Curs 7

- **Excepții**

- **Spații de nume (Namespace)**
- **Interfețe grafice utilizator - Qt**

Tratarea excepțiilor în c++

excepții - situați anormale ce apar în timpul execuției

tratarea execepțiilor - mod organizat de a gestiona situațiile excepționale ce apar în timpul execuției

O exceptie este un eveniment ce se produce în timpul executiei unui program si care provoaca întreruperea cursului normal al executiei.

Elemente:

- **try block** marchează blocul de instructiuni care poate arunca excepții.
- **catch block** bloc de instrucțiuni care se executa în cazul în care apare o excepție (tratează excepția).
- Instrucțiunea **throw** mecanism prin care putem arunca (genera excepții) pentru a semnala codului client apariția unei probleme.

```
void testTryCatch() {
    // some code
    try {
        //code that may throw an exception
        throw 12;
        //code
    } catch (int error) {
        //error handling code
        cout << "Error ocurred." << error;
    }
}
```

Tratarea excepţiilor

- Codul care susceptibil de a arunca excepţie se pune într-un bloc de try .
- Adăugăm unu sau mai multe secțiuni de **catch** . Blocul de instrucțiuni din interiorul blocului catch este responsabil sa trateze excepția aparută.
- Dacă codul din interiorul blocului try (sau rice cod apelat de acesta) aruncă excepție, se transferă execuția la clauza catch corespunzătoare tipului excepției apărute. (excepțion handler)

```
void testTryCatchFlow(bool throwEx) {
	// some code
	try {
		cout << "code before the exception" << endl;
		if (throwEx) {
			cout << "throw (raise) exception" << endl;
			throw 12;
		}
		cout << "code after the exception" << endl;
	} catch (int error) {
		cout << "Error handling code " << endl;
	}
}
```

```
testTryCatchFlow(0);
testTryCatchFlow(1);
```

- Clauza catch nu trebuie neapărat sa fie în același metodă unde se aruncă excepția. Excepția se propagă.
- Când se aruncă o excepție, se caută cel mai apropiat bloc de **catch** care poate trata excepția ("unwinding the stack").
- Dacă nu avem clauză **catch** in funcția în care a apărut excepția, se caută clauza **catch** în funcția care a apelat funcția .
- Căutarea continuă pe stack până se găsește o clauză **catch** potrivită**.** Dacă excepția nu se tratează (nu există o clauză **catch** potrivită) programul se oprește semnalând eroarea apărută.

Excepții - obiecte

- Când se aruncă o excepție se poate folosi orice tip de date. Tipuri predefinite (int, char,etc) sau tipuri definite de utilizator (obiecte).

- Este recomandat să se creeaze clase pentru diferite tipuri de excepții care apar în aplicație
- Obiectul excepție este transmis ca referință sau pointer (pentru a evita copierea)
- Obiectul excepție este folosit pentru a transmite informații despre eroarea apărută

```
class POSError {
public:
	POSError(string message) :
			message(message) {
	}
	const string& getMessage() const {
		return message;
	}
private:
	string message;
};
```

```
class ValidationError: public POSError {
public:
	ValidationError(string message) :
			POSError(message) {
	}
};
```

```
void Sale::addSaleItem(double quantity, Product* product) {
	if (quantity < 0) {
		throw ValidationError("Quantity must be positive");
	}
	saleItems.push_back(new SaleItem(quantity, product));
}
```

```
try {
	pos->enterSaleItem(quantity, product);
	cout << "Sale total: " << pos->getSaleTotal() << endl;
} catch (ValidationError& err) {
	cout << err.getMessage() << endl;
}
```

Specificații

Putem include lista de excepții pe care o metodă le poate arunca în declarația funcției folosind specificațiile de excepții

Dacă nu includem specificațiile de excepții pentru o metoda asta înseamnă că metoda poate arunca orice tip excepție

```
//can throw any type of exceptions
void f1() {

}
```

```
//can not throw exceptions
void f3() throw (){

}
```

```
/**
 * create ,validate and store a product
 * code - product code
 * desc - product description
 * price - product price
 * throw ValidatorException if the product is invalid
 * throw Repository exception if the code is already used by a product
 */
Product Warehouse::addProduct(int code, string desc, double price)
           throw (ValidatorException, RepositoryException) {
     //create product
     Product p(code, desc, price);
     //validate product
     validator->validate(p);
     //store product
     repo->store(p);
     return p;
}
```

Excepții - Avantaje

Metode de gestiune a situațiilor excepționale:

- folosind coduri de eroare (funcția returnează un cod de eroare daca a apărut o problemă)
- folosind flaguri de control (în caz de erori se setează flaguri, care pot fi verificate ulterior de codul apelant)
- folosind mecanismul de excepții

Avantajele mecanismului de excepții:

- putem separa codul de gestiune în caz de eroare (error handling code) de fluxul normal de execuție
- multiple tipuri de erori se pot trata cu metodă. Clauza **catch** se selectează conform tipului excepției aruncate (moștenire). Putem folosi (...) pentru a trata orice tip de excepție (nerecomandat)
- Excepțiile nu se pot ignora. Dacă nu tratăm excepția (nu există clauză catch corespunzătoare) programul se termină.
- Funcțiile au mai puține parametrii, nu se pun diferite valori de retur. Funcția e mai ușor de înțeles și folosit
- Orice informații se pot transmite folosind excepțiile. Informația se propagă de la locul unde a apărut excepția până la clauza catch care tratează eroarea.
- Dacă într-o metodă nu putem trata excepția putem ignora și acesta se propagă la metoda apelantă. Se propagă până în locul unde putem lua acțiuni (să tratăm eroarea)

Când să aruncăm excepții

- Se aruncă excepție dacă metoda nu poate realiza operaţia promisă
- Folosim excepţii pentru a semnala erori neasteptate

- Putem folosi excepţii pentru a semnala încălcarea precondiţiilor.
    - În mod normal cel care apelează metoda este responsabil să furnizeze parametri actuali care satisfac precondiţia.

- Este de evitat folosirea prea frecventă (în alte scupuri decăt semnalarea unei situaţii excepţionale) de excepţii fiindcă excepţiile frecvente fac codul mai greu de inteles (execuţia sare de la flux normal la codul de tratare a exceptţiei)
- Aruncarea de excepţie cu singurul scop de a schimba fluxul de execuţie nu este recomandat.
- Constructorul / destructorul nu ar trebui sa arunce excepţii

Clauze catch

```
try {
      Product p = wh->addProduct(code, desc, price);
      cout << "product " << p.getDescription() << " added." << endl;
} catch (RepositoryException& ex) {
      cout << "Error on store:" << ex.getMsg() << endl;
} catch (ValidatorException& ex) {
      cout << "Error on validate:" << ex.getMsg() << endl;
} catch (...) {
      cout << "unknwn exception";
}
```

- Dacă se aruncă o excepţie în codul din blocul **try** se executa clauza catch conform tipului excepţiei aruncate
- Se execută doar una dintre clauzele **catch**
- Se executa clauza catch care corespunde tipului – exepţia este de acelaşi tip sau este derivat din tipul indicat în clauza **catch**
- **(...)** corespunde oricărui tip. Orice tip de excepţie se aruncă această clauză catch corespunde

Ierarhii de clase exceptii

- Excepțiile permit izolarea codului care gestionează eroarea apărută. De asemenea cu o organizare potrivită se poate reduce volumul de cod scris pentru tratarea excepțiilor din aplicație

- Numărul de clauze catch nu ar trebui sa crească odată cu evoluția programului.
- Clasele folosite pentru excepții trebuie organizate în ierarhii de clase, asta permite reducerea numărului de clauze catch.
- Un grup de tipuri de excepții poate fi tratat uniform, daca între aceste tipuri există relația de moștenire.
- Folosind ierarhiile de excepții putem beneficia și de polimorfism

```
try {
      Product p = wh->addProduct(code, desc, price);
      cout << "product " << p.getDescription() << " added." << endl;
} catch (WarehouseException& ex) {
      cout << "Error on store:" << ex.getMsg() << endl;
}
```

`catch (WarehouseException& ex) – se executa daca apare excepția WarehouseException sau clase derivate din WarehouseException (ValidatorException, RepositoryException)`

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;
}
```

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, MSVS)
- 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: http://qt-project.org/

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 compilare

- **Qt Eclipse integration** – plugin eclipse care ajută la crearea de aplicații QT folosind eclipse

Instalare QT, eclipse plugin

Este necesar: Eclipse CDC (MinGW) funcțional

1) Download /instalare Qt Library

http://qt-project.org/downloads → for windows **Qt libraries 4.8.1 for Windows (minGW 4.4, 319 MB)**

Conține:

- Qt SDK (library)
- Qt Designer, Assistant, Linguist, Samples and demos

2) Download / instalare plugin eclipse pentru Qt

Oferă:

- Wizards pentru a crea proiecte Qt
- Configurare build automată
- Un editor QT integrat în eclipse (QT Designer)

OBS:

- pluginul nu funcționează cu Eclipse Indigo 64 bit (folosiți 32 bit Indigo or Helios)
- Qt installer poate da un mesaj warning la anumite versiuni de MinGW (diferit de 3.13).Dacă aveți versiune mai nouă ignorați mesajul și continuați instalarea.

Qt Hello World

Creați un QT Eclipse Project: File → New → Qt GUI Project

Adăugați în main:

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

Project → Build Project

Rulați aplicația

QApplication

Clasa QApplication gestioneaza fluxul de evenimente și setarile 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 sitemul 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();
```

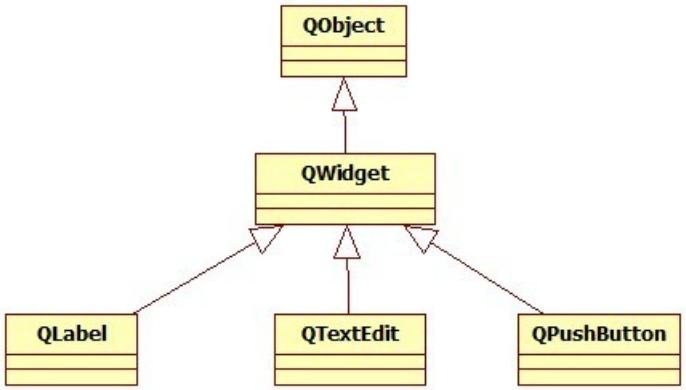
- pornește procesarea de evenimente din QApplication (event loop). În timp ce rulează aplicația evenimente sunt generate și trimise catre componentele grafice.

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 defapt 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 deobicei ca și o etichetă pentru o componentă interactivă. For this use QLabel oferă mecanism de mnemonic, o scurtătură prin care se setează folcusul 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.

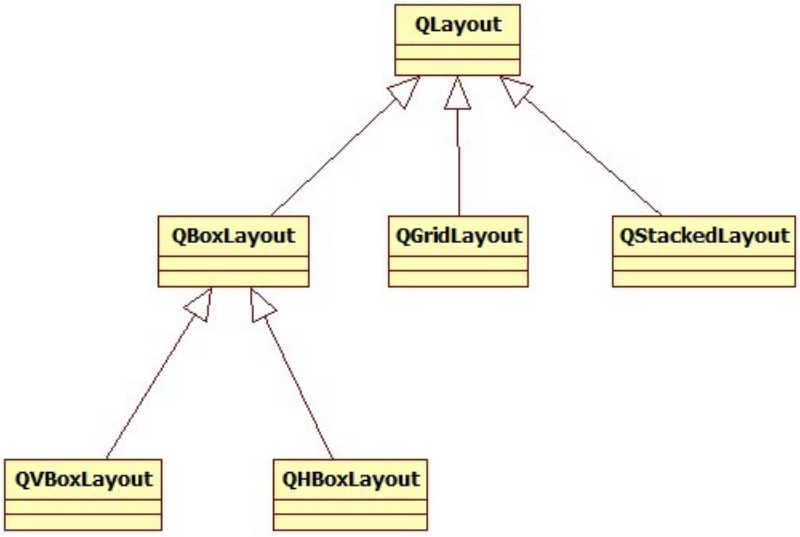
QListWidget

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 compinente 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:

- instanțiem 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 reanjarea 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 adapteaza componenta pentru a evita trunchiera 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 ramân 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://qt-project.org/doc/qt-4.8/
- Pentru orice clasă găsiți descrieri detaliate pentru metode, atribute, semnale , sloturi