Curs 8

- **Polimorfism**
- **Interfețe grafice utilizator - Qt**

Curs 7 Moștenire, Polimorfism

- **Moștenire**
- **Polimorfism – Metode pur virtuale, Clase abstracte**
- **Operații de intrări ieșiri în C++**
  - **Fișiere**

Polimorfism

Proprietatea unor entități de:

- a se comporta diferit în funcție de tipul lor
- a reacționa diferit la același mesaj

Obiecte din diverse clase care sunt legate prin relații de moștenire să răspundă diferit la același mesaj (apel de metodă).

Proprietate a unui limbaj OO de a permite manipularea unor obiecte diferite prin intermediul unei interfețe comune

Polimorfism in C++

1 Moștenire

2 Suprascriere metoda in clasa derivata

3 Metoda virtuala in clasa de baza (sau pur virtuala)

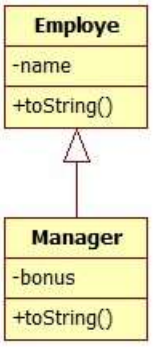
4 Accesat prin pointer sau referința (nu prin valoare => apare fenomenul de sclicing)

Metoda potrivita se alege in timpul execuției (legare întârziata/dynamic bindig/late binding) in loc sa fie ales la compilare (legare statica/static binding)

Mecanismul din spate: VTable – tabela cu adresa metodelor virtuale

Exercițiu polimorfism

Scrieți codul C++ care corespunde diagramei UML de clase:



Compania are mai mulți angajați (angajați normali si manageri).
Metoda ***toString*** din clasa **Employe** returnează un string cu numele angajatului.
Metoda ***toString*** din clasa **Manager** returnează un string ce in cepe cu „Manager:” apoi numele managerului.

Scrieți un program care creează o lista de angajați (atât angajați cât si manageri) si tipărește lista. Creați o funcție care primește o lista de angajați si tipărește stringul returnat de metoda ***toString*** de la **Employe** respectiv **Manager.**

Ex de output:
Ion
Manager: Pop

Rezolvare:

```
class Employee {
private:
      string name;
public:
    Employee(string n) :name{ n } {}
     virtual void toString() {
           cout << name;
     }
     virtual ~Employee() {
     }
};
```

```
class Manager :public Employee {
private:
     double bonus;
public:
   Manager(string n, double b):
        Employee{ n }, bonus{ b } {}

   void toString() override {
          cout << "Manager:";
          Employee::toString();
   }
};
```

```
void printAll(const vector<Employee*>& emps) {
     for (auto& emp : emps) {emp->toString();}
}
int main(){
     vector<Employee*> emps;
     emps.push_back(new Employee{ "Ion" });
     emps.push_back(new Manager{ "Ion",5.0 });
     emps.push_back(new Employee{ "Pop" });
     printAll(emps);
     for (auto emp : emps) {delete emp;} //delocam obiectele
}
```

```
//varianta cu unique_ptr
void printAllSmartPointer(const vector<unique_ptr<Employee>>& emps) {
     //unique_ptr nu se poate copia, trebuie sa folosim&
     for (auto& emp : emps) {emp->toString();}
}
void createAndPrintSmartPointer() {
     vector<unique_ptr<Employee>> emps;
     //make_unique - varianta preferata de a crea un unique_ptr
     //make_unique apeleaza constructorul de la Employee
     emps.push_back(make_unique<Employee>("Ion" ));
     emps.push_back(make_unique<Manager>( "Ion",5.0 ));

     unique_ptr<Employee> up{ new Employee{ "Pop" } };
     //fiindca unique_ptr nu se poate copia trebuie sa facem move
     //dupa move variabila up nu mai contine pointerul
     emps.push_back(std::move( up));
     printAllSmartPointer(emps);
}
```

Exercițiu:

Scrieți un program C++ care:
a. Definește o clasă **B** având un atribut privat ***b*** de tip întreg şi o metodă de tipărire care afișează atributul ***b*** la ieșirea standard.

b. Definește o clasă **D** derivată din **B** având un atribut ***d*** de tip șir de caractere şi de asemenea o metodă de tipărire pe ieșirea standard care va afișa atributul ***b*** din clasa de bază şi atributul ***d***.

c. Definește o funcție care construiește o listă conținând:
un obiect **o1** de tip **B** având ***b*** egal cu 8;
un obiect **o2** de tip **D** având ***b*** egal cu 5 şi ***d*** egal cu ″D5″;
un obiect **o3** de tip **B** având ***b*** egal cu -3;
un obiect **o4** de tip **D** având ***b*** egal cu 9 şi ***d*** egal cu ″D9″.

d. Definește o funcție care primește o listă cu obiecte de tip **B** şi tipărește lista de obiecte folosind metoda de tipărire.

Spații de nume

Introduc un domeniu de vizibilitate care nu poate conține duplicate

```
namespace testNamespace1 {
class A {
};
}
namespace testNamespace2 {
class A {
};
}
```

Accesul la elementele unui spațiu de nume se face folosind operatorul de rezoluție

```
void testNamespaces() {
	testNamespace1::A a1;
	testNamespace2::A a2;
}
```

Folosind directiva using putem importa:
toate elementele definite într-un spațiu de nume

```
void testUsing() {
	using namespace testNamespace1;
	A a;
}
```

Doar clasa/metoda pe care vrem sa folosim

```
using std::vector;
using std::copy_if;
using std::cout;
```

Qt Toolkit

Qt este un framework pentru crearea de aplicații cross-platform (același code pentru diferite sisteme de operare, dispozitive) în C++.

Folosind QT putem crea interfețe grafice utilizator. Codul odată scris poate fi compulat pentru diferite sisteme de operare, platforme mobile fără a necesita modificări în codul sursă.

Qt suportă multiple platforme de 32/64-bit (Desktop, embedded, mobile).

- Windows (MinGW, MSVC)
- Linux (gcc)
- Apple Mac OS
- Mobile / Embedded (Windows CE, Symbian, Embedded Linux)

Este o librărie C++ dar există posibilitatea de a folosi și din alte limbaje: C# ,Java, Python(PyQt), Ada, Pascal, Perl, PHP(PHP-Qt), Ruby(RubyQt)

Qt este disponibil atât sub licență GPL v3, LGPL v2 cât și licențe comerciale.

Exemple de aplicații create folosind Qt:

Google Earth, KDE (desktop enviroment for Unix-like OS), Adobe Photoshop Album, etc

Resurse: https://www.qt.io/

QT - Module și utilitare

- **Qt Library -** bibliotecă de clase C++, oferă clasele necesare pentru a crea aplicații (cross-platform applications)

- **Qt Creator** - mediu de dezvoltare integrat (IDE) pentru a crea aplicații folosind QT

- **Qt Designer** – instrument de creare de interfețe grafice utilizator folosind componente QT

- **Qt Assistant** – aplicație ce conține documentație pentru Qt și facilitează accesul la documentațiile diferitelor parți din QT

- **Qt Linguist** – suport pentru aplicații care funcționează în diferite limbi (internaționalizare)

- **qmake** – aplicație folosit în procesul de creare proiecte/compilare

Download instal QT

Ultima versiune QT 6.0.3

https://www.qt.io/download

Alegeți varianta Open Source (Downloads for open source users).

Installer ce downloadeaza cele necesare. Atenție daca instalați tot ocupa destul de mult spațiu pe disk (peste 10Gb), puteți sa instalați doar cele esențiale (ex. Instalați doar varianta compilata cu VisualStudio 2019)

Important: Sa va asigurati ca folositi varianta potrivita pentru calculatorul vostru:

- procesorul 32 vs 64 biti
- sistemul de operare (windows, linux, mac)
- compilatorul folosit: MinGW (eclipse) vs Microsoft (Visual studio) – in timpul instalarii

După ce aveți Qt instalat aveți mai multe variante de a dezvolta aplicații C++ cu Qt:

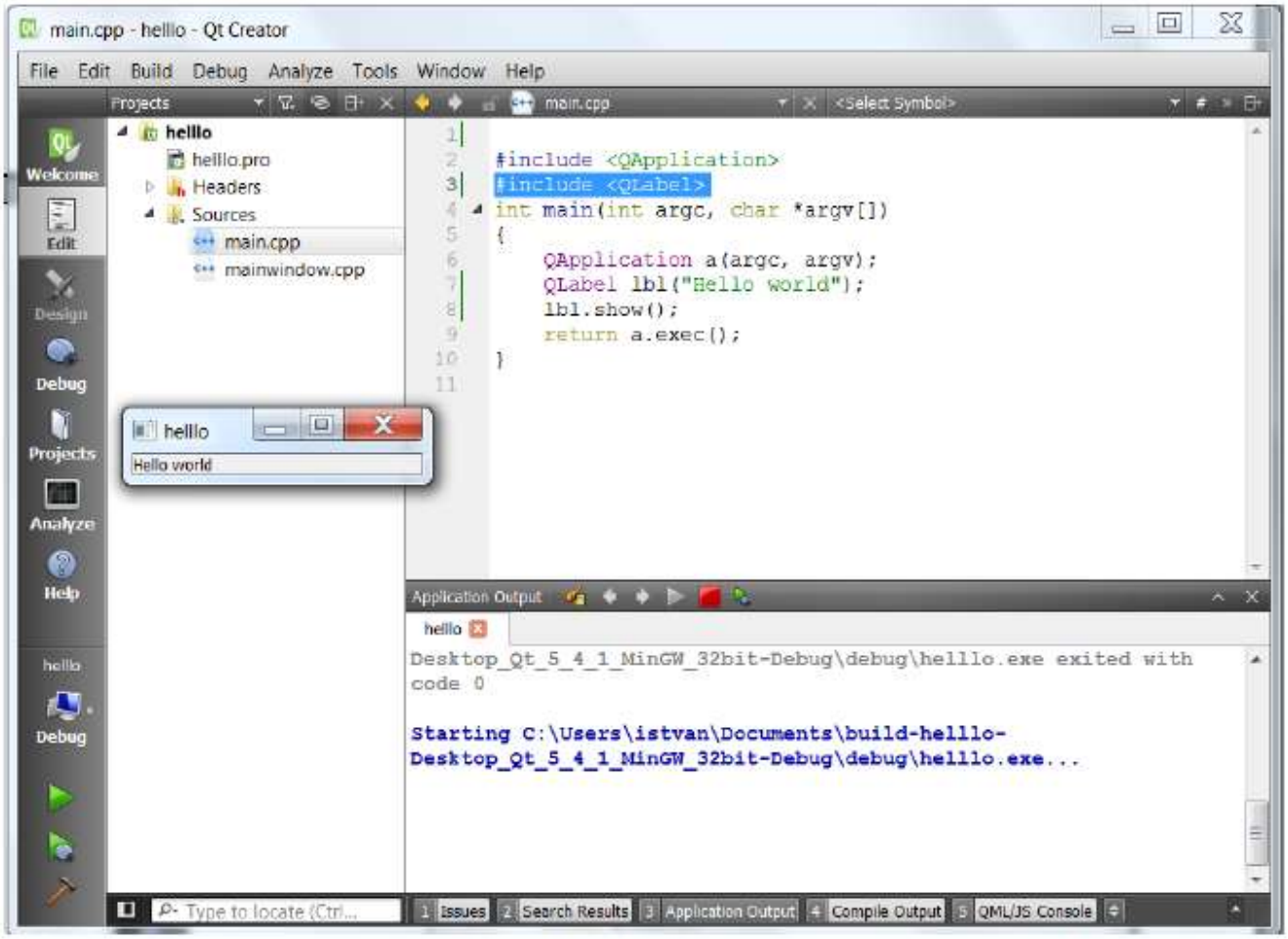
- Folosind Visual Studio + Extensia Qt - varianta preferata la aceasta materie
- QtCreator – IDE instalat cu Qt
- Din linie de comanda folosind utilitarul qmake – puteți ulterior folosi orice IDE care poate lucra cu proiecte care folosesc makefile

QT Creator – Hello world

Prerequisite: Download/install Qt

Qt Creator - IDE pentru dezvoltare de aplicații Qt

File -> New File or Project -> Qt Widget Application



Qt Hello World – Exemplu folosire linie de comanda

Prerequisite: MinGW; MinSYS; Qt installed

Variabila PATH trebuie sa conține calea către qt (qmake.exe), mingw (g++.exe), minsys (make.exe)

```
set PATH=C:\MinGW\msys\1.0\bin\
set PATH=%PATH%;C:\Qt\5.10.1\mingw491_32\bin\
set PATH=%PATH%;C:\Qt\Tools\mingw491_32\bin\
```

Pas 1: Creati un fisier .cpp (main.cpp)

```
#include <QApplication>
#include <QLabel>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QLabel lbl("Hello world");
    lbl.show();
    return a.exec();
}
```

Pas 2: Executați: **qmake -project** => se generează un fișier .pro

Pas 3: Editați fișierul .pro Adaugati: QT += core gui

Pas 4: Executați: **qmake** => Makefiles generated

Pas 5: Executați: **make** => proiectul este compilat, link-editat => exe

Pas 6: rulați aplicația **main.exe**

Puteți edita fișierele .cpp cu ce editor doriți, compilați cu comanda make

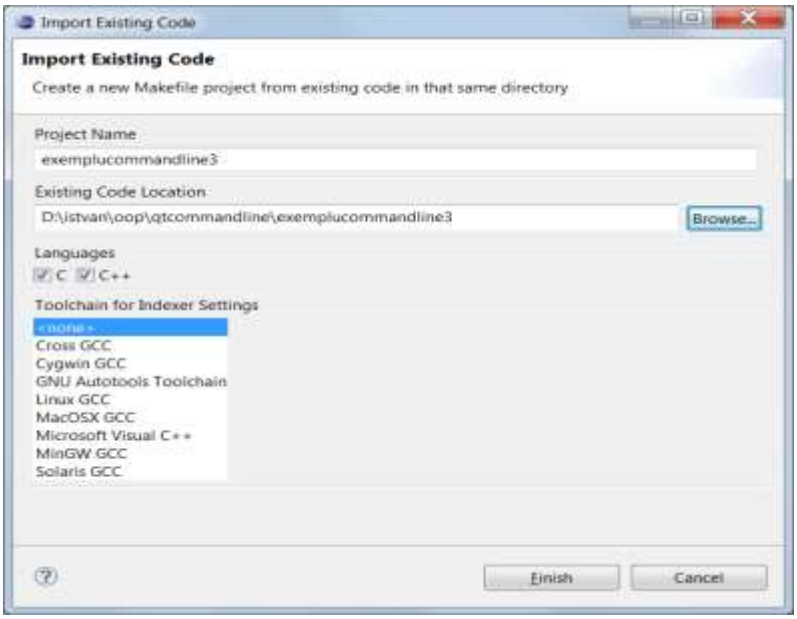
La pasul 4 s-a generate un proiect care folosește makefile, astfel de proiecte se pot importa in majoritatea IDE-urilor care suporta C++

Qt Hello World - Eclipse

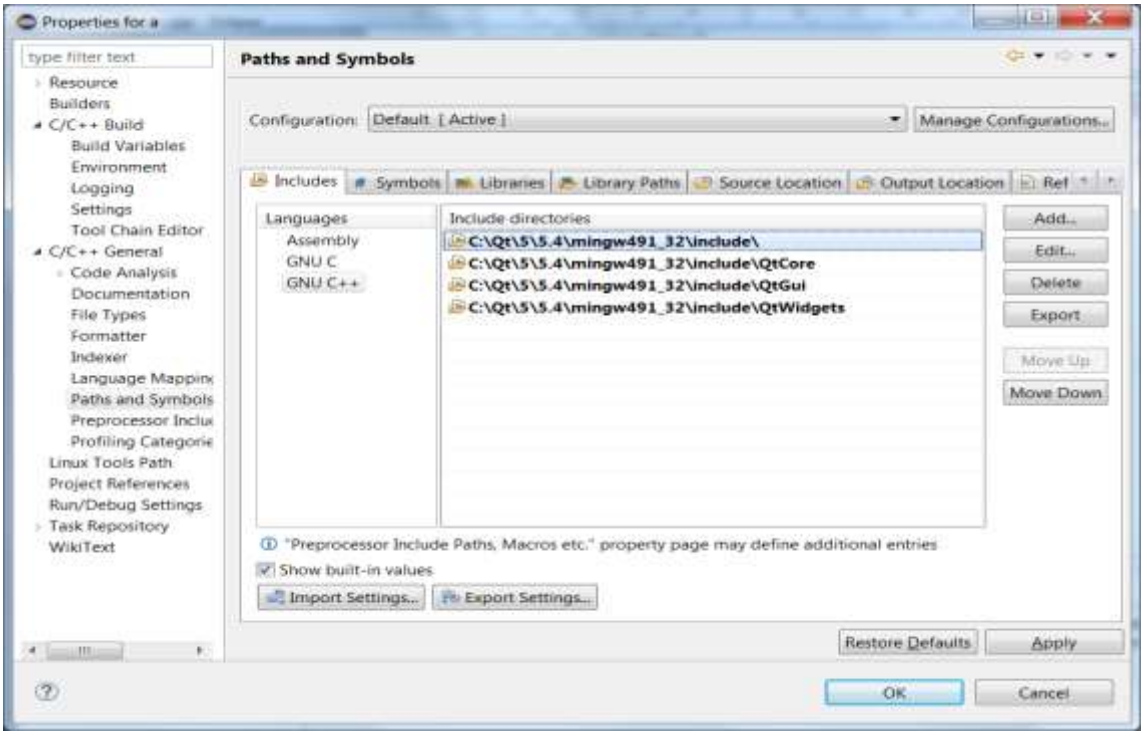
Prerequisite: Working Eclipse CDC (MinGW), Qt Library installed

Executați Pașii **1 - 5**

Proiectul se poate importa in Eclipse: **File->New-> Makefile project with Existing Code**



Adaugat libraria Qt: Project->Properties->C/C++ General->Path and Symbols → Includes



Qt Hello World – Visual Studio

Instalați pachetul de extensie QT in Visual Studio:

Tools->Extension and Updates->Qt Visual Studio Tools

Apare un nou meniu (Qt VS Tools) si noi tipuri de proiect pe care puteți crea in Visual Studio (File-> New->Project-> **Qt Widgets Application**)

Adăugați în main:

```
int main(int argc, char *argv[]) {
     QApplication app(argc, argv);
     QLabel *label = new QLabel("hello world");
     label->show();
     return app.exec();
}
```

Compilați/Rulați aplicația

QApplication

Clasa QApplication gestionează fluxul de evenimente și setările generale pentru aplicațiile Qt cu interfață grafică utilizator (GUI)

QApplication preia evenimentele de la sistemul de operare și distribuie către componentele Qt (main event loop), toate evenimentele ce provin de la sistemul de ferestre (windows, kde, x11,etc) sunt procesate folosind această clasă.

Pentru orice aplicație Qt cu GUI, există un obiect QApplication (indiferent de numărul de ferestre din aplicație există un singur obiect QApplication)

Responsabilități:

- inițializează aplicația conform setărilor sistem
- gestionează fluxul de evenimente - generate de sistemul de ferestre (x11, windows) și distribuite către componentele grafice Qt (widgets).
- Are referințe către ferestrele aplicației
- definește look and feel

Pentru aplicații Qt fără interfață grafică se folosește QCoreApplication.

```
app.exec();
```

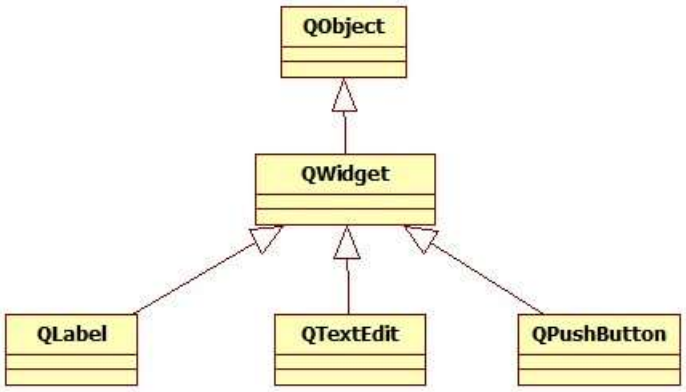
- pornește procesarea de evenimente din QApplication (event loop). În timp ce rulează aplicația evenimente sunt generate și trimise către componentele grafice.

Componente grafice QT (widgets)

Sunt elementele de bază folosite pentru a construi interfețe grafice utilizator

- butoane, etichete, căsuțe de text , etc

Orice componentă grafică Qt (widget) poate fi adăugat pe o fereastră sau deschis independent într-o fereastră separată.



Dacă componenta nu este adăugat într-o componentă părinte avem de fapt o fereastră separată .
Ferestrele separă vizual aplicațiile între ele și sunt decorate cu diferite elemente (bară de titlu, instrumente de poziționare, redimensionare, etc)

Widget : Etichetă, buton, căsuță de text, listă

QLabel

- QLabel se folosește pentru a prezenta un text sau o imagine. Nu oferă interacțiune cu utilizatorul
- QLabel este folosit de obicei ca și o etichetă pentru o componentă interactivă. QLabel oferă mecanism de mnemonic, o scurtătură prin care se setează focusul pe componenta atașată (numit "buddy").

```
QLabel *label = new QLabel("hello world");
label->show();

QLineEdit txt(parent);
QLabel lblName("&Name:", parent);
lblName.setBuddy(&txt);
```

QPushButton

QPushButton widget - buton

- Se apasă butonul (click) pentru a efectua o operație
- Butonul are un text si opțional o iconiță. Poate fi specificat si o tastă rapidă (shortcut) folosind caracterul & în text

```
QPushButton btn("&TestBTN");
btn.show();
```

Widget : Etichetă, buton, căsuță de text, listă

QLineEdit

- Căsuța de text (o singură linie)
- Permite utilizatorului sa introducă informații. Oferă funcții de editare (undo, redo, cut, paste, drag and drop).

```
QLineEdit txtName;
txtName.show();
```

QTextEdit este o componentă similară care permite introducere de text pe mai multe linii și oferă funcționalități avansate de editare/vizualizare.

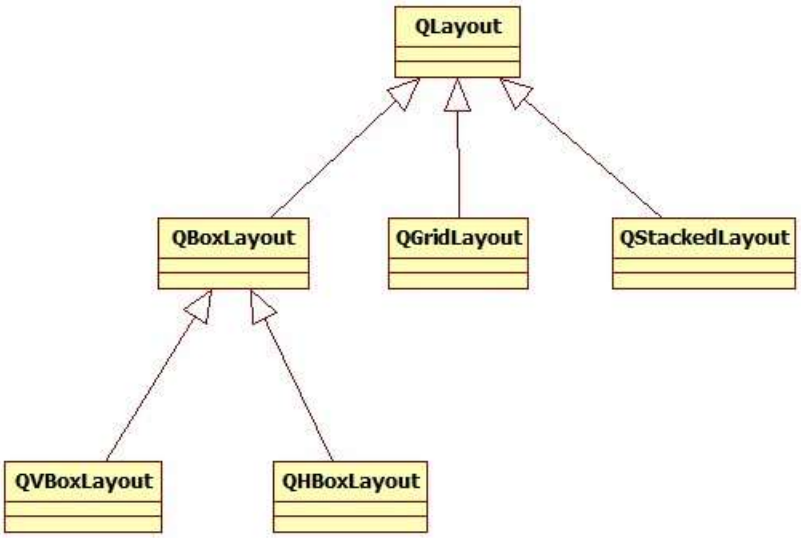
QListWidget

Prezintă o listă de elemente

```
QListWidget *list = new QListWidget;
new QListWidgetItem("Item 1", list);
new QListWidgetItem("Item 2", list);
QListWidgetItem *item3 = new QListWidgetItem("Item 3");
list->insertItem(0, item3);
list->show();
```

Layout management

- Orice QWidget are o componentă părinte.
- Sistemul Qt layout oferă o metodă de a aranja automat componentele pe interfață.
- Qt include un set de clase pentru layout management, aceste clase oferă diferite strategii de aranjare automată a componentelor.
- Componentele sunt poziționate / redimensionate automat conform strategiei implementate de layout manager și luând în considerare spațiul disponibil pe ecran. Folosind diferite layouturi putem crea interfețe grafice utilizator care acomodează diferite dimensiuni ale ferestrei.



Layout management

```
QWidget *wnd = new QWidget;
QHBoxLayout *hLay = new QHBoxLayout();
QPushButton *btn1 = new QPushButton("Bt &1");
QPushButton *btn2 = new QPushButton("Bt &2");
QPushButton *btn3 = new QPushButton("Bt &3");
hLay->addWidget(btn1);
hLay->addWidget(btn2);
hLay->addWidget(btn3);
wnd->setLayout(hLay);
wnd->show();
```

```
QWidget *wnd2 = new QWidget;
QVBoxLayout *vLay = new QVBoxLayout();
QPushButton *btnn1=new QPushButton("B&1");
QPushButton *btnn2= new QPushButton("B&2");
QPushButton *btnn3= new QPushButton("B&3");
vLay->addWidget(btnn1);
vLay->addWidget(btnn2);
vLay->addWidget(btnn3);
wnd2->setLayout(vLay);
wnd2->show();
```

GUI putem compune multiple componente care folosesc diferite strategii de aranjare pentru a crea interfața utilizator dorită

```
QWidget *wnd3 = new QWidget;
QVBoxLayout *vL = new QVBoxLayout;
wnd3->setLayout(vL);
//create a detail widget
QWidget *details = new QWidget;
QFormLayout *fL = new QFormLayout;
details->setLayout(fL);
QLabel *lblName = new QLabel("Name");
QLineEdit *txtName = new QLineEdit;
fL->addRow(lblName, txtName);
QLabel *lblAge = new QLabel("Age");
QLineEdit *txtAge = new QLineEdit;
fL->addRow(lblAge, txtAge);
//add detail to window
vL->addWidget(details);
QPushButton *store = new QPushButton("&Store");
vL->addWidget(store);
//show window
wnd3->show();
```

Layout management

addStretch() se folosește pentru a consuma spațiu. Practic se adaugă un spațiu care se redimensionează in funcție de strategia de aranjare

```
QHBoxLayout* btnsL = new QHBoxLayout;
btns->setLayout(btnsL);
QPushButton* store = new QPushButton("&Store");
btnsL->addWidget(store);
btnsL->addStretch();
QPushButton* close = new QPushButton("&Close");
btnsL->addWidget(close);
```

Layout manager o să adauge spațiu între butonul Store și Close

Layout management

Cum construim interfețele grafice:

- creăm componentele necesare
- setăm proprietățile componentelor dacă este necesar
- adăugăm componenta la un layout (layout manager se ocupă cu dimensiunea poziția componentelor)
- conectăm componentele între ele folosind mecanismul de signal și slot

Avantaje:

- oferă un comportament consistent indiferent de dimensiunea ecranului/ferestrei, se ocupa de rearanjarea componentelor în caz de redimensionare a componentei

- setează valori implicite pentru componentele adăugate

- se adaptează în funcție de fonturi și alte setări sistem legate de interfețele utilizator.

- Se adaptează în funcție de textul afișat de componentă. Aspect important dacă avem aplicații care funcționează în multiple limbi (se adaptează componenta pentru a evita trunchierea de text).

- Dacă adugăm/ștergem componente restul componentelor sunt rearanjate automat (similar și pentru *show() hide()* pentru o componentă)

Poziționare cu coordonate absolute

```
/**
 * Create GUI using absolute positioning
 */
void createAbsolute() {
      QWidget* main = new QWidget();
      QLabel* lbl = new QLabel("Name:", main);
      lbl->setGeometry(10, 10, 40, 20);
      QLineEdit* txt = new QLineEdit(main);
      txt->setGeometry(60, 10, 100, 20);
      main->show();
      main->setWindowTitle("Absolute");
}
/**
 * Create the same GUI using form layout
 */
void createWithLayout() {
      QWidget* main = new QWidget();
      QFormLayout *fL = new QFormLayout(main);
      QLabel* lbl = new QLabel("Name:", main);
      QLineEdit* txt = new QLineEdit(main);
      fL->addRow(lbl, txt);
      main->show();
      main->setWindowTitle("Layout");
      //fix the height to the "ideal" height
      main->setFixedHeight(main->sizeHint().height());
}
```

Dezavantaje:

- Utilizatorul nu poate redimensiona fereastra (la redimensionare componentele rămân pe loc și fereastra nu arată bine, nu folosește spațiu oferit).
- Nu ia în considerare fontul, dimensiunea textului (orice schimbare poate duce la text trunchiat).
- Pentru unele stiluri (look and feel) dimensiunea componentelor trebuie ajustată.
- Pozițiile și dimensiunile trebuie calculate manual (ușor de greșit, greu de întreținut)

Documentație Qt

- Qt Reference Documentation – conține descrieri pentru toate clasele, metodele din Qt
- Este disponibil în format HTML (directorul doc/html din instalarea Qt) și se poate citi folosind orice browser
- Qt Assistant – aplicație care ajută programatorul să caute în documentația Qt (mai ușor de folosit decât varianta cu browser)
- Documentația este disponibilă și online http://doc.qt.io/qt-5/index.html
- Pentru orice clasă găsiți descrieri detaliate pentru metode, atribute, semnale , sloturi