering committee (PSC). If the new feature requires any changes to the QGIS architecture, a request for comment (RFC) is needed.
- Create a branch Create a new svn branch for the development of the new feature (see Using-Subversion for the svn syntax). Now you can start developing.
- Merge from trunk regularly It is recommended to merge the changes in trunk to the branch on a regular basis. This makes it easier to merge the branch back to trunk later.
- Documentation on wiki It is also recommended to document the intended changes and the current status of the work on a wiki page.
- Testing before merging back to trunk

When you are finished with the new feature and happy with the stability, make an announcement on the developer list. Before merging back, the changes will be tested by developers and users. Binary packages (especially for OsX and Windows) will be generated to also involve non-developers. In trac, a new Component will be opened to file tickets against. Once there are no remaining issues left, the technical advisor of the PSC merges the changes into trunk.

14.5.3 Creating a branch

We prefer that new feature developments happen out of trunk so that trunk remains in a stable state. To create a branch use the following command:

```
svn copy https://svn.osgeo.org/qgis/trunk/qgis \
https://svn.osgeo.org/qgis/branches/qgis_newfeature
svn commit -m "New feature branch"
```

14.5.4 Merge regularly from trunk to branch

When working in a branch you should regularly merge trunk into it so that your branch does not diverge more than necessary. In the top level dir of your branch, first type ‘`svn info`‘ to determine

the revision numbers of your branch which will produce output something like this:

```
timlinux@timlinux-desktop:~/dev/cpp/qgis_raster_transparency_branch$ svn info
Caminho: .
URL: https://svn.osgeo.org/qgis/branches/raster_transparency_branch
Raiz do Repository: https://svn.osgeo.org/qgis
UUID do repository: c8812cc2-4d05-0410-92ff-de0c093fc19c
Revision: 6546
Tipo de No: diretorio
Agendado: normal
Autor da Ultima Mudanca: timlinux
Revision da Ultima Mudanca: 6495
Data da Ultima Mudanca: 2007-02-02 09:29:47 -0200 (Sex, 02 Fev 2007)
Propriedades da Ultima Mudanca: 2007-01-09 11:32:55 -0200 (Ter, 09 Jan 2007)
```

The second revision number shows the revision number of the start revision of your branch and the first the current revision. You can do a dry run of the merge like this:

```
svn merge --dry-run -r 6495:6546 https://svn.osgeo.org/qgis/trunk/qgis
```

After you are happy with the changes that will be made do the merge for real like this:

```
svn merge -r 6495:6546 https://svn.osgeo.org/qgis/trunk/qgis
svn commit -m "Merged upstream changes from trunk to my branch"
```

14.6 Submitting Patches

There are a few guidelines that will help you to get your patches into QGIS easily, and help us deal with the patches that are sent to use easily.

14.6.1 Patch file naming

If the patch is a fix for a specific bug, please name the file with the bug number in it e.g. **bug777fix.diff**, and attach it to the original bug report in trac (https://trac.osgeo.org/qgis/).

If the bug is an enhancement or new feature, its usually a good idea to create a ticket in trac (https://trac.osgeo.org/qgis/) first and then attach you

14.6.2 Create your patch in the top level QGIS source dir

This makes it easier for us to apply the patches since we don't need to navigate to a specific place in the source tree to apply the patch. Also when I receive patches I usually evaluate them using kompare, and having the patch from the top level dir makes this much easier. Below is an example of you you can include multiple changed files into your patch from the top level directory:

```
cd qgis
svn diff src/ui/somefile.ui src/app/somefile2.cpp > bug872fix.diff
```

14.6.3 Including non version controlled files in your patch

If your improvements include new files that don't yet exist in the repository, you should indicate to svn that they need to be added before generating your patch e.g.

```
cd qgis
svn add src/lib/somenewfile.cpp
svn diff > bug7887fix.diff
```

14.6.4 Getting your patch noticed

QGIS developers are busy folk. We do scan the incoming patches on bug reports but sometimes we miss things. Don't be offended or alarmed. Try to identify a developer to help you - using the ["Project Organigram"] and contact them asking them if they can look at your patch. If you dont get any response, you can escalate your query to one of the Project Steering Committee members (contact details also available on the ["Project Organigram"]).

14.6.5 Due Diligence

QGIS is licensed under the GPL. You should make every effort to ensure you only submit patches which are unencumbered by conflicting intellectual property rights. Also do not submit code that you are not happy to have made available under the GPL.

14.7 Obtaining SVN Write Access

Write access to QGIS source tree is by invitation. Typically when a person submits several (there is no fixed number here) substantial patches that demonstrate basic competance and understanding

of C++ and QGIS coding conventions, one of the PSC members or other existing developers can nominate that person to the PSC for granting of write access. The nominator should give a basic promotional paragraph of why they think that person should gain write access. In some cases we will grant write access to non C++ developers e.g. for translators and documentors. In these cases, the person should still have demonstrated ability to submit patches and should ideally have submtted several substantial patches that demonstrate their understanding of modifying the code base without breaking things, etc.

14.7.1 Procedure once you have access

Checkout the sources:

```
svn co https://svn.osgeo.org/qgis/trunk/qgis qgis
```

Build the sources (see INSTALL document for proper detailed instructions)

```
cd qgis
mkdir build
ccmake ..    (set your preferred options)
make
make install  (maybe you need to do with sudo / root perms)
```

Make your edits

```
cd ..
```

Make your changes in sources. Always check that everything compiles before making any commits. Try to be aware of possible breakages your commits may cause for people building on other platforms and with older / newer versions of libraries.

Add files (if you added any new files). The svn status command can be used to quickly see if you have added new files.

```
svn status src/pluguns/grass/modules
```

Files listed with ? in front are not in SVN and possibly need to be added by you:

```
svn add src/pluguns/grass/modules/foo.xml
```

Commit your changes

```
svn commit src/pluguns/grass/modules/foo.xml
```

Your editor (as defined in $EDITOR environment variable) will appear and you should make a comment at the top of the file (above the area that says 'dont change this'. Put a descriptive comment and rather do several small commits if the changes across a number of files are unrelated. Conversely we prefer you to group related changes into a single commit.

Save and close in your editor. The first time you do this, you should be prompted to put in your username and password. Just use the same ones as your trac account.

15 Unit Testing

As of November 2007 we require all new features going into trunk to be accompanied with a unit test. Initially we have limited this requirement to qgis_core, and we will extend this requirement to other parts of the code base once people are familiar with the procedures for unit testing explained in the sections that follow.

15.1 The QGIS testing framework - an overview

Unit testing is carried out using a combination of QTestLib (the Qt testing library) and CTest (a framework for compiling and running tests as part of the CMake build process). Lets take an overview of the process before I delve into the details:

* **There is some code you want to test**, e.g. a class or function. Extreme programming advocates suggest that the code should not even be written yet when you start building your tests, and then as you implement your code you can immediately validate each new functional part you add with your test. In practive you will probably need to write tests for pre-existing code in QGIS since we are starting with a testing framework well after much application logic has already been implemented.

* **You create a unit test.** This happens under <QGIS Source Dir>/tests/src/core in the case of the core lib. The test is basically a client that creates an instance of a class and calls some methods on that class. It will check the return from each method to make sure it matches the expected value. If any one of the calls fails, the unit will fail.

* **You include QtTestLib macros in your test class.** This macro is processed by the Qt meta object compiler (moc) and expands your test class into a runnable application.

* **You add a section to the CMakeLists.txt** in your tests directory that will build your test.

* **You ensure you have ENABLE_TESTING enabled in ccmake / cmakesetup.** This will ensure your tests actually get compiled when you type make.

* **You optionally add test data to <QGIS Source Dir>/tests/testdata** if your test is data driven (e.g. needs to load a shapefile). These test data should be as small as possible and wherever possible you should use the existing datasets already there. Your tests should never modify this data in situ, but rather may a temporary copy somewhere if needed.

* **You compile your sources and install.** Do this using normal make && (sudo) make install procedure.

* **You run your tests.** This is normally done simply by doing **make test** after the make install step, though I will explain other aproaches that offer more fine grained control over running tests.

Right with that overview in mind, I will delve into a bit of detail. I've already done much of the configuration for you in CMake and other places in the source tree so all you need to do are the easy bits - writing unit tests!

15.2 Creating a unit test

Creating a unit test is easy - typically you will do this by just creating a single .cpp file (not .h file is used) and implement all your test methods as public methods that return void. I'll use a simple test class for QgsRasterLayer throughout the section that follows to illustrate. By convention we will name our test with the same name as the class they are testing but prefixed with 'Test'. So our test implementation goes in a file called testqgsrasterlayer.cpp and the class itself will be TestQgsRasterLayer. First we add our standard copyright banner:

```
/***************************************************************************
     testqgsvectorfilewriter.cpp
     ---------------------------------------
    Date                 : Frida  Nov 23  2007
    Copyright            : (C) 2007 by Tim Sutton
    Email                : tim@linfiniti.com
 ***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
```

Next we use start our includes needed for the tests we plan to run. There is one special include all tests should have:

```
#include <QtTest>
```

Beyond that you just continue implementing your class as per normal, pulling in whatever headers you may need:

```
//Qt includes...
#include <QObject>
#include <QString>
#include <QObject>
#include <QApplication>
#include <QFileInfo>
#include <QDir>

//qgis includes...
#include <qgsrasterlayer.h>
#include <qgsrasterbandstats.h>
#include <qgsapplication.h>
```

Since we are combining both class declaration and implementation in a single file the class declaration comes next. We start with our doxygen documentation. Every test case should be properly documented. We use the doxygen **ingroup** directive so that all the UnitTests appear as a module in the generated Doxygen documentation. After that comes a short description of the unit test:

```
/** \ingroup UnitTests
 * This is a unit test for the QgsRasterLayer class.
 */
```

The class **must** inherit from QObject and include the Q_OBJECT macro.

```
class TestQgsRasterLayer: public QObject
{
  Q_OBJECT;
```

All our test methods are implemented as **private slots**. The QtTest framework will sequentially call each private slot method in the test class. There are four 'special' methods which if implemented will be called at the start of the unit test (**initTestCase**), at the end of the unit test (**cleanupTestCase**).

Before each test method is called, the **init()** method will be called and after each test method is called the **cleanup()** method is called. These methods are handy in that they allow you to allocate and cleanup resources prior to running each test, and the test unit as a whole.

```
private slots:
    // will be called before the first testfunction is executed.
    void initTestCase();
    // will be called after the last testfunction was executed.
    void cleanupTestCase(){};
    // will be called before each testfunction is executed.
    void init(){};
    // will be called after every testfunction.
    void cleanup();
```

Then come your test methods, all of which should take **no parameters** and should **return void**. The methods will be called in order of declaration. I am implementing two methods here which illustrates to types of testing. In the first case I want to generally test the various parts of the class are working, I can use a **functional testing** approach. Once again, extreme programmers would advocate writing these tests **before** implementing the class. Then as you work your way through your class implementation you iteratively run your unit tests. More and more test functions should complete sucessfully as your class implementation work progresses, and when the whole unit test passes, your new class is done and is now complete with a repeatable way to validate it.

Typically your unit tests would only cover the **public** API of your class, and normally you do not need to write tests for accessors and mutators. If it should happen that an acccessor or mutator is not working as expected you would normally implement a **regression** test to check for this (see lower down).

```
    //
    // Functional Testing
    //

    /** Check if a raster is valid. */
    void isValid();

    // more functional tests here ...
```

Next we implement our **regression tests**. Regression tests should be implemented to replicate the conditions of a particular bug. For example I recently received a report by email that the cell count by rasters was off by 1, throwing off all the statistics for the raster bands. I opened a bug (ticket #832) and then created a regression test that replicated the bug using a small test dataset (a 10x10

raster). Then I ran the test and ran it, verifying that it did indeed fail (the cell count was 99 instead of 100). Then I went to fix the bug and reran the unit test and the regression test passed. I committed the regression test along with the bug fix. Now if anybody breakes this in the source code again in the future, we can immediatly identify that the code has regressed. Better yet before committing any changes in the future, running our tests will ensure our changes dont have unexpected side effects - like breaking existing functionality.

There is one more benifit to regression tests - they can save you time. If you ever fixed a bug that involved making changes to the source, and then running the application and performing a series of convoluted steps to replicate the issue, it will be immediately apparent that simply implementing your regression test **before** fixing the bug will let you automate the testing for bug resolution in an efficient manner.

To implement your regression test, you should follow the naming convention of regression<TicketID> for your test functions. If no trac ticket exists for the regression, you should create one first. Using this approach allows the person running a failed regression test easily go and find out more information.

```
    //
    // Regression Testing
    //

    /** This is our second test case...to check if a raster
     reports its dimensions properly. It is a regression test
     for ticket #832 which was fixed with change r7650.
     */
    void regression832();

    // more regression tests go here ...
```

Finally in our test class declaration you can declare privately any data members and helper methods your unit test may need. In our case I will declare a QgsRasterLayer * which can be used by any of our test methods. The raster layer will be created in the initTestCase() function which is run before any other tests, and then destroyed using cleanupTestCase() which is run after all tests. By declaring helper methods (which may be called by various test functions) privately, you can ensure that they wont be automatically run by the QTest executeable that is created when we compile our test.

```
  private:
    // Here we have any data structures that may need to
    // be used in many test cases.
    QgsRasterLayer * mpLayer;
};
```

That ends our class declaration. The implementation is simply inlined in the same file lower down. First our init and cleanup functions:

```
void TestQgsRasterLayer::initTestCase()
{
  // init QGIS's paths - true means that all path will be inited from prefix
  QString qgisPath = QCoreApplication::applicationDirPath ();
  QgsApplication::setPrefixPath(qgisPath, TRUE);
#ifdef Q_OS_LINUX
  QgsApplication::setPkgDataPath(qgisPath + "/../share/qgis");
#endif
  //create some objects that will be used in all tests...

  std::cout << "Prefix  PATH: " \
  << QgsApplication::prefixPath().toLocal8Bit().data() << std::endl;
  std::cout << "Plugin  PATH: " \
  << QgsApplication::pluginPath().toLocal8Bit().data() << std::endl;
  std::cout << "PkgData PATH: " \
  << QgsApplication::pkgDataPath().toLocal8Bit().data() << std::endl;
  std::cout << "User DB PATH: " \
  << QgsApplication::qgisUserDbFilePath().toLocal8Bit().data() << std::endl;

  //create a raster layer that will be used in all tests...
  QString myFileName (TEST_DATA_DIR); //defined in CmakeLists.txt
  myFileName = myFileName + QDir::separator() + "tenbytenraster.asc";
  QFileInfo myRasterFileInfo ( myFileName );
  mpLayer = new QgsRasterLayer ( myRasterFileInfo.filePath(),
            myRasterFileInfo.completeBaseName() );
}

void TestQgsRasterLayer::cleanupTestCase()
{
  delete mpLayer;
}
```

The above init function illustrates a couple of interesting things.

1. I needed to manually set the QGIS application data path so that resources such as srs.db can be found properly. 2. Secondly, this is a data driven test so we needed to provide a way to generically locate the 'tenbytenraster.asc file. This was achieved by using the compiler define **TEST_DATA_PATH**. The define is created in the CMakeLists.txt configuration file under <QGIS Source

Root>/tests/CMakeLists.txt and is available to all QGIS unit tests. If you need test data for your test, commit it under <QGIS Source Root>/tests/testdata. You should only commit very small datasets here. If your test needs to modify the test data, it should make a copy of if first.

Qt also provides some other interesting mechanisms for data driven testing, so if you are interested to know more on the topic, consult the Qt documentation.

Next lets look at our functional test. The isValid() test simply checks the raster layer was correctly loaded in the initTestCase. QVERIFY is a Qt macro that you can use to evaluate a test condition. There are a few other use macros Qt provide for use in your tests including:

```
QCOMPARE ( actual, expected )
QEXPECT_FAIL ( dataIndex, comment, mode )
QFAIL ( message )
QFETCH ( type, name )
QSKIP ( description, mode )
QTEST ( actual, testElement )
QTEST_APPLESS_MAIN ( TestClass )
QTEST_MAIN ( TestClass )
QTEST_NOOP_MAIN ()
QVERIFY2 ( condition, message )
QVERIFY ( condition )
QWARN ( message )
```

Some of these macros are useful only when using the Qt framework for data driven testing (see the Qt docs for more detail).

```
void TestQgsRasterLayer::isValid()
{
  QVERIFY ( mpLayer->isValid() );
}
```

Normally your functional tests would cover all the range of functionality of your classes public API where feasible. With our functional tests out the way, we can look at our regression test example.

Since the issue in bug #832 is a misreported cell count, writing our test if simply a matter of using QVERIFY to check that the cell count meets the expected value:

```
void TestQgsRasterLayer::regression832()
{
   QVERIFY ( mpLayer->getRasterXDim() == 10 );
   QVERIFY ( mpLayer->getRasterYDim() == 10 );
```

```
   // regression check for ticket #832
   // note getRasterBandStats call is base 1
   QVERIFY ( mpLayer->getRasterBandStats(1).elementCountInt == 100 );
}
```

With all the unit test functions implemented, there one final thing we need to add to our test class:

```
QTEST_MAIN(TestQgsRasterLayer)
#include "moc_testqgsrasterlayer.cxx"
```

The purpose of these two lines is to signal to Qt's moc that his is a QtTest (it will generate a main method that in turn calls each test funtion. The last line is the include for the MOC generated sources. You should replace 'testqgsrasterlayer' with the name of your class in lower case.

15.3 Adding your unit test to CMakeLists.txt

Adding your unit test to the build system is simply a matter of editing the CMakeLists.txt in the test directory, cloning one of the existing test blocks, and then search and replacing your test class name into it. For example:

```
#
# QgsRasterLayer test
#
SET(qgis_rasterlayertest_SRCS testqgsrasterlayer.cpp)
SET(qgis_rasterlayertest_MOC_CPPS testqgsrasterlayer.cpp)
QT4_WRAP_CPP(qgis_rasterlayertest_MOC_SRCS ${qgis_rasterlayertest_MOC_CPPS})
ADD_CUSTOM_TARGET(qgis_rasterlayertestmoc ALL DEPENDS ${qgis_rasterlayertest_MOC_SRCS})
ADD_EXECUTABLE(qgis_rasterlayertest ${qgis_rasterlayertest_SRCS})
ADD_DEPENDENCIES(qgis_rasterlayertest qgis_rasterlayertestmoc)
TARGET_LINK_LIBRARIES(qgis_rasterlayertest ${QT_LIBRARIES} qgis_core)
INSTALL(TARGETS qgis_rasterlayertest RUNTIME DESTINATION ${QGIS_BIN_DIR})
ADD_TEST(qgis_rasterlayertest ${QGIS_BIN_DIR}/qgis_rasterlayertest)
```

I'll run through these lines briefly to explain what they do, but if you are not interested, just clone the block, search and replace e.g.

```
:'<,'>s/rasterlayer/mynewtest/g
```

Lets look a little more in detail at the individual lines. First we define the list of sources for our test. Since we have only one source file (following the methodology I described above where class declaration and definition are in the same file) its a simple statement:

```
SET(qgis_rasterlayertest_SRCS testqgsrasterlayer.cpp)
```

Since our test class needs to be run through the Qt meta object compiler (moc) we need to provide a couple of lines to make that happen too:

```
SET(qgis_rasterlayertest_MOC_CPPS testqgsrasterlayer.cpp)
QT4_WRAP_CPP(qgis_rasterlayertest_MOC_SRCS ${qgis_rasterlayertest_MOC_CPPS})
ADD_CUSTOM_TARGET(qgis_rasterlayertestmoc ALL DEPENDS ${qgis_rasterlayertest_MOC_SRCS})
```

Next we tell cmake that it must make an executeable from the test class. Remember in the previous section on the last line of the class implementation I included the moc outputs directly into our test class, so that will give it (among other things) a main method so the class can be compiled as an executeable:

```
ADD_EXECUTABLE(qgis_rasterlayertest ${qgis_rasterlayertest_SRCS})
ADD_DEPENDENCIES(qgis_rasterlayertest qgis_rasterlayertestmoc)
```

Next we need to specify any library dependencies. At the moment classes have been implemented with a catch-all QT_LIBRARIES dependency, but I will be working to replace that with the specific Qt libraries that each class needs only. Of course you also need to link to the relevant qgis libraries as required by your unit test.

```
TARGET_LINK_LIBRARIES(qgis_rasterlayertest ${QT_LIBRARIES} qgis_core)
```

Next I tell cmake to the same place as the qgis binaries itself. This is something I plan to remove in the future so that the tests can run directly from inside the source tree.

```
INSTALL(TARGETS qgis_rasterlayertest RUNTIME DESTINATION ${QGIS_BIN_DIR})
```

Finally here is where the best magic happens - we register the class with ctest. If you recall in the overview I gave in the beginning of this section we are using both QtTest and CTest together. To recap, **QtTest** adds a main method to your test unit and handles calling your test methods within the class. It also provides some macros like QVERIFY that you can use as to test for failure of the tests using conditions. The output from a QtTest unit test is an executeable which you can run from the command line. However when you have a suite of tests and you want to run each executeable in turn, and better yet integrate running tests into the build process, the **CTest** is what we use. The next line registers the unit test with CMake / CTest.

```
ADD_TEST(qgis_rasterlayertest ${QGIS_BIN_DIR}/qgis_rasterlayertest)
```

The last thing I should add is that if your test requires optional parts of the build process (e.g. Postgresql support, GSL libs, GRASS etc.), you should take care to enclose you test block inside a IF () block in the CMakeLists.txt file.

15.4 Building your unit test

To build the unit test you need only to make sure that ENABLE_TESTS=true in the cmake configuration. There are two ways to do this:

1. Run ccmake .. (cmakesetup .. under windows) and interactively set the ENABLE_TESTS flag to ON. 1. Add a command line flag to cmake e.g. cmake -DENABLE_TESTS=true ..

Other than that, just build QGIS as per normal and the tests should build too.

15.5 Run your tests

The simplest way to run the tests is as part of your normal build process:

```
make && make install && make test
```

The make test command will invoke CTest which will run each test that was registered using the ADD_TEST CMake directive described above. Typical output from make test will look like this:

```
Running tests...
Start processing tests
Test project /Users/tim/dev/cpp/qgis/build
1/  3 Testing qgis_applicationtest          ***Exception: Other
2/  3 Testing qgis_filewritertest           *** Passed
3/  3 Testing qgis_rasterlayertest          *** Passed

0% tests passed, 3 tests failed out of 3

  The following tests FAILED:
  1 - qgis_applicationtest (OTHER_FAULT)
  Errors while running CTest
  make: *** [test] Error 8
```

If a test fails, you can use the ctest command to examine more closely why it failed. User the -R option to specify a regex for which tests you want to run and -V to get verbose output:

```
[build] ctest -R appl -V
Start processing tests
Test project /Users/tim/dev/cpp/qgis/build
Constructing a list of tests
Done constructing a list of tests
Changing directory into /Users/tim/dev/cpp/qgis/build/tests/src/core
1/  3 Testing qgis_applicationtest
Test command: /Users/tim/dev/cpp/qgis/build/tests/src/core/qgis_applicationtest
********* Start testing of TestQgsApplication *********
  Config: Using QTest library 4.3.0, Qt 4.3.0
PASS   : TestQgsApplication::initTestCase()
  Prefix  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
  Plugin  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//lib/qgis
  PkgData PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis
  User DB PATH: /Users/tim/.qgis/qgis.db
PASS   : TestQgsApplication::getPaths()
  Prefix  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/../
  Plugin  PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//lib/qgis
  PkgData PATH: /Users/tim/dev/cpp/qgis/build/tests/src/core/..//share/qgis
  User DB PATH: /Users/tim/.qgis/qgis.db
  QDEBUG : TestQgsApplication::checkTheme() Checking if a theme icon exists:
  QDEBUG : TestQgsApplication::checkTheme()
  /Users/tim/dev/cpp/qgis/build/tests/src/core/../\
  /share/qgis/themes/default//mIconProjectionDisabled.png
  FAIL!  : TestQgsApplication::checkTheme() '!myPixmap.isNull()' returned FALSE. ()
  Loc: [/Users/tim/dev/cpp/qgis/tests/src/core/testqgsapplication.cpp(59)]
PASS   : TestQgsApplication::cleanupTestCase()
  Totals: 3 passed, 1 failed, 0 skipped
  ********* Finished testing of TestQgsApplication *********
  -- Process completed
  ***Failed

  0% tests passed, 1 tests failed out of 1

  The following tests FAILED:
1 - qgis_applicationtest (Failed)
  Errors while running CTest
```

Well that concludes this section on writing unit tests in QGIS. We hope you will get into the habit of writing test to test new functionality and to check for regressions. Some aspects of the test system (in particular the CMakeLists.txt parts) are still being worked on so that the testing framework works

in a truly platform way. I will update this document as things progress.

16 HIG (Human Interface Guidelines)

In order for all graphical user interface elements to appear consistant and to all the user to instinctively use dialogs, it is important that the following guidelines are followed in layout and design of GUIs.

1. Group related elements using group boxes: Try to identify elements that can be grouped together and then use group boxes with a label to identify the topic of that group. Avoid using group boxes with only a single widget / item inside.
2. Capitalise first letter only in labels: Labels (and group box labels) should be written as a phrase with leading capital letter, and all remaing words written with lower case first letters
3. Do not end labels for widgets or group boxes with a colon: Adding a colon causes visual noise and does not impart additional meaning, so dont use them. An exception to this rule is when you have two labels next to each other e.g.: Label1 Plugin Label2 [/path/to/plugins]
4. Keep harmful actions away from harmless ones: If you have actions for 'delete', 'remove' etc, try to impose adequate space between the harmful action and innocuous actions so that the users is less likely to inadvertantly click on the harmful action.
5. Always use a QButtonBox for 'OK', 'Cancel' etc buttons: Using a button box will ensure that the order of 'OK' and 'Cancel' etc, buttons is consistent with the operating system / locale / desktop environment that the user is using.

17 GNU General Public License

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under

the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from

distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PRO-

GRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17.1 Quantum GIS Qt exception for GPL

In addition, as a special exception, the QGIS Development Team gives permission to link the code of this program with the Qt library, including but not limited to the following versions (both free and commercial): Qt/Non-commerical Windows, Qt/Windows, Qt/X11, Qt/Mac, and Qt/Embedded (or with modified versions of Qt that use the same license as Qt), and distribute linked combinations including the two. You must obey the GNU General Public License in all respects for all of the code used other than Qt. If you modify this file, you may extend this exception to your version of the file, but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version.

Literature

Web-References