# Pokročilé programování v jazyce C pro chemiky

(C3220)

# Knihovna Qt – část 1.

# Implicitní parametry funkce

- Pro funkce (a metody), které přijímají parametry, můžeme definovat implicitní hodnoty parametrů
- Funkci potom můžeme volat, jako kdyby byla bez parametrů, přičemž parametrům je přiřazena zadaná výchozí hodnota
- U metod se implicitní hodnoty udávají jen v deklaraci metody v těle třídy (bloku class), nikoli v samostatné definici metody mimo třídu

```
// Parametru funkce priradime implicitni hodnotu 10
void printValue(int val = 10)
{
  cout << "Hodnota: " << val << endl;
}

int main()
{
  int a = 3;
  // Pokud funkci zavolame bez parametru, pouzije se implicitni hodnota
  // v tomto pripade se tedy na obrazovku vypise hodnota 10
  printValue();

  // Pokud funkci parametr predame, pouzije se predana hodnota.
  // Nasledujici volani funkce zpusobi vypis hodnoty 7
  printValue(7);
  // Nasledujici volani funkce zpusobi vypis hodnoty v promenne a, tj. 3
  printValue(a);
  return 0;
}
```

# Implicitní parametry funkce

- Pokud má funkce více parametrů, můžeme definovat implicitní hodnoty jen pro některé parametry. Nejdříve uvádíme parametry bez implicitních hodnot a až potom parametry s implicitními hodnotami.
- Při volání funkce musíme povinně předat všechny parametry, které nemají specifikovány implicitní hodnoty. Parametry, které mají specifikovány implicitní hodnoty, předávat nemusíme (v takovém případě se pro chybějící parametry použijí jejich implicitní hodnoty).
- Volitelné parametry nelze “přeskakovat” (vynechat lze jen nějakou souvislou skupinu parametrů zprava)

```
// Některým parametrům funkce přiřadíme implicitní hodnoty
void printValue(int n1, int n2, int n3 = 5, int n4 = 10)
{
  cout << "Hodnoty: " << n1 << n2 << n3 << n4 << endl;
}

int main()
{
  int a = 3, b = 9, c = 1, d = 2;
  // Všechna následující volání jsou platná
  printValue(a, b);
  printValue(a, b, c);
  printValue(a, b, c, d);
  // Něco takového ale nelze: printValue(a, b, n4=d);
  return 0;
}
```

# Nepojmenované proměnné

- Někdy potřebujeme vytvořit dočasnou proměnnou pouze za účelem předání do volané funkce, v takovém případě ji můžeme definovat až v okamžiku předávání a nemusíme ji nijak pojmenovávat

```
// Tady bude definovana trida Vector3D, viz. predchozi prednasky

void printVector(const Vector3D &v)
{
  cout << "Vektor: " << v.getX() << v.getY() << v.getZ() << endl;
}

int main()
{
  // Do funkce printVector() predame nepojmenovanou promennou, ktera
  // bude vytvorena az v okamziku volani funkce; hodnotu teto
  // nepojmenovane promenne nastavime predanim prislusnych hodnot
  // do konstruktoru
  printVector(Vector3D(1.0, 2.0, 3.0));

  // Take muzeme udelat nasledujici: dve nepojmenovane promenne se
  // predaji do operatoru + a ten vrati hodnotu, ktera se
  // preda do printVector();
  printVector(Vector3D(4.0, 5.0, 4.0) + Vector3D(1.0, 2.0, 3.0));
  return 0;
}
```

# Grafické objektové knihovny

- Pro tvorbu grafických aplikací v C++ se používají objektové knihovny, které obsahují třídy pro práci s okny, interaktivními prvky (tlačítka, seznamy, editační políčka, ...) a podpůrné třídy
- Nejznámější knihovny:

  MFC (Microsoft Foundation Classes) – knihovna od firmy Microsoft pro tvorbu programů pro MS Windows, např. MS Office

  .NET Framework – novější rozsáhlá knihovna od firmy Microsoft pro tvorbu grafických i negrafických programů

  Qt – grafická multiplatformní knihovna dostupná pod svobodnou licencí

  GTK+ - open-source knihovna původně určená pro Unixové systémy, nyní dostupná i pro Windows
- Více informací: https://en.wikipedia.org/wiki/List_of_widget_toolkits

# Knihovna Qt

- Vlastnosti knihovny Qt (vysl. „cute“):
  - Dostupná jako svobodný software pod (L)GPL licencí (nebo alternativně pod proprietární licencí)
  - Multiplatformní, podporuje operační systémy Unix/Linux, MS Windows, macOS, Android, iOS, mikrokontroléry, …
  - Kvalitní implementace tříd pro tvorbu grafických rozhraní
  - Obsahuje také třídy pro práci s databázemi, soubory v XML formátu, síťové aplikace, prohlížení webu, multimédia, atd.
- Nejnovější verze je Qt6, zatím ale nejrozšířenější stále Qt5
- Více informací na https://www.qt.io/ a https://wiki.qt.io/Qt_for_Beginners
- Kompletní dokumentace ke Qt knihovně: https://doc.qt.io/qt-5/
- Kniha o Qt: C++ GUI Programming with Qt4
  - PDF volně ke stažení pod Open Publication License na https://www.topfreebooks.org/c-gui-programming-with-qt-4/

# Kompilace programu používajícího Qt

- Knihovna Qt vyžaduje složitější překlad, pro snadnější práci je s knihovnou dodáván program *qmake*, který slouží k automatickému generování souboru *Makefile*
- Každý program umisťujeme do samostatného adresáře, jehož jméno by mělo odpovídat jménu projektu
- V adresáři vytvoříme soubory **.cpp* se zdrojovým kódem programu
- V adresáři spustíme `qmake -project`, čímž vygenerujeme soubor projektu (má koncovku *.pro* a jméno je totožné s názvem adresáře)
- Potom vygenerujeme *Makefile* příkazem `qmake soubor.pro` kde soubor.pro je jméno souboru s projektem (vygenerovaný v předchozím kroku). Často stačí jen `qmake` (projekt najde sám).
- Kompilaci spouštíme příkazem `make` (bez parametrů)
- Při každé změně zdrojového kódu vždy spustíme kompilaci zavoláním příkazu *make* (tj. nemusíme již opakovat předcházející kroky s voláním *qmake*)

# Program v Qt

- Pro použití knihovny Qt je třeba vložit příslušné hlavičkové soubory, pro každou třídu existuje samostatný hlavičkový soubor, jehož název odpovídá názvu třídy
- Každý program používající knihovnu Qt musí vytvořit objekt třídy `QApplication` a zavolat její metodu `exec()`
- Konstruktoru třídy `QApplication` se předají parametry z příkazového řádku
- Metoda `exec()` obsahuje cyklus, který řídí celý program v závislosti na uživatelských vstupech (klávesnice, myš) a volá další objekty

```
#include <iostream>
#include <QApplication>
using namespace std;

int main(int argc, char *argv[])
{
  // Vytvorime objekt tridy QApplication a do konstruktoru predame
  // parametry prikazoveho radku
  QApplication app(argc, argv);
  // Zavolame metodu exec() ktera obsahuje cyklus udalosti, ktery
  // zpracovava vstup z klavesnice a mysi dokud neni program ukoncen
  return app.exec();
}
```

# Vytvoření okna v Qt

- Pro vytvoření jednoduchého okna v Qt se používá třída `QWidget`
- Po vytvoření objektu třídy `QWidget` se pro zobrazení okna musí zavolat metoda `show()`
- Titulek v záhlaví okna můžeme nastavit metodou `setWindowTitle()`

```
#include <iostream>
#include <QApplication>
#include <QWidget>
using namespace std;

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  // Vytvorime objekt okna tj. objekt tridy QWidget
  QWidget window;
  // Nastavime titulek ktery se zobrazi v zahlavi okna
  window.setWindowTitle("Muj prvni program vytvoreny v Qt!");
  // Zavolame metodu show() ktera okno zobrazi
  window.show();
  return app.exec();
}
```

# Grafický výstup v Qt

- Pro kreslení do okna vytvoříme třídu odvozenou z `QWidget` a tu použijeme pro vytvoření objektu okna
- Kreslení v okně se provádí definováním virtuální metody `paintEvent()`, která překrývá metodu `paintEvent()` definovanou v `QWidget` (tato metoda je **protected**)
- Metoda `paintEvent()` je zavolána vždy, když je potřeba okno překreslit

```
class GraphicWidget : public QWidget
{
  // Zde budou dalsi metody a data dle potreby
  protected:
    // V metode paintEvent() budou uvedeny prikazy pro kresleni
    virtual void paintEvent(QPaintEvent* event);
};

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  GraphicWidget graphicWidget;
  graphicWidget.setWindowTitle("Program s vlastnim widgetem!");
  graphicWidget.show();
  return app.exec();
}
```

# Třída QPainter

- Pro kreslení vytvoříme v metodě `paintEvent()` objekt typu `QPainter`, do konstruktoru předáme ukazatel na objekt volající metody, který získáme pomocí klíčového slova **`this`**
- Třída `QPainter` obsahuje metody pro kreslení, např.:

```
void drawLine ( int x1, int y1, int x2, int y2 )
void drawRect ( int x, int y, int width, int height )
void drawEllipse ( int x, int y, int width, int height )
```

- Počátek souřadnic je v levém horním rohu, souřadnice y nabývá kladných hodnot směrem dolů, souřadnice x směrem doprava

```
// Na zacatku musime vlozit hlavickovy soubor QPainter
void GraphicWidget::paintEvent(QPaintEvent* event)
{
  QPainter painter(this);
  // Kresli caru z bodu se souradnicemi 30, 10 do bodu 270, 290
  painter.drawLine(30, 10, 270, 290);
  // Kresli obdelnik se souradnicemi leveho horniho rohu 170 a 20,
  // sirkou 110 a vyskou 80
  painter.drawRect(170, 20, 110, 80);
  // Kresli elipsu se souradnicemi "leveho horniho rohu" 20 a 190,
  // sirkou 120 a vyskou 90
  painter.drawEllipse(20, 190, 120, 90);
}
```

# Třídy pro nastavení parametrů kreslení

- Pro nastavení barev pro kreslení, tloušťky čáry, výplně, velikosti a typu fontu a pod. používáme pomocné třídy, které uchovávají příslušné hodnoty:

  `QColor` – pomocná třída pro specifikaci barvy je využívána následujícími třídami

  `QPen` – pro nastavení tloušťky, barvy a stylu čáry

  `QBrush` – pro nastavení typu a barvy výplně

  `QFont` – pro nastavení typu, velikosti a barvy písma

- Hodnoty často specifikujeme tak, že je předáme do konstruktoru; tyto třídy obsahují několik přetížených konstruktorů umožňující předat jen ty parametry, které chceme nastavit
- V objektu typu `QPainter` voláme metody `setPen()`, `setBrush()`, `setFont()`, kterým předáme proměnné těchto typů jako parametry
- V těchto případech často používáme nepojmenované dočasné proměnné

# Třída QColor

- Třída QColor obsahuje celočíselné hodnoty 0 – 255 složek barvy (red, blue, green) a průhlednost (tzv. alfa kanál)
- Hodnoty složek barvy nejčastěji předáváme přímo do konstruktoru `QColor ( int r, int g, int b, int a = 255 )`
- V knihovně Qt je předdefinovaných několik barev: `Qt::white`, `Qt::black`, `Qt::red`, `Qt::green`, `Qt::blue`, `Qt::cyan`, `Qt::magenta`, `Qt::yellow`, `Qt::gray` atd. (viz. https://doc.qt.io/qt-5/qt.html#GlobalColor-enum)
- Objekty QColor se často využívají pro nastavení barvy v objektech QPen a QBrush

```
QColor color1(128, 210, 255);

QPen pen1(color1);
QPen pen2(Qt::red);
QPen pen3(QColor(90, 128, 128));

QBrush brush1(color1);
QBrush brush2(Qt::green);
QBrush brush3(QColor(90, 128, 128));
```

# Třída QPen

- Třída QPen obsahuje parametry pro kreslení čar a obrysů
- Třída obsahuje informace o barvě, tloušťce čáry, stylu čáry (plná, čárkovaná a pod.) a některé další parametry
- Hodnoty se zpravidla předávají přímo do konstruktorů:

```
QPen ( const QColor & color )
QPen ( const QBrush & brush, qreal width,
   Qt::PenStyle style = Qt::SolidLine,
   Qt::PenCapStyle cap = Qt::SquareCap,
   Qt::PenJoinStyle join = Qt::BevelJoin )
```

- Více na: https://doc.qt.io/qt-5/qpen.html

```
// Vytvori pero cervene barvy, ostatni hodnoty budou implicitni
QPen pen1(Qt::red);

// Vytvori pero modre barvy s tloustkou cary 10 pixelu
QPen pen2(QBrush(Qt::blue), 10);

// Misto parametru QBrush muzeme predat primo hodnotu barvy
QPen pen3(Qt::green, 10);

// Vytvori cervene pero s tloustkou 5 kreslici carkovanou caru
QPen pen4(Qt::red, 5, Qt::DashLine);
```

# Třída QBrush

- Třída QBrush obsahuje parametry pro kreslení výplně objektů
- Třída obsahuje informace o barvě výplně a stylu výplně
- Hodnoty se zpravidla předávají přímo do konstruktoru:

  QBrush ( const QColor & color,
  Qt::BrushStyle style = Qt::SolidPattern )
- Více na: https://doc.qt.io/qt-5/qbrush.html

```
// Vytvorime stetec pro vyplneni souvislou cernou barvou
QBrush brush1(Qt::red);

// Vytvorime stetec pro vyplneni barvou s prislusnymi hodnotami RGB
QBrush brush2(QColor(128, 100, 190));

// Vytvorime zeleny stetec s vyplnovym vzorem tvorenym
// diagonalnimi carami
QBrush brush3(Qt::green, Qt::BDiagPattern);
```

# Třída QFont

- Třída QFont slouží pro specifikaci fontu
- Specifikovat lze název fontu, velikost fontu, tloušťku (tučnost) písma a použití kurzívy
- Hodnoty se zpravidla předávají do konstruktoru:

```
QFont ( const QString & family, int pointSize = -1,
    int weight = -1, bool italic = false )
```

- Je-li hodnota pointSize záporná, použije se velikost fontu 12
- Parametr weight nabývá hodnot 0 – 99, lze použít předdefinované konstanty QFont::Normal, QFont::Bold a další
- Pro vypsání textu do okna se používá metoda:

```
void drawText ( int x, int y, const QString & text )
```

```
QPainter painter(this);
// Vytvorime font Arial s implicitni velikosti 12 bodu
QFont font1("Arial");
// Vytvorime font Times New Roman s velikosti 18 bodu
QFont font2("Times New Roman", 18);
// Vytvorime font Helvetica s velikosti 18 bodu, tucna kurziva
QFont font3("Helvetica", 18, QFont::Bold, true);
painter.setFont(font3);
painter.drawText(20, 330, "Toto je text");
```

# Použití tříd QColor, QPen, QBrush a QFont

- Třídy QColor, QPen, QBrush a QFont používáme ve spojení s metodami třídy QPainter pro nastavení parametrů kreslení:

```
void setPen ( const QPen & pen )
void setBrush ( const QBrush & brush )
void setFont ( const QFont & font )
```

- Třídy můžeme použít pro vytvoření objektové proměnné, nebo je používáme přímo, tj. jako nepojmenované proměnné

```
QPainter painter(this);
QPen pen1(Qt::blue, 4);
QBrush brush1(Qt::red);

painter.setPen(pen1);
painter.setBrush(brush1);
painter.drawRect(170, 20, 110, 80);

painter.setPen(QPen(QColor(255, 0, 0), 2));
painter.setBrush(QBrush(Qt::green));
painter.drawEllipse(20, 190, 120, 90);

// Misto QBrush muzeme pouzivat barvu QColor, protoze QBrush ma
// konstruktor ktery umi automaticky konvertovat QColor
painter.setBrush(Qt::blue);
painter.drawRect(20, 20, 110, 80);
```

# Zpracování událostí od myši

- Události od myši jsou zpracovávány v cyklu událostí v metodě `QApplication::exec()`, která při výskytu události (kliknutí myši, pohyb myší) zavolá příslušné metody okna (tj. metody třídy `Qwidget`), v němž se nachází kurzor myši
- Pro různé události myši se volají odlišné metody:

```
virtual void mousePressEvent ( QMouseEvent * event )
virtual void mouseReleaseEvent ( QMouseEvent * event )
virtual void mouseDoubleClickEvent ( QMouseEvent * event )
virtual void mouseMoveEvent ( QMouseEvent * event )
```

- Pro zpracování události od myši definujeme příslušnou virtuální metodu (např. `mousePressEvent()`), která překryje příslušnou metodu definovanou v `QWidget` (je **`protected`**)
- Informace o pozici kurzoru a stisknutém tlačítku získáme z parametru `event`, který je objektem třídy `QMouseEvent`
- Třída `QMouseEvent` obsahuje metody `x()` a `y()` poskytující souřadnice kurzoru muši a `button()` pro získání informace o stisknutém tlačítku (nabývá hodnot `Qt::LeftButton`, `Qt::MiddleButton`, `Qt::RightButton`). Nepoužívejte zastaralý `Qt::MidButton`!

# Zpracování událostí od myši - příklad

```
// Vytvorime tridu GraphicWidget, kterou pouzijeme pro vytvoreni okna
class GraphicWidget : public QWidget
{
  protected:
    virtual void paintEvent(QPaintEvent *event);
    virtual void mousePressEvent(QMouseEvent *event);
    // Zde mohou byt dalsi cleny tridy
};

// Metoda mousePressEvent() bude volana pokazde, kdyz stiskneme
// tlacitko mysi
void GraphicWidget::mousePressEvent(QMouseEvent *event)
{
  cout << "Stisknuto tlacitko mysi!" << endl;
  cout << "  Souradnice mysi: ";
  cout << event->x() << ", " << event->y() << endl;
  cout << "  Je stisknuto nasledujici tlacitko mysi: ";
  if (event->button() == Qt::LeftButton) cout << "leve";
  else if (event->button() == Qt::MiddleButton) cout << "prostredni";
  else if (event->button() == Qt::RightButton) cout << "prave";
  cout << endl;
}
```

# Událost od myši a překreslení okna

- Na událost od myši program často potřebuje reagovat tak, že změní obsah vykreslovaný do okna
- Pokud chceme vynutit překreslení okna, zavoláme metodu `update()` třídy `QWidget`:

```
void update ()
```

- Po zavolání metody `update()` provede program překreslení, tedy zavolá metodu `paintEvent()` pro dané okno.

# Událost od myši a překreslení okna - příklad

```
// Na zacatku musime vlozit hlavickovy soubor QMouseEvent
class GraphicWidget : public QWidget
{
  public:
    GraphicWidget();
  protected:
    virtual void mousePressEvent(QMouseEvent *event);
    virtual void paintEvent(QPaintEvent *event);
  private:
    // Nasledujici promenne slouzi pro ulozeni souradnic mysi
    // ktere se jim priradi vzdy pri udalosti stisknuti tlacitka
    // a pak se pouziji pro vykresleni ctverecku na dane pozici.
    int posX = 0, posY = 0;
};

void GraphicWidget::mousePressEvent(QMouseEvent *event)
{
  posX = event->x(); posY = event->y();
  update();  // Vyvolame pozadavek na prekresleni okna
}

void GraphicWidget::paintEvent(QPaintEvent *event)
{
  QPainter painter(this);
  painter.drawRect(posX - 5, posY - 5, 10, 10);
}
```

# Použití třídy Application s Qt

- Pokud v programu používáme třídu Application, je vhodné ji odvodit z QApplication
- Konstruktor třídy Application musí přijímat argumenty z příkazového řádku a předat je konstruktoru QApplication
- Argument argc je třeba vždy předávat referencí, jinak program havaruje

```
#include <QApplication>
#include <QWidget>
using namespace std;
// Na zacatku bude definovana trida MyWindow odvozena z QWidget

class Application : public QApplication
{
  public:
    Application(int &argc, char *argv[]);
    // Pokud bychom potrebovali provadet nejaky uklid dat pri ukonceni
    // aplikace, muzeme pridat i destruktor ~Application();

    int run();
  private:
    MyWindow window;
};
// Pokracovani na nasledujici strance
```

# Použití třídy Application s Qt - pokračování

```
// Pokracovani z predchozi stranky
Application::Application(int &argc, char *argv[]) : QApplication(argc, argv)
{
}

int Application::run()
{
  window.setWindowTitle("Muj program vytvoreny v Qt!");
  window.show();
  return QApplication::exec();
}

int main(int argc, char *argv[])
{
  Application app(argc, argv);
  return app.run();
}
```

# Pomocné třídy v Qt

- Knihovna Qt hojně používá následující pomocné třídy
  - `QPoint` pro bod se souřadnicemi x a y
  - `QRect` pro obdélníkový region
  - `QString` jako bohatší alternativa ke třídě `string`
  - `QFile` pro práci se soubory (obohacená alternativa k `iostream`)
- Ke třídám `QPoint` a `QRect` existují dvě podobné alternativy `QPointF` a `QRectF` které pracují s hodnotami typu **`float`** namísto **`int`**
- Dokumentace k jednotlivým třídám: https://doc.qt.io/qt-5/classes.html

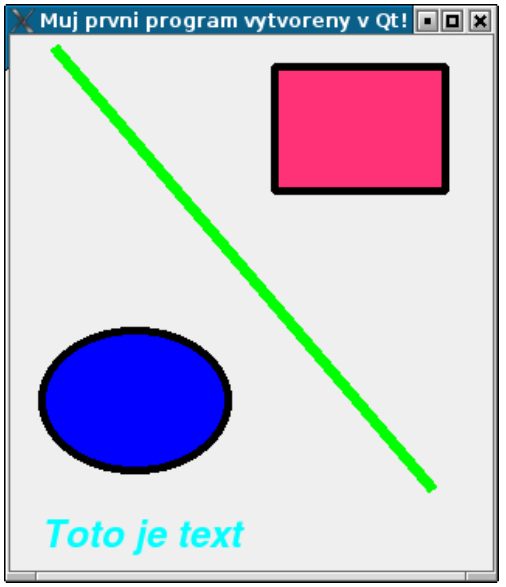
# Dodržujte následující pravidla

- Zdrojové soubory každé úlohy umístěte do samostatného adresáře, jehož jméno se bude rovnat jménu výsledného programu. Pak vygenerujte soubor projektu.
- Poté co vygenerujete soubor projektu **.pro* (příkazem `qmake -project`) nezapomeňte do tohoto souboru přidat řádek `QT += widgets`. Pro snazší ladění přidejte i řádek `CONFIG += debug`. Teprve potom vygenerujte soubor *Makefile* příkazem `qmake`
- Při použití kterékoli třídy z Qt knihovny nezapomeňte vložit příslušný hlavičkový soubor (má stejný název jako třída).
- Dokumentaci k jednotlivým třídám najdete zde: https://doc.qt.io/qt-5/classes.html
- Seznam barev předdefinovaných v Qt knihovně lze najít zde: https://doc.qt.io/qt-5/qt.html#GlobalColor-enum
- Ke každé úloze odevzdejte .cpp a .pro soubor (neodevzdávejte generovaný *Makefile*) do příslušné podsložky v odevzdávárně.

# Cvičení – část 1

1. Vytvořte program využívající knihovnu Qt, který nakreslí do okna elipsu, obdélník a čáru zhruba tak, jak je uvedeno na obrázku. Obdélník bude vyplněný barvou s RGB hodnotami 255, 50, 120 a elipsa modrou barvou (`Qt::blue`). Tloušťka čar bude 7. Do spodní části okna se vypíše libovolný text (fontem Helvetica, velikost 18, tučné písmo, barva `Qt::cyan`). **1 bod**

2. Do programu přidejte obsluhu stisknutí tlačítka myši tak, že po stisknutí se vypíše na terminál pozice kurzoru myši a informace o tom, které tlačítko bylo stisknuto. Dále se po stisknutí tlačítka nakreslí v okně malý červený čtvereček na místě, kde bylo tlačítko stisknuto. Zajistěte, aby se čtvereček nekreslil ještě před prvním kliknutím do okna. **2 body**



# Cvičení – část 2

3. Upravte program z úlohy 2 ze cvičení 7 (načítání grafických objektů ze souboru `shapes2.dat` a jejich ukládání do jednoho společného vektoru) tak, aby místo knihovny g2 využíval Qt.
   - Odstraňte `#include g2.h` a `g2_X11.h`
   - Všechny metody `draw()` změňte tak, aby místo čísla zařízení přijímaly referenci na `QPainter`.
   - Třídu `Drawing` odvoďte od `QWidget` a použijte ji k reprezentaci hlavního okna aplikace. Metodu `Drawing::draw()` přetvořte ve virtuální metodu `Drawing::paintEvent()` a v ní volejte `draw()` pro všechny obrazce ve vektoru.
   - Deklarujte třídu `Application` odvozenou od `QApplication`. Všechnu logiku z `main()` přesuňte do `Application::run()`.
   - V `readFile()` budete muset psát `std::move()` místo holého `move()`.
   - Všechna čísla barev z g2 nahraďte Qt konstantami (`Qt::black` aj.). Hodnoty typu `int` můžete následně používat takto: `setBrush(Qt::GlobalColor(fillColor))`.
   - Pro zjednodušení programu můžete odmazat nepotřebné metody (`printValues()`, `printClass()`, `test()`, atd.)
   - Po načtení rozměrů okna je předejte metodě `setMinimumSize()` objektu Drawing.
   - Nezapomeňte upravit souřadnice předávané `drawRect/drawEllipse` vzhledem k odlišnému významu argumentů v g2 a Qt (souřadnice rohu vs středu, šířka a výška vs poloměr). Odlišnou orientaci os (vertikální převrácení obrazu) ignorujte.

   **nepovinná, 4 body**

4. Předchozí program modifikujte tak, aby se místo `int` pro uložení barev používal typ `QColor` (tzn. `getColorNumberFromString` bude vracet `QColor`, všechny ostatní metody budou přijímat a vracet konstantní reference). **nepovinná, 2 body**