АКАДЕМИЈА ТЕХНИЧКО-УМЕТНИЧКИХ СТРУКОВНИХ СТУДИЈА БЕОГРАД

ОДСЕК ВИСОКА ШКОЛА ЕЛЕКТРОТЕХНИКЕ И РАЧУНАРСТВА

Ђорђе Рмуш

Имплементација Ц++ библиотеке за графички интерфејс

- завршни рад -

Београд, октобар 2022.

Кандидат: **Ђорђе Рмуш**

Број индекса: **РТ-14/19**

Студијски програм: **Рачунарска техника**

Тема: **Имплементација Ц++ библиотеке за графички интерфејс**

Основни задаци:

1. Опис функционалности оригиналне Ц++ библиотеке за графички интерфејс

2. Имплементација Ц++ библиотеке за графички интерфејс

3. Пример коришћења имплементиране библиотеке

Ментор:

Београд, октобар 2022 године.

Др. Перица Штрбац ВИШЕР

РЕЗИМЕ:

Главни циљ овог рада је да читачу објасни рад и начин коришћења моје *C*$^{++}$ библиотеке. Прво ћу представити структуралну анализу, а онда ћемо прећи преко упутства коришћења библиотеке – отварање прозора, елемената интерфејса, и цртања по њима, кроз специфичних имплементација њених елемената и образложења одлука о дизајну и структури овог пројекта. На крају, спровешћу вас кроз изворни код демонстрационе апликације за наручивање пице.

Кључне речи: *библиотека, имплементација, прозор, графички интерфејс*

ABSTRACT:

The main goal of this paper is to provide the reader with an explanation of the implementation and usage of my C++ library. First, I will present the structural analysis of this project. We will then go over the instructions on using this library, and the specific implementations of its elements; answer why I went with the decisions I did in the design and structure of this project. Finally, I will lead you through the code of the demo app I made for ordering pizzas.

Key words: *library, implementation, window, graphical interface*

САДРЖАЈ:

1. УВОД

У поглављу "Структура пројекта" прво прелазим преко организације фајлова овог пројета – где стоје *header,* а где *source* фајлови, у које под пројекте је библиотека подељена и све остало у вези именовања фајлова и директоријума. Онда прелазимо преко структуре елемената библиотеке – односно *namespace*-ова, класа и функција; како се именују, и где налазе.

У трећем поглављу "Примена библиотеке" прелазим преко практичног коришћења елемената библиотеке – прво контрола: њихово стварање, подешавање и додавање у прозор; а онда и преко тога како цртати по контролама помоћу класе за цртање и њених помоћних класа.

Четврто поглавље је намењено кратком опису имплементације, која би повезала елементе библиотеке са коришћеним технологијама.

Пето поглавље "Демо апликација" прелази преко имплементације апликације која демонстрира рад библиотеке. Ово би требало да повеже све узорке кода које сте видели током читања трећег поглавља.

2. СТРУКТУРА ПРОЈЕКТА

2.1. ОРГАНИЗАЦИЈА ФАЈЛОВА

Овај пројекат је урађен из више делова (под-пројеката). Главно решење (*solution*) је под називом *cauldron*. Под њим се налазе три пројекта везана за библиотеку и један за демонстрацију библиотеке. Три пројекта везана за библиотеку у корену држи све своје **.h* фајлове - међу њима, и "*_include.h*", који ће укључити све *header* фајлове пројекта. Поред њих, у корену постоји директоријум "*_src*" у коме стоје имплементације(*.cpp). Сви пројекти користе *ISO C++17* стандард, а три пројекта библиотеке су типа статичне библиотеке.

cauldron-common

У овом пројекту стоје сви основни елементи пројекта опште намене. Сви ови елементи су независни од оперативног система, односно користе само стандардну библиотеку при имплементацији.

садржај:

1. *_include.h*
2. *primitives.h,*
3. *vectors.h, vectors.ostream.h, vectors.types.h,*
4. *bounds.h, bounds.ostream.h,*
5. *color.h,*
6. *callback.h,*
7. *eventData.h,*
8. *math.h,*
9. *random.h,*
10. *macro.autoOperator.h, macro.coreManip.h*

cauldron-platform

Овде се налазе сви елементи чија имплементација зависи од оперативног система, директно или индиректно. Овај пројекат се ослања на *cauldron-common*.

садржај:

1. *_include.h*
2. *clock.h,*
3. *key.h,*
4. *thread.h,*
5. *message.h,*
6. *mutex.h,*
7. *semaphore.h,*
8. *task.h*

cauldron-gui

Овај пројекат ће бити централни део овог рада, јер је он садржи елементе који су задужени за графички интерфејс. Овај пројекат се ослања на *cauldron-common* и *cauldron-platform*.

садржај:

1. *_include.h*
2. *orientation.h,*
3. *paint.h,*
4. *theme.h,*
5. *defaults.h*
6. *control.h,*
7. *anchoredControl.h,*
8. *label.h,*
9. *button.h,*
10. *fillbar.h,*
11. *checkInput.h,*
12. *textInput.h*
13. *group.h,*
14. *picture.h,*
15. *scrollBar.h*
16. *window.h*

cauldron-demo

У овом пројекту налази се аликација која демонстрира рад ове библиотеке. Ослања се на пројекте библиотеке.

садржај:

1. *pizzaForm.h*
2. *main.cpp*

2.2.ОРГАНИЗАЦИЈА ЕЛЕМЕНАТА БИБЛИОТЕКЕ

cauldron-common/primitives.h

```
using u64   = unsigned long long;  using i64     = long long;
using u32   = unsigned long;       using i32     = long;
using u16   = unsigned short;      using i16     = short;
using u8    = unsigned char;       using i8      = char;
using f64   = double;              using f32     = float;
using ch    = char;                using wch     = wchar_t;
using str   = char*;               using wstr    = wchar_t*;
using cstr  = const char*;         using cwstr   = const wchar_t*;
```

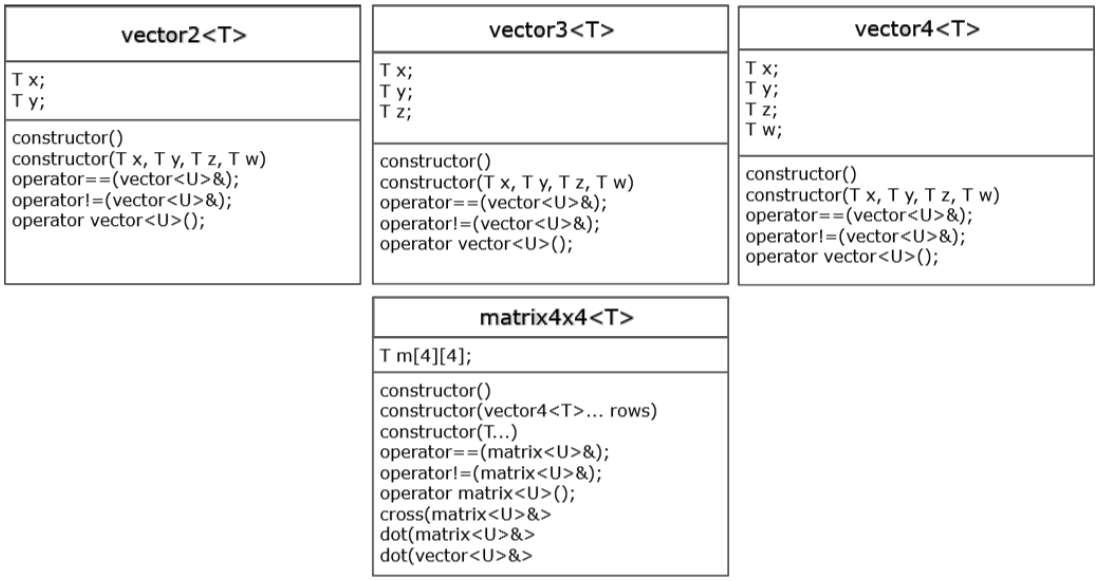
Овде су дефинисани надимци за све основне типове података. Они ће се даље користити у свим осталим под-пројектима. Ово радимо јер су имена неких примитивних типова података предуга и непогодна за писање/читање(нпр. *unsigned long long*).

cauldron-common/vectors.h,
cauldron-common/vectors.ostream.h,
cauldron-common/vectors.types.h

Овде су дефинисани векторски типови података (*vectors.h*), оператори за исписивање на главни излаз (*vectors.ostream.h*) и надимци за векторе примитивних типова (*vectors.types.h*). Вектори су шаблонске класе – узимају један параметар који је тип елемената вектора. Вектори и долазе у три варијанте - дводимензионални, тродимензионални и четвородимензионални. Сваки вектор има аритметичке операторе, операторе за поређење и претварање у друге топове вектора(различитих типова и

димензионалности). Поред вектора садржи и матрицу димензија 4 са 4, са свим очекиваним операторима. Њихов садржај можете видети на слици 2.1.

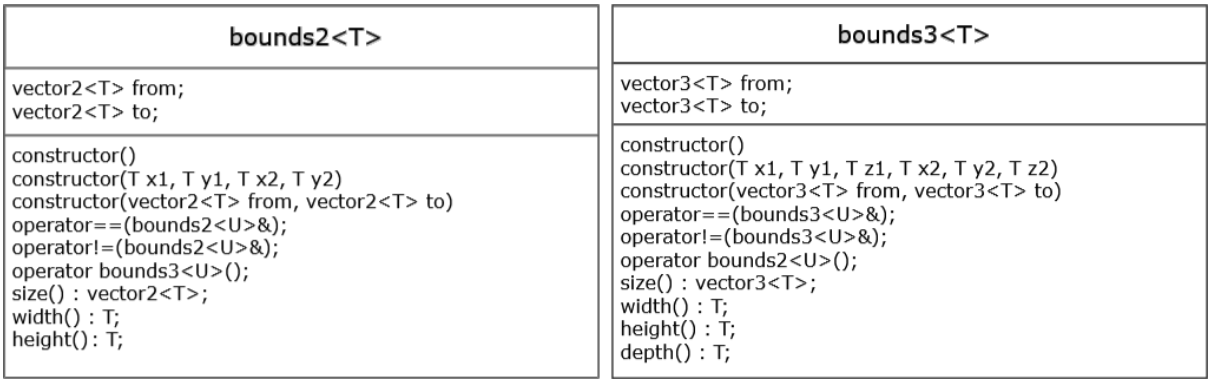
Надимци прате конвенцију називања *v#T*, где # означава број димензија који вектор обухвата, а *T* је надимак за основни тип података нпр. *v4f64* – 4Д вектор *double* елемената.



Слика 2.1 Векторски типови

`cauldron-common/bounds.h`
`cauldron-common/bounds.ostream.h`

Границе су шаблонска класа, са једним параметром који представља тип елемента вектора граница. Она садржи два вектора који представљају почетну и крајњу тачку. Границе садрже операторе поређења и претварања, и методе за добављање величине граница, односно висине, ширине и (ако је 3Д граница) дубине. Испод, на слици 2.2 можете видети њихов садржај.



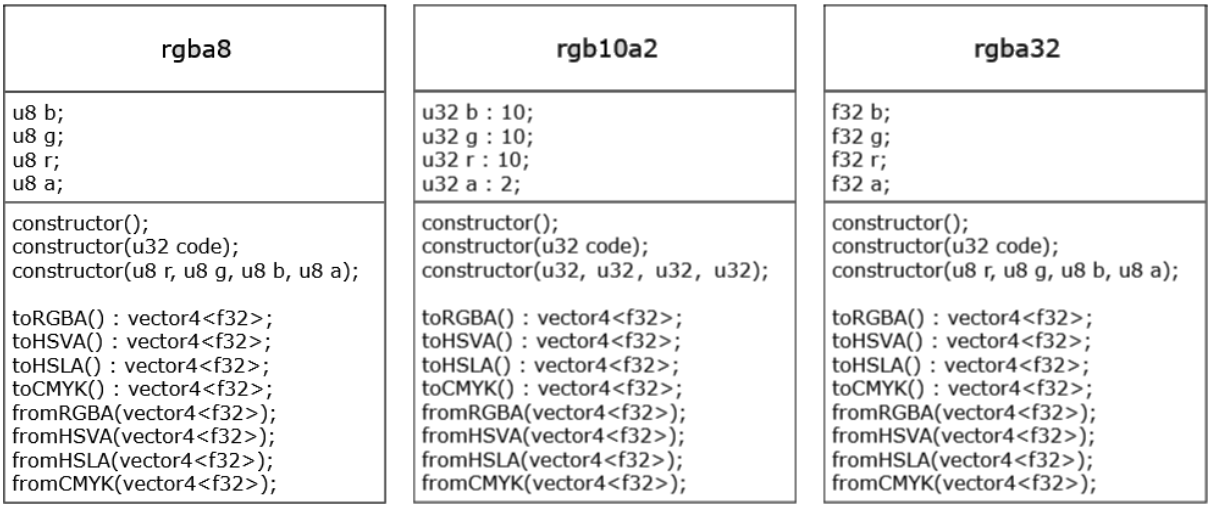
Слика 2.2 Класе 2Д и 3Д граница

`cauldron-common/color.h`

Боје су дефинисане у три формата - *rgba8* са 8 бита по каналу(укупно 32 бита),

rgb10a2 са 10 бита по каналу боје, и само 2 бита за алфа канал(укупно 32 бита), и rgba32 са 32 бита по каналу(укупно 128 бита, канали су бројеви са покретним зарезом).

Садрже операторе за рад са различитим форматима: *RGBA, HSVA, HSLA, CMYK* односно за конверзију из датхи формата и у дате формате. Испод, на слици 2.3 можете видети њихову структуру.



Слика 2.3 Класе боја

`cauldron-common/callback.h`

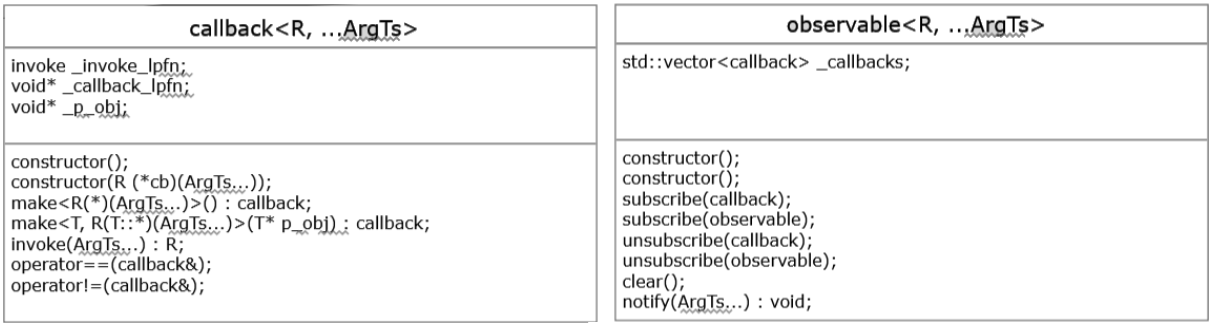
Шаблонска класа *callback<R, ...ArgTs>* има сличну улогу *std::function* из стандардне библиотеке. Исто јој дате функцију, или објекат и његову методу, да бисте касније могли да је позовете посредно кроз инстанцу ове класе. Једини разлог зашто сам морао да имплементирам сопствену верзију *std::function*, је јер не подржава операторе поређења. Нове инстанце *callback* објекта правимо са статичком функцијом *make<fn>()* за статичке функције, или *make<T, T::*fn>(T*)* за чланске функције. Да бисмо позвали повезану функцију користимо *R invoke(Args…)*.

```
void opAdd(int lhs, int rhs) { return lhs + rhs; }
callback<void, int, int> cb = callback<void, int, int>::make<printAdd>();
int result = cb.invoke(5, 6); // ‘result’ je 11
```

Шаблонска класа *observable<T, ...ArgTs>* служи да подржава *observer pattern*. он је колекција *callback<T, ...ArgTs>* којој можете додавати нове елементе, уклањати елементе који су вам познати, и позвати све елементе редом са прослеђеним аргументима. Да би додали прислушкивача користимо *subscribe(const callback<...>&)*, а да би их одстранили користимо *unsubscribe(const callback<...>&).* Пошто овај објекат позива много различитих функција, нема смисла враћати коју год повратну вредност поједне функције враћају - Зато функција за позивање свих садржаних функција не враћа ништа - *void notify(Args…)*. Испод на слици 2.4 можете видети састав ове две шаблонске класе.

```
void printAdd(int lhs, int rhs) { return std::cout << (lhs + rhs) << ‘\n’; }
void printSub(int lhs, int rhs) { return std::cout << (lhs - rhs) << ‘\n’; }
void printMul(int lhs, int rhs) { return std::cout << (lhs * rhs) << ‘\n’; }
void printDiv(int lhs, int rhs) { return std::cout << (lhs / rhs) << ‘\n’; }
observable<void, int, int> obs;
obs.subscribe(printAdd); // printAdd pretvorena u callback<void, int, int>
```

```
obs.subscribe(printSub); // printSub pretvorena u callback<void, int, int>
obs.subscribe(printMul); // printMul pretvorena u callback<void, int, int>
obs.subscribe(printDiv); // printDiv pretvorena u callback<void, int, int>
obs.notify(10, 5);
// Izlaz: “15\n5\n50\n2\n”;
```



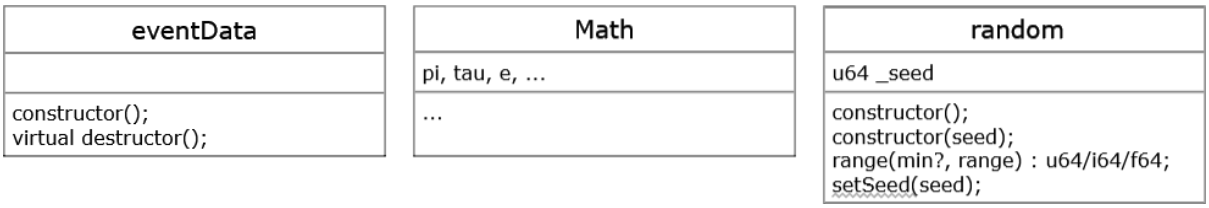
Слика 2.4 Класе callback и observable

cauldron-common/eventData.h,
cauldron-common/math.h,
cauldron-common/random.h

eventData класа служи за пренос података о догађајима. Сама по себи не ради ништа - служи само да би се од ње наследило. Погледајте слику 2.5.

math класа садржи релевантне функције(ограничење вредности, интерполација, инверзна интерполација, ремапирање вредности, итд.). Поред тога садржи константе пи, тау, и е. Погледајте слику 2.5.

random класа служи са стварање псеудо насумичних бројева(*u64, i64, f64*) у задатим опсезима. Погледајте слику 2.5.



Слика 2.5 Класе *eventData, math, i random*

cauldron-common/*macro.autoOperator.h*,
cauldron-common/*macro.coreManip.h*

Први наведени фајл садржи макро-е за аутоматскп дефинисање *inline* битских оператора за енум типове. оператори су и, или, екскузивно или, и инверзија.

```
#define INLINE_BITWISE_AND(ENUM_T)...
#define INLINE_BITWISE_OR(ENUM_T)...
#define INLINE_BITWISE_XOR(ENUM_T)...
#define INLINE_BITWISE_INVERT(ENUM_T)...
```

Други фајл садржи макро-е за рад са објектима који користе *pimpl idiomом*. Аутоматски дефинише *inline* методе за добављање језгра, поређења по језгру, премештања и замењивања језгра.

```
#define INLINE_CORE_GET(CORE_T, VAR_ID)...
#define INLINE_CORE_CMP(T, CORE_T, VAR_ID)...
```

```
#define INLINE_CORE_CMP(T, CORE_T, VAR_ID)...
#define INLINE_CORE_MOVE(T, CORE_T, VAR_ID)...
```

cauldron-platform/*clock.h*

Садржи класу *clock* - статичка класа са две функције:

```
static u64 getFrequency(); // Vraca frekvenciju sistemskog sata
      static u64 getTimeStamp(); // Vraca vrednost sistemskog sata
```

cauldron-platform/key.h

Садржи класу *key*. Инстанца ове класе идентификују тастер на тастатури или мишу. Ради по *pimpl idiom-u.* Има операторе поређења по језгру, и добављања језгра.

cauldron-platform/thread.h

Садржи класу *thread*. Уводи све очекиване операторе – методе за спајање, упите о томе да ли тренутно ради, и да ли може да се споји са позивајућим тредом. Садржи *opaque* тип за идентификацију тредова - он је потребан за слање порука датом треду.

Треад објекти се праве или празним конструктором – који не стартује тред, или са конструктором који узима процедуру (*void (*) (T*)*) и аргумент типа *T**. Накнадно се може стартовати помоћу методе *start(void(*)(T*), T*).*

cauldron-platform/message.h

Садржи класу *message*. Користи се за читање, обраду и слање порука тредовима. Конструише се само помоћу празног конструктора. Поруку корисник не може директно читати, односно погледати у њен садржај, може само да је обради.

За учитавање порука треда користимо дате методе, прва чека на поруку ако их нема у реду, друга не чека, него враћа одмах из функције:

```
bool awaitPull();         // True ako je bilo poruka za preuzimanje
bool immediatePull();     // True ako je bilo poruka za preuzimanje
```

Обрада методе за поруке које су послане post методом подразумева извршење прослеђене функције са прослеђеним аргументом. Поред њих системске поруке су обрађене на начин специфичан платформи, односно корисник не брине за имплементацију њихове обраде. За обраду поруке користимо дату наредну:

```
void dispatch();
```

За слање поруке користимо дату функцију:

```
template<class arg_t>
static bool post(thread::id* target, void(*procedure)(arg_t*), arg_t* arg)
```

За сада једину поруку коју можемо слати је захтев за извршење процедуре - и не постоји начин да погледамо садржај поруке - само да је обрадимо.

cauldron-platform/*mutex.h*,
cauldron-platform/*semaphore.h*

Класе *semaphore* и *mutex* раде на очекиван начин – три функције, за преузимање ресурса, за тренутно преузимање ресурса и за отпуштање ресурса. При конструкцији семафор узима тренутни број ресурса и максимални број ресурса, а мутекс узима само булеан, који одређује да ли је аутоматски усвојен од стране треада који га је створио. Дате функције су:

За чекање и преузимање ресурса користимо:
```
bool acquire();
```

Ако би желели да тренутно преузмемо ресурс али не ако морамо да чекамо:
```
bool tryAcquire();
```

Да отпустимо ресурс:
```
bool release();
```

cauldron-platfom/*task.h*

Класа *task<T>* је класа која се бави одлагањем посла на друге тредове. Она садржи виртуелну процедуру (која поредставља тај посао) коју ће циљани треад да изврши, и семафор који је одговоран чекање на извршење дате процедуре. Након чекања извршења процедуре, корисник може да приступи "одговору" у виду инстанце типе *T*. Ако је параметар *void*, онда она не враћа ништа.

Таскови су намењени за коришћење у раду са контролама. Ако бисмо желели да извршимо неку измену над њима, морамо то урадити са треда који их је створио. Тако да ако бисмо желели то да урадимо у неком другом треду(да ажурирамо стање нпр. Променимо проценат завршења неке операције која ради на другом треду), могли бисмо да пошаљемо поруку треду који је одговоран за дате контроле.

За подношење задатка треду:

```
void post(thread::id* threadIdentifier);
```

Да сачекамо резултат користимо једну од ове две методе:

```
void awaitCompletion();   // za T==void
T& awaitResult()          // za T!=void
```

Методу коју треба да надјачамо да бисмо дефинисали процедуру задатка:

```
virtual void procedure();              // za T==void
virtual void procedure(T& output);     // za T!=void
```

cauldron-gui/orientation.h

Садржи енум *orientation* три члана: *horizontal* - за хоризонталну ориентацију, *vertical* - за вертикалну ориентацију, и *automatic* за аутоматско ориентисање, најчешће према димензијама.

cauldron-gui/paint.h

Садржи класу *paint* коју користимо да цртамо по контролама. Садржи друге класе попут четкица, оловки, фонтова и слика који се користе при цртању. О коришћењу ове класе погледајте поглавље 3.5.

cauldron-gui/theme.h

Садржи 15 четкица за цртање - све стандардне контроле користе теме ради лакшег подешавања изгледа.

cauldron-gui/defaults.h

Садржи стандардне теме и фонтове - они се користе у случају да контроли не уделите тему/фонт. Од тема садржи Светлу и тамну, од којих је једна изабрана за стандардну, и пар фонтова за главан текст, и за *hint* текст.

cauldron-gui/control.h

Основна контрола, на којој се заснивају све остале. Поседује многа својства које која можете читати/подешавати. Подржава догађаје које се могу пратити (користи *observer* модел).

cauldron-gui/anchoredControl.h

Контрола са уграђеним аутоматским подешавањем величине према усидрењу(*anchor*) и према маргинама(*margins, offset*). То дозвољава да се позиција контроле подешава и процентуално, и по пикселима.

cauldron-gui/label.h,
cauldron-gui/button.h,
cauldron-gui/fillbar.h,
cauldron-gui/checkInput.h,
cauldron-gui/textInput.h
cauldron-gui/group.h,
cauldron-gui/picture.h,
cauldron-gui/scrollBar.h,
cauldron-gui/window.h

Садрже контроле према именовању фајла у коме се налазе. Преко њих дубље прелазим у наредном поглављу.

3. ПРИМЕНА БИБЛИОТЕКЕ

3.1. ГРАФИЧКИ ЕЛЕМЕНТИ БИБЛИОТЕКЕ

Пре него што говоримо о специфичним елементима, морамо да пређемо преко садржаја основне контроле(табела 3.1).

Табела 3.1 Својства контроле

својство	**уписивање** (*setter*)	**читање** (*getter*)
родитељ контроле	`adopt(x)`	`getParent()`
границе контроле	`setBounds(x)`	`getBounds(), getClientSize()`
стил фокуса контроле	`setFocusStyle(x)`	`getFocusStyle()`
контрола је валидна	`terminate()`	`isValid()`
контрола је активна	`setActive(x)`	`isActive()`
контрола је фокусирана	`setFocused(x)`	`isFocused()`
контрола је укључена	`setEnabled(x)`	`isEnabled()`
курсор на контроли	`N/A`	`isCursorOver()`
користи дупли бафер	`setDoubleBuffered(x)`	`isDoubleBuffered()`

Свака контрола је слободна да надјача сваку од *setter* метода према својим потребама. Надјачавање *getter* метода није дозвољено.

Сил фокуса контроле (*control::focusStyle*) може бити једна од: *focusable, unfocusable,* и *defer_to_child*. Стил *focusable* значи да контрола може да преузме фокус (нпр, на клик) - то значи да ће њој да се преносе притисци и отпуштања тастера(само контрола са фокусом их емитује). *Unfocusable* значи да не може да преузме фокус - ни са *setFocus*(x). *Defer_to_child* значи да ће се фокус који је намењен за њу пренети на прво дете које прихвата фокус - ако такво не постоји, фокус је одбачен.

Свака контрола емитује догађаје. Ослушкивање ових догађаја вршимо преко *cauldron::common::observable<void, control&, ...>*. Као што можемо видети из потписа, функције које би да ослушкују догађаје, немају поврату вредност, а узимају референцу на *control* објекат који је емитовао догађај, и додатне податке о догађају(обично инстанце класе која наслеђује од *cauldron::common::eventData*). Поред основних догађаја, контрола може емитовати и догађаје специфичних себи(нпр. ради валидације улаза). Ево листа догађаја основне контроле(табела 3.2):

Табела 3.2 Основни догађаји контроле

догађај	**податци о догађају**	**ослушкивање**
позван по захтеву за затварања контроле	`closeData`	`onClose()`
позван пре терминације контроле	`terminateData`	`onTerminate()`
позван по активацији контроле	`activateData`	`onActivate()`
позван по деактивацији контроле	`deactivateData`	`onDeactivate()`
позван по добијању фокуса	`gainFocusData`	`onGainFocus()`
позван по губљењу фокуса	`loseFocusData`	`onLoseFocus()`
позван по укључивању	`enableData`	`onEnabled()`
позван по искључивању	`disableData`	`onDisabled()`
позван по промени родитеља(дете емитује)	`changeParentData`	`onChangeParent()`
позван по усвајању(нови родитељ емитује)	`adoptData`	`onAdopt()`
позван по одрицању(стари родитељ емит.)	`disownData`	`onDisown()`
позван по померању контроле	`moveData`	`onMove()`
позван по корисничком померању контроле	`movingData`	`onMoving()`
позван по мењању величине	`sizeData`	`onSize()`
позван по корисничком мењању величине	`sizingData`	`onSizing()`
позван по притиску тастера	`keyDownData`	`onKeyDown()`
позван по отпуштању тастера	`keyUpData`	`onKeyUp()`
позван по уносу карактера	`characterData`	`onCharacter()`
позван по уплазу курсора у контролу	`cursorEnterData`	`onCursorEnter()`
позван по померању курсора по контроли	`cursorMoveData`	`onCursorHover()`
позван по излазу курсора из контролу	`cursorLeaveData`	`onCursorLeave()`
позван по померању точкића миша	`scrollData`	`onScroll()`
позван по притиску тастера миша	`mouseDownData`	`onMouseDown()`
позван по отпуштању тастера миша	`mouseUpData`	`onMouseUp()`
позван по дуплом клику	`doubleClickData`	`onDoubleClick()`
позван по потреби за цртње контроле	`paintData`	`onPaint()`

Пример рада са контролама (лабела):

```
cauldron::gui::label label;
parent.adopt(label);
label.setAnchor({ 0.0f, 0.f, 1.f, 1.f });
label.setOffset({ 5, 5, -5, -5 });
label.setText(L"Hello from gui::label!");
label.setVerticalAlignment(gui::paint::alignment::center);
label.setHorizontalAlignment(gui::paint::alignment::center);
label.setTheme(theme);
label.setFont(font);
label.setVisible(true);
```

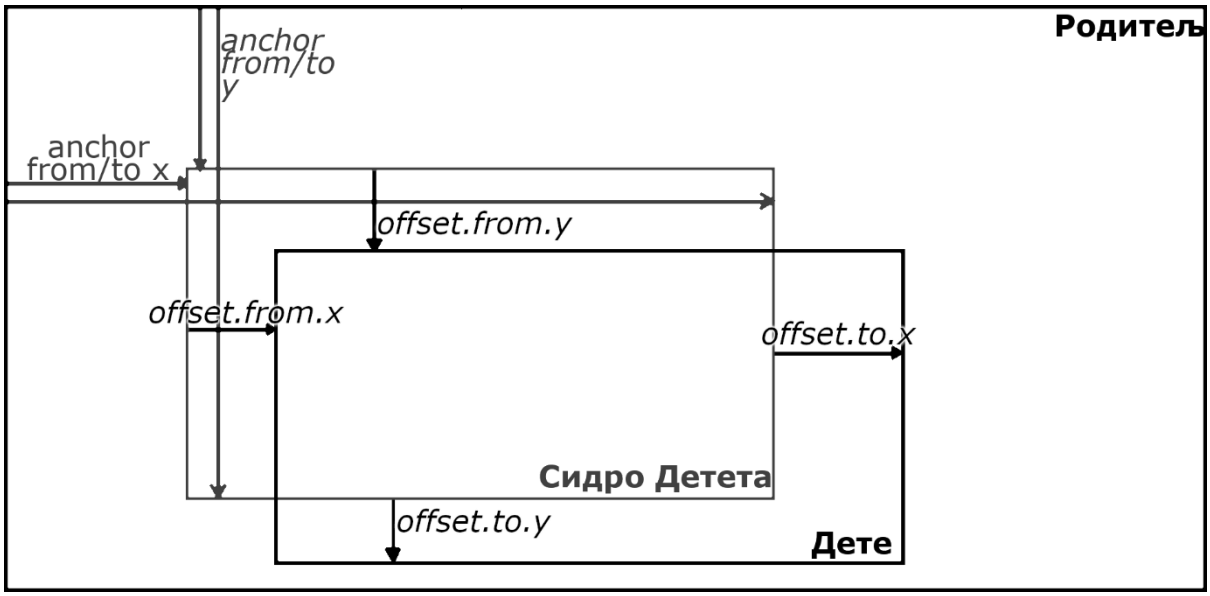
Одавде видимо редослед рада са контролама:

1. Усвоји га родитељу, ако је планирано да то буде дете контрола,
2. Подеси својства контроле по жељи,
3. Подеси му стање (*setState*) na *state::normal*. da bi ga prikazali.

Наравно, ако бисмо желели да ослушкујемо *onChangedParent* догађај, морамо да се прислушкујемо пре *parent.adopt(...);*

Наредна најбитнија контрола је *anchoredControl*. Она увои два нова својства - усидрење(anchor) и маргине, односно офсет(*margins, offset* - исто својство, само зависи од интерпретације).

Усидрење представља процентуални помак у односу на *klijentne granice* родитеља. Оно је састављено од четири елемента - једно за сваку ивицу контроле. Маргине, односно офсет се рачунају у пикселима, и примењују се у односу на четвороугао добијен након примене усидрења. Пример слика 3.3.



Слика 3.3 Визуелизација усидрене контроле

На слици можете да видите начин рада сидра - ако желимо да покријемо целу површину родитеља, онда ће сидро се протезати од (0%, 0%) до (100%, 100%) - где је 0% = 0.0f, а 100% = 1.0f. након тога, ако бисмо желели да увучемо границе детета за пет пиксела, само морамо да му дамо маргине (5, 5, 5, 5) односно офсет од (5, 5, -5, -5). Ако

желелите да закачите дете за специфични ћошак или страницу, можемо користити вредности за сидро на табели 3.4:

Табела 3.4 Честе вредности сидра

ивица	вредност сидра	ћошак	вредност сидра
горе	(0, 0, 1, 0)	горе лево	(0, 0, 0, 0)
доле	(0, 1, 1, 1)	доле лево	(0, 1, 0, 1)
лево	(0, 0, 0, 1)	горе десно	(1, 0, 1, 0)
десно	(1, 0, 1, 1)	доле десно	(1, 1, 1, 1)

Офсет можемо превести у маргине тако што ћемо крајње границе помножити са -1. То значи за офсет *(x1, y1, x2, y2)* одговарајуће маргине су *(x1, y1, -x2, -y2)*. Исто радимо да би маргине пребацили назад у офсет.

Функције којима би смо управљали овим својствима можете погледати на наредној табели 3.5.

Табела 3.5 Својства усидрене контроле

име својства	уписивање	Читање

`сидро контроле`
`маргине контроле`
`офсет контроле`
	`setAnchor(x)`
	`setMargin(x)`
	`setOffset(x)`
		`getAnchor()`
		`getMargin()`
		`getOffset()`

window контрола је намењена за прављење *top-level* прозора. Корисници ове класе морају да додају посебни *callback* за *onPaint* догађај. Поседује методу *adopt(control*)* за усвајање елемената. Резултат наредног кода види се на слици 3.6.

```
gui::window::style window_style =
      gui::window::style::bordered |
      gui::window::style::resizable |
      gui::window::style::captioned |
      gui::window::style::minimize_button |
      gui::window::style::maximize_button;
bounds2<i32> window_bounds = { {200, 200}, {360, 240} };
onClose().subscribe(gui::control::terminateOnClose);
onPaint().subscribe(
      callback<void, control&, paintData&>::
      make<test, &test::paintForm>(this));
setStyle(window_style);
setBounds(window_bounds);
setCaption(L"Test form");
setFocusStyle(gui::control::focusStyle::unfocusable);
setVisible(true);
```

Слика 3.6 Изглед *window* контроле

Стање прозора (*window::state*) може бити једно од: *hidden, normal, minimized,* и *maximized.* последња два су валидна само када се говори о *top level* прозорима.

Сил прозора (*window::style*) може бити комбинација: *bordered, captioned, resizable, minimize_button, maximize_button* и *child*. Опције саме себе описују. Већина су само валидна када се говори о *top level* прозорима. Опције *child* се сама додаје при усвајању.

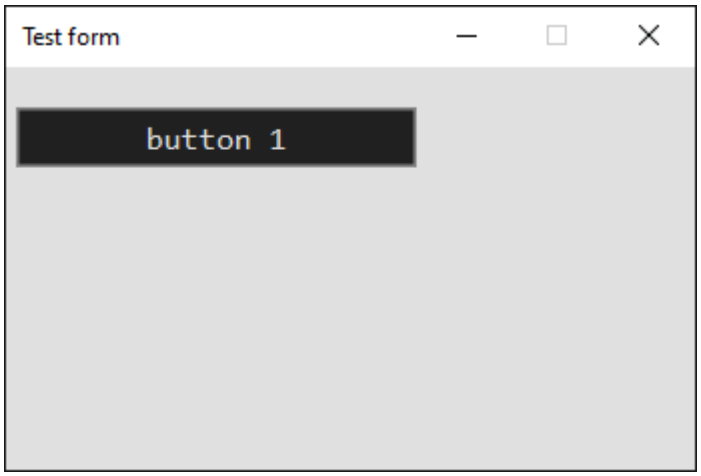
label контрола наслеђује од усидрене контроле, што значи да има аутоматско позиционирање. Поред тога, можемо јој променити текст, поравнање текста - вертикално и хоризонтално, променити тему и фонт. Пример коришћења ових својстава можете видети у исечку испод. Резултат наредног кода види се на слици 3.7.

```
parent.adopt(&label);
label.setAnchor({ 0.0f, 0.f, 1.f, 1.f });
label.setOffset({ 5, 5, -5, -5 });
label.setText(L"Hello from gui::label!");
label.setVerticalAlignment(gui::paint::alignment::center);
label.setHorizontalAlignment(gui::paint::alignment::center);
label.setTheme(theme);
label.setFont(font);
label.setVisible(true);
```

Слика 3.7 Изглед лабеле

button контрола наслеђује од усидрене контроле, што значи да има аутоматско позиционирање. Поред тога, можемо му променити текст, поравнање текста - вертикално и хоризонтално, тему и фонт. Дугме уводи још један догађај - *onClick()*, који се емитује сваки пут када се дугме кликне. То није еквивалентно догађајима *onMouseDown/onMouseUp*, јер се *onClick* емитује и када је дугме у фокусу, а корисник притисне размак и узима у обзир кретање(односно излазак) курсора са контроле. Пример коришћења ових својстава можете видети у исечку испод. Резултат наредног кода види се на слици 3.8.

```
void callbackFunction(control& control, eventData& e) {
      std::cout << “CLICKED\n”;
}
...
parent.adopt(&button);
button.onClick().subscribe(callbackFunction);
button.setAnchor({ 0.0f, 0.1f, 0.0f, 0.1f });
button.setOffset({ { 5, 0}, {200, 30 } });
button.setText(L"button 1");
button.setTheme(theme);
button.setFont(font);
button.setVisible(true);
```



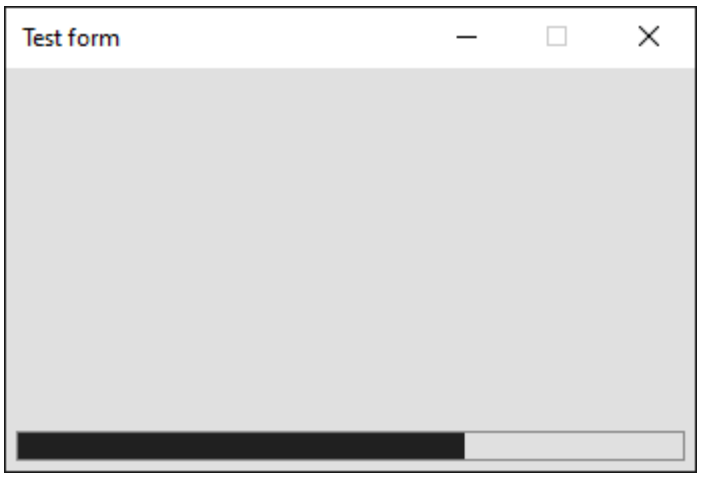
Слика 3.8 Изглед дугмета

Сваки пут када кликнемо на дугме, у испису у конзоли би требало да изађе *“CLICKED”* - Осим ако је дугме искључено.

fillbar елемент служи за обавештење корисника о проценту готовости одређеног процеса. Сходно томе, поседује функције за добављање и подешавање датог процента, као и догађај који се емитује по промени процента. Када се вредност(проценат) подеси на 0.0, елемент је потпуно празан. Са вредношћу 1.0, елемент је потпуно попуњен - и опционо фарбан као да је фокусиран(за подешавање користимо *setPaintAsFocusedWhenFull*, односно *isPaintedAsFocusedWhenFull* за читање). Ова контрола се може преоријентисати(*setOrientation*). Резултат наредног кода види се на слици 3.9.

```
parent.adopt(&fillbar);
fillbar.setAnchor({ 0.0f, 1.f, 1.f, 1.f });
fillbar.setOffset({ 5, -20, -5, -5 });
```

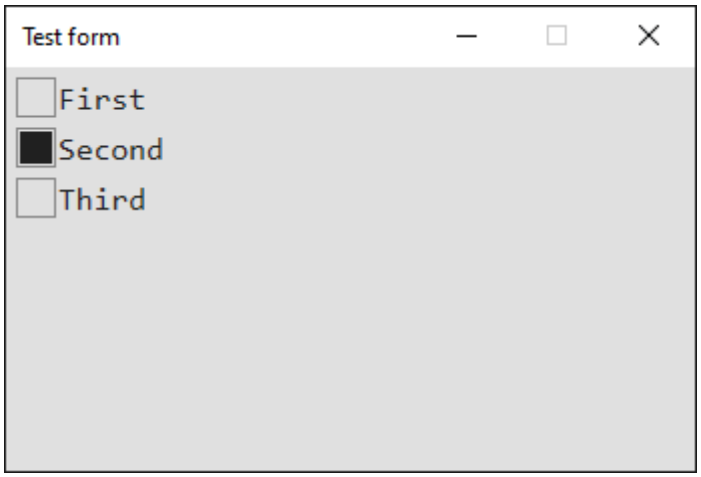
```
fillbar.setValue(0.67f);
fillbar.setVisible(true);
```



Слика 3.9 Изглед филбар елемента

checkInput елемент служи за унос тачно/нетачно вредности. Поседује методе за подешавање текста (*getText() / setText(const wstring&)*), методе за читање и уписивање вредности (*getValue() / setValue(bool)*) и догађај *onValueChanged* који се емитује сваки пут када је вредност елемента промењена. Можемо да учинимо да се понаша као радио дугме, тако што ћемо му уделити групу(*std::shared_ptr<std::vector<checkInput*>>*) помоћу методе *setRadioGroup*. Резултат наредног кода види се на слици 3.10.

```
const wchar_t* text[3] = { L"First", L"Second", L"Third" };
for (long i = 0; i < 3; i++) {
	parent.adopt(check[i]);
	check[i].setValue(i & 1);
	check[i].setAnchor({ 0, 0, 1, 0});
	check[i].setText(text[i]);
	check[i].setOffset({ 5.f, 5.f + i * 25, -5.f, 25.f + i * 25 });
	check[i].setVisible(true);
}
```



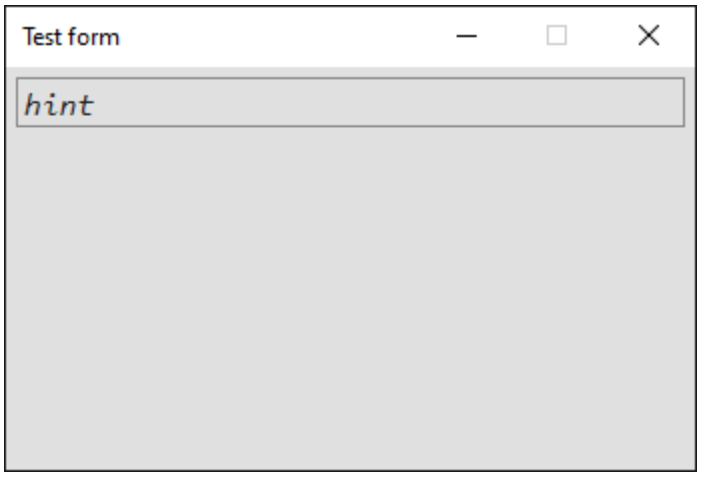
Слика 3.10 Изглед *checkInput* елемента

textInput елемент служи за унос текстуалних вредности. Поседује методе за подешавање текста (*getText() / setText(const wstring&)*) и *hint-a*(*getHint() / setHint(const wstring&)*) за подешавање селекције (*get/setSelect(const vector2<i32>&)*), методе за

подешавање фонтова(set/get Text/Hint Font(shared_ptr<font>)), и догађај *onValidate* који се емитује сваки пут када је текст измењен. Сви валидатори(осматрачи догађаја) могу да откажу измену текста.

Hint је текст који се види уместо обичног текста, ако је текст празан. *Select* је опсег карактера које је корисник обележио, односно индекс курсора. Резултат наредног кода види се на слици 3.11.

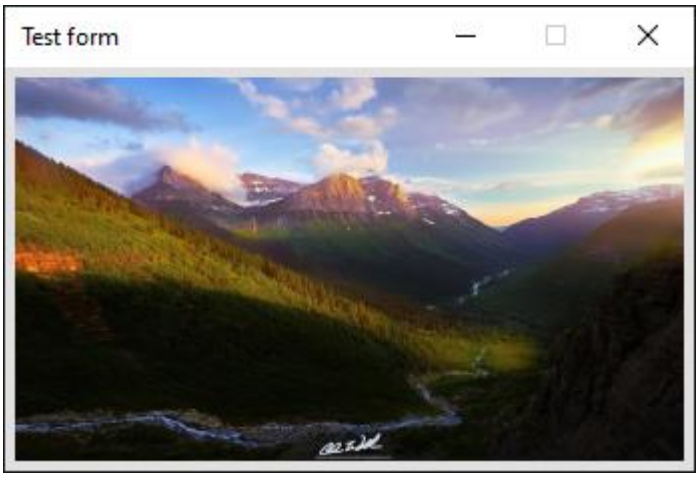
```
parent.adopt(textInput);
textInput.setAnchor({ 0, 0, 1, 0});
textInput.setHint(L"hint");
textInput.setText(L"");
textInput.setOffset({ 5.f, 5.f, -5.0f, 30.0f });
textInput.setVisible(true);
```



Слика 3.11 Изглед *textInput* елемента

picture елемент служи цртање слика на екран. Поседује методе за подешавање текста (*getImage() / setImage(shared_ptr<paint::image>)* и за начин исцртавања (*get/setMode(picture::mode)*). Мод исцртавања одређује да ли ће се слика растегнути преко целе контроле (*stretch*) што мења односе ширине и висине слике, умањити слику тако да стане цела без дисторција (*fit*) и исећи делове који би вирели са контроле(*clip*). Резултат наредног кода види се на слици 3.12.

```
parent.adopt(picture);
picture.setAnchor({ 0, 0, 1, 1});
picture.setOffset({ 5.f, 5.f, -5.0f, -5.0f });
picture.setMode(gui::picture::mode::stretch);
picture.setImage(
	std::make_shared<gui::paint::image>(
		L"C:\\Users\\Admin\\Desktop\\img.png"));
picture.setVisible(true);
```



Слика 3.12 Изглед *picture* елемента

scrollBar елемент служи скроловању контрола. Садржи три битне вредности: вредност, односно позицију дршке(*get/setValue(f64)*); величину дршке - у односу на целу дужину контроле (get/setHandleSize(f64)), и корак - померај по ротацији точкића(*get/setStep(f64)*). Уводи нови догађај - *onValueChanged* која се емитује по померају дршке(односно промене вредности). Резултат наредног кода види се на слици 3.13.

```
parent.adopt(scrollBar);
scrollBar.setAnchor({ 0, 0, 1, 0});
scrollBar.setOffset({ 5.f, 5.f, -5.0f, 20.0f });
scrollBar.setHandleSize(0.1f);
scrollBar.setValue(0.33);
scrollBar.setStep(0.5);
scrollBar.setOrientation(orientation::automatic);
scrollBar.setVisible(true);
```

Слика 3.13 Изглед *scrollBar* елемента

`anchored`*Group* је контрола намењена за груписање контрола. Наслеђује од класе *group* и са тиме добија методу *adopt(control*)* за додавање контрола групи - њу можете да надјачате. По свему осталом се понаша нормално.

3.2.ЦРТАЊЕ ПО ГРАФИЧКИМ ЕЛЕМЕНТИМА

За цртање по графичким елементима користимо класу ***paint*** и њене поделементе.

- **paint**
  - **alignment**
  - **font**
    - **style**
  - **image**
  - **bitmap**
  - **brush**
  - **solidBrush**
  - **textureBrush**
  - **pen**

Поравнање текста (***alignment***) је енум тип са три вредности:

```
1. near      // levo, odnosno gore
2. center    // centrirano vertikalno ili horizontalno
3. far       // desno, odnosno dole
```

Стил фонта (***font::style***) је енум тип који одређује изглед текста. Стилови се могу комбиновати *bitwise* операторима. Има три вредности:

```
1. normal          // Normalan tekst
2. bold            // Podebljan tekst
3. italic          // Italicizovan tekst
4. strikeout       // Precrtan tekst
5. underline       // Podvucen tekst
```

```
paint::font f(L“Consolas”, 10, paint::font::style::bold);
```

paint::image једино можете да конструишете путем ка слици са диска. Након конструкције можете да добавите ширину и висину учитане слике (*getWidth(), getHeight()*).

paint::bitmap наслеђује од *image* класе. Исто је можете учитати са диска, или можете да креирате нову слику са одређеном ширином и висином.

```
// Ucitavanje slike sa diska
paint::image img(L”L"C:\\Users\\Admin\\Desktop\\img.png")”);
// Pravljenje bitmap-a sirokog 1024px, visokog 512px
paint::bitmap bmp(1024, 512);
// Ispisuje “1024X512”
std::cout << bmp.getWidth() << “X” << bmp.getHeight();
```

paint::brush је основна класа четкице. сама по себи не ради ништа, служи само да би од ње наследиле друге две врсте четкица.

paint::solidBrush је четкица која фарба једном бојом. Једноставно се и конструише - тако што јој се проследи *rgba8* објекат, који јој одређује боју. Након конструкције можемо и променити боју четкице помоћу *setColor(rgba8)* функције, и добити тренутну боју помоћу *getColor().*

```
// Cetkica ce biti plave boje
paint::solidBrush br(rgba8(0x0000FFFF));
// Menjanje boje u crvenu
br.setColor(0xFF0000FF);
// Dobavljanje trenutne boje(crvena)
auto col = br.getColor();
```

paint::textureBrush је четкица која користи слику као шаблон за цртање. За конструкцију узима референцу на слику (*paint::image*), и четвороугао који дефинише дестинациони правоугаоник. Након конструкције њена својства не могу бити читана или мењана.

```
	// Cetkica ce koristiti ranije konstruisanu paint::image
paint::textureBrush br(img);
```

paint::pen је оловка која користи четкицу за цртање линија. За конструкцију узима референцу на четкицу (*paint::brush*), и ширину линије(*float*). Након конструкције можете да читате и мењате ширину(getWidth(), setWidth(f32)), али четкицу само можете да мењате (*setBrush(const brush& brush)*).

paint се може инстанцирати помоћу контроле или *bitmap*-а по коме ће цртати. Његове функције цртања можемо поделити у групе: цртање текста, цртање ивица, и фарбање.

Све функције за цртање текста и фарбање узимају референце на *brush* објекте. Све функције цртање ивица узимају референце на *pen* објекте.

Од функција за рад са текстом имамо три - прва за мерење текста - враћа границе у којем ће текст бити написан:

```
measureWrite(
cwstr text,             // tekst koji merimo
i32 length,             // duzina teksta koji merimo
const bounds_t& bounds, // granice teksta
const font& font,       // font teksta
alignment horizontal,   // horizontalno pravnanje teksta
alignment vertical);    // vertikalno pravnanje teksta
```

Друга за превођење тачке у индекс карактера у коме је та тачка(коришћено за *tekstInput* контролу, за превођење клика корисника у индекс карактера при процесу селекције):

```
pointToCaretIndex(
	cwstr text,             // tekst koji merimo
	i32 length,             // duzina teksta koji merimo
	const bounds_t& bounds, // granice teksta
	const font& font,       // font teksta
	alignment horizontal,   // horizontalno pravnanje teksta
	alignment vertical,     // vertikalno pravnanje teksta
	const vector_t& point); // Tacka u kojoj trazimo indeks karaktera
```

Трећа за испис текстса:

```
write(
	cwstr text,             // tekst koji merimo
	i32 length,             // duzina teksta koji merimo
	const bounds_t& bounds, // granice teksta
	const font& font,       // font teksta
	const brush& brush,     // cetkica teksta
	alignment horizontal,   // horizontalno pravnanje teksta
	alignment vertical);    // vertikalno pravnanje teksta
```

Функције цртања ивица (префиксоване са *draw*) су:

```
drawRect(
      const bounds_t& bounds,       // granice pravougaonika za crtanje
      const pen& pen);              // olovka za crtanje

drawEllipse(
      const bounds_t& bounds,       // granice elipse za crtanje
      const pen& pen);              // olovka za crtanje

drawLine(
      const vector_t& from,         // pocetak linije
      const vector_t& to,           // kraj linije
      const pen& pen);              // olovka za crtanje
drawArc(
      const bounds_t& bounds,       // granice elipse za crtanje
      const vector_t& angleSweep,   // pocetni ugao i pomeraj ugla crtanja
      const pen& pen);              // olovka za crtanje

drawPoly(
      const vector_t* points,       // niz tacaka poligona
      u64 count,                    // broj tacaka u nizu
      const pen& pen);              // olovka za crtanje

drawCurve(
      const vector_t* points,       // niz tacaka za crtanje
      u64 count,                    // broj tacaka u nizu
      const pen& pen);              // olovka za crtanje

drawBezier(
      const vector_t& from,         // pocetak linije
      const vector_t& control1,     // prva kontrolna tacka
      const vector_t& control2,     // druga kontrolna tacka
      const vector_t& to,           // kraj linije
      const pen& pen);              // olovka za crtanje
```

Функције фарбање (префиксоване са *fill*) су:

```
clear(rgba8 color);                 // prekriva celu povrsinu datom bojom

      fillRect(
      const bounds_t& bounds,       // granice pravougaonika za farbanje
      const brush& pen);            // cetkica za farbanje
fillEllipse(
      const bounds_t& bounds,       // granice elipse za farbanje
      const brush& pen);            // cetkica za farbanje
fillArc(
      const bounds_t& bounds,       // granice elipse za farbanje
      const vector_t& angleSweep,   // pocetni ugao i pomeraj ugla farbanja
      const brush& brush);          // cetkica za farbanje
fillPoly(
      const vector_t* points,       // niz tacaka poligona za farbanje
      u64 count,                    // broj tacaka u nizu
      const pen& pen);              // cetkica za farbanje
```

Поред ових функција, постоје и оне које не цртају директно на контролу или слику, него утичу на даље цртање - за границе цртања:

```
      setClip(const bounds_t& bounds);    // Za postavljanja granica crtanja

      clearClip();                        // Za odstranjivanje granica
```

Функције за постављање трансформационе матрице:

```
void setTransform(
      f32 m11, f32 m12,
      f32 m21, f32 m22,
      f32 m31, f32 m32);
```

Функције за глатко цртање:

```
void setSmoothing(bool smoothing);
```

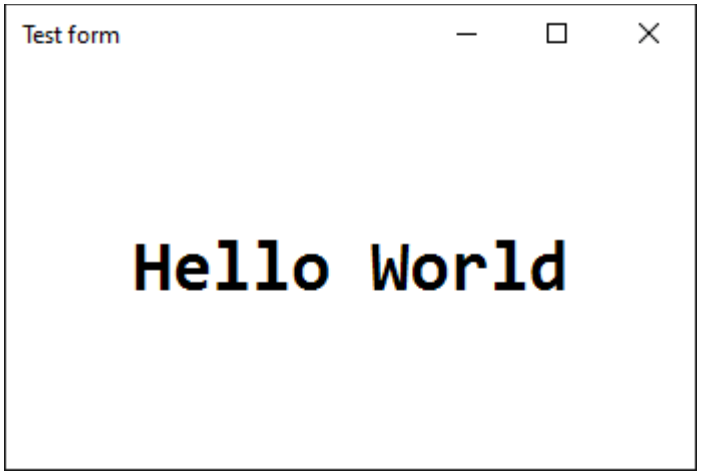
Цртање по контролама, односно по сликама би изгледало овако:

```
void onPaintHandler(control& sender, paintData& e) {
      paint& gfx = e.getPaint();
      //paint gfx(control) ako nije paint event

      gfx.clear(0x000000FF);
      paint::solidBrush br(0xFFFFFFFF);
      paint::font font(L"Consolas", 36.0f, );

      gfx.write(
            L"Hello World",
            wcslen(L"Hello World")
            { {}, sender.getClientSize() },
            font,
            br,
            alignment::center,
            alignment::center);
}
```

Резултат предходног кода види се испод на слици 3.14.



Слика 3.14 Изглед *прозора* елемента

4. ИМПЛЕМЕНТАЦИЈА ЕЛЕМЕНАТА БИБЛИОТЕКЕ

4.1. ОСНОВНА КОНТРОЛА

Имплементација ове класе се врши преко *pimpl* идиома – Ово се ради да би се сакрили имплементацијони детаљи класе, специфично да не морам да вршим инклузију *windows.h* фајла у *control.h* фајлу – јер бих са тим корисник ове библиотеке исто морао да инклудује тај фајл специфичан платформи на којој ради.

Код пимпл идиома класа у себи саржи **само** показивач на имплементацију. Ту ова класа одступа од овог идиома, јер у себи саржи далеко више од само тог показивача. То сам урадио да бих учинио екстензију ове класе лакшом.

Сам показивач на имплементацију је заправо показивач на системску имплементацију прозора – тип *HWND*. Шта требамо знати о овом типу је то да:

1. Пре инстанцирања архитипа системског прозора морамо му дати процедуру која ће обрађивати генерисане поруке (*window procedure*). Поруке попут промене величине прозора, притисак тастера итд. Та процедура је иста за све прозоре тадог архитипа.

2. При инстанцирању овог архитипа(отварања прозора), можемо му прикачити додатне податке - прослеђивањем екстра меморије која ће бити алоцирана по прозору.

Процедура неће бити видљива, односно замењива од стране корисника. Да би корисник мењао понашање контроле, користи раније поменуте догађаје. Те догађаје емитује контрола – односно процедура прозора. То ради превођењем системских порука у емитовање догађаја контроле. Ту наилазимо на један проблем – контрола поседује *HWND*, али он није „свестан“ контроле, и зато његова процедура не може емитовати њене догађаје. То решавамо помоћу тачке бр.2 – прикачивањем екстра меморије. Сваком системском прозору би прикачили показивач на контролу која га поседује – одатле би могао да анимира садржај и емитује догађаје дате контроле.

Приступ приватним члановима контроле од стране процедуре прозора постижемо тако што у имплементационом фајлу правимо нову класу, која ће садржати све потребне процедуре, и која ће бити пријатељ контроле. О тој класи треба да размишљамо као скривеним делом класе контроле – зато и има смисла да јој приступа приватним члановима.

Као што сам раније поменуо, контрола садржи много више од показивача на имплементацију. Ти додадни чланови углавном служе за кеширање података о прозору.

Поред пимпл идиома, могли смо користити и интерфејсе – основни интерфејс би имао виртуелне верзије свих функција које има контрола, и онда бисте имали посебну имплементацију тог интерфејса за сваку платформу. Али то би онда онемогућило наслеђивање од класе контроле. То је због тога што су имплементације датог интерфејса сакривене, па од њих се и не може наследити. Када би наша контрола наследили од основног интерфејса, онда не бисмо могли да користимо имплементацију специфичне платформе. Све то би могли решити помоћу другог интефејса, који бисмо прикачили контроле. Он би диктирао понашање дате контроле, и преко њега би корисник имплементирао врсте контроле(уместо *TextInput* контроле имали бисте инстанцу основне контроле, са прикаченим *TextInputBehaviour*).

4.2. УСИДРЕНА КОНТРОЛА

Усидрена контрола наслеђује од обичне контроле. Она служи да би аутоматски одређивала своје границе у односу на њеног родитеља(било која *group* контрола). Формуле за рачунање њених граница су испод:

```
bounds.from.x     = parent_size.x * anchor.from.x + offset.from.x;
bounds.from.y     = parent_size.y * anchor.from.y + offset.from.y;
bounds.to.x       = parent_size.x * anchor.to.x + offset.to.x;
bounds.to.y       = parent_size.y * anchor.to.y + offset.to.y;
```

Калкулација граница се врши у три случаја: При промени родитеља, при промени вечичине родитеља, и када се било која од релевантних вредности(*anchor,offset*) промени. Имплементира се преко догађаја родитеља – односно преко свог *onChangeParent* и родитељевог *onSize*. Пре рачунања своје величине када се догоди *onChangeParent*, она прво одстрани ослушкиваче *onSize* са прошлог родитеља, и закачи их за новог родитеља:

```
if (e.getOldParent() != nullptr) {
      e.getOldParent()->onSize().unsubscribe(
      callback<void, control&, sizeData&>::
      make<anchoredControl,
      &anchoredControl::recalcBoundsOnParentSize>(this));
}
if (e.getNewParent() != nullptr) {
      e.getNewParent()->onSize().subscribe(
      callback<void, control&, sizeData&>::
      make<anchoredControl,
      &anchoredControl::recalcBoundsOnParentSize>(this));
}
```

По терминацији ове контроле она одстрањује своје ослушкиваче са родитеља, како не би дошло до *изузетака*.

4.3. КЛАСА ЗА ЦРТАЊЕ

Слично попут класи контроле, ова класа је пример пимпл идиома, дакле садржи показивач на имплементацију, који је у овом случају инстанца системске класе *Gdiplus::Graphics*. Не садржи додатне чланове промнљиве поред тог показивача. Велика већина функција ове класе се у суштини преводе у позив функције системске класе – практично је само индирекција за системску класу, и ништа више.

Стварање ресурса(фонтови, четкице, сама класа за цртање итд.) имају предуслов да је *Gdiplus* иницијализован, према томе, њихово стварање се ослања на статичку функцију *initGdi()* – она је позвана сваки пут када покушамо да створимо било који ресурс*:*

```
void initGdi() {
      static ULONG_PTR ptr = 0;
      static Gdiplus::GdiplusStartupInput gdiplusStartupInput = nullptr;
      if (initialized) return;
      initialized = true;
      Gdiplus::GdiplusStartup(&ptr, &gdiplusStartupInput, nullptr);
}
```

5. ДЕМО АПЛИКАЦИЈА

Демо апликација ће бити задужена за поручивање пица, и све порџбине ће бити записане у једном фајлу на диску. Класа апликације ће изгледати овако:

```
class pizzaForm : public gui::window {
public:
	pizzaForm() = default;
	void start();
private:
	gui::textInput		_ti_name;
	gui::textInput		_ti_lastname;
	gui::anchoredGroup	_gr_type;
	gui::label		_lb_type;
	std::shared_ptr<std::vector<gui::checkInput*>> _group_type;
	gui::checkInput		_ci_type[4];
	gui::anchoredGroup	_gr_size;
	gui::label		_lb_size;
	std::shared_ptr<std::vector<gui::checkInput*>> _group_size;
	gui::checkInput		_ci_size[3];
	gui::anchoredGroup	_gr_options;
	gui::label		_lb_options;
	gui::checkInput		_ci_options[3];
	gui::textInput		_ti_address;
	gui::button		_btn_submit;

	void onPaintPizzaForm(control& sender, paintData& e);
	void onSizingPizzaForm(control& sender, sizingData& e);
	void onSubmit(control& sender, eventData& e);
	static void validateName(control&, gui::textInput::validationData&);
	static void validateLastName(control&,
		gui::textInput::validationData&);
	};
```

Одма са врха видимо да класа одма наслеђује од класе прозор, што значи да сама представља прозор у коме ће корисник уносити поруџбину. Празан конструктор постоји само да бисте могли да одложите иницијализацију класе, на позив чланске функције *start()*. Старт функција има пратећу имплементацију у којој иницијализује све под елементе:

```
void pizzaForm::start() {
	//
	// MAIN WINDOW SETUP
	gui::window::style window_style =
	gui::window::style::bordered |
	gui::window::style::captioned |
	gui::window::style::resizable |
	gui::window::style::minimize_button;

	bounds2<i32> window_bounds = { {200, 200}, {640, 360} };

	onClose().subscribe(gui::control::terminateOnClose);
	onSizing().subscribe(
		callback<void, control&, sizingData&>::
		make<pizzaForm, &pizzaForm::onSizingPizzaForm>(this));
	onPaint().subscribe(
		callback<void, control&, paintData&>::
		make<pizzaForm, &pizzaForm::onPaintPizzaForm>(this));
```

```
setStyle(window_style);
setBounds(window_bounds);
setCaption(L"Pizza online");
setFocusStyle(gui::control::focusStyle::unfocusable);
setVisible();

//
// TEXT INPUT NAME SETUP
adopt(_ti_name);
_ti_name.setFocusStyle(gui::control::focusStyle::focusable);
_ti_name.setAnchor({0.f, 0.f, 0.5f, 0.f});
_ti_name.setOffset({ { 10, 10 }, {-20, 30} });
_ti_name.setHint(L"Ime poručioca");
_ti_name.onValidate().subscribe(validateName);
_ti_name.setSelect({0, 0});
_ti_name.setVisible();
//
// TEXT INPUT LASTNAME SETUP
adopt(_ti_lastname);
_ti_lastname.setAnchor({ 0.5f, 0.f, 1.f, 0.f });
_ti_lastname.setOffset({ { 10, 10 }, {-20, 30} });
_ti_lastname.setHint(L"Prezime poručioca");
_ti_lastname.onValidate().subscribe(validateLastName);
_ti_lastname.setVisible();

//
// TYPE RADIO BUTTONS SETUP

      adopt(_gr_type) ;
      _gr_type.setAnchor({ 0, 0, 0.315f, 0 });
      _gr_type.setOffset({ {10, 50}, {-20, 150} });
      _gr_type.setVisible();
      _gr_type.adopt(_lb_type);
      _lb_type.setAnchor({ 0.f, 0.f, 1.f, 0.f });
      _lb_type.setOffset({ {1, 1}, {-2, 21} });
      _lb_type.setText(L"Izaberite tip pice");
      _lb_type.setVisible();
_group_type = std::make_shared<std::vector<gui::checkInput*>>();
for (auto& ci : _ci_type)
      _group_type->push_back(&ci);
for (auto& ci : _ci_type)
      ci.setRadioGroup(_group_type);
for (i32 i = 0; i < (sizeof(_ci_type) / sizeof(*_ci_type)); i++) {
      _gr_type.adopt(_ci_type[i]);
      _ci_type[i].setValue(i == 0);
      _ci_type[i].setAnchor({ 0, 0, 1, 0 });
      _ci_type[i].setOffset(
            { { 10.f, 30.f + i * 30},
            { -20.f, 18.f } });
      _ci_type[i].setText(
            i == 0 ? L"Vezuvio" :
            i == 1 ? L"Margarita" :
            i == 2 ? L"Čikago pica" :
            i == 3 ? L"New York pica" : L"N/A");
      _ci_type[i].setVisible();
}

//
// SIZE RADIO BUTTONS SETUP
adopt(_gr_size);
```

```
_gr_size.setAnchor({ 0.33f, 0.0f, 0.655f, 0 });
_gr_size.setOffset({ {10, 50}, {-20, 150} });
_gr_size.setVisible();
_gr_size.adopt(_lb_size);
_lb_size.setAnchor({ 0.f, 0.f, 1.f, 0.f });
_lb_size.setOffset({ {1, 1}, {-2, 21} });
_lb_size.setText(L"Izaberite veličinu pice");
_lb_size.setVisible();
_group_size = std::make_shared<std::vector<gui::checkInput*>>();
for (auto& ci : _ci_size)
      _group_size->push_back(&ci);
for (auto& ci : _ci_size)
      ci.setRadioGroup(_group_size);
for (i32 i = 0; i < (sizeof(_ci_size) / sizeof(*_ci_size)); i++) {
      _gr_size.adopt(_ci_size[i]);
      _ci_size[i].setValue(i == 0);
      _ci_size[i].setAnchor({ 0, 0, 1, 0 });
      _ci_size[i].setOffset(
            { { 10.f, 30.f + i * 30},
            { -20.f, 18.f } });

      _ci_size[i].setText(
            i == 0 ? L"24cm - mala" :
            i == 1 ? L"36cm - srednja" :
            i == 2 ? L"48cm - velika" : L"N/A");
            _ci_size[i].setVisible();
}

//
// OPTIONS CHECK INPUT SETUP
adopt(_gr_options);
_gr_options.setAnchor({ 0.67f, 0.0f, 1.0f, 0 });
_gr_options.setOffset({ {10, 50}, {-20, 150} });
_gr_options.setVisible();
_gr_options.adopt(_lb_options);
_lb_options.setAnchor({ 0.f, 0.f, 1.f, 0.f });
_lb_options.setOffset({ {1, 1}, {-2, 21} });
_lb_options.setText(L"Izaberite dodatne opcije");
_lb_options.setVisible();
_group_size = std::make_shared<std::vector<gui::checkInput*>>();
for (auto& ci : _ci_size)
      _group_size->push_back(&ci);
for (auto& ci : _ci_size)
      ci.setRadioGroup(_group_size);
for (i32 i = 0; i < (sizeof(_ci_options) / sizeof(*_ci_options));
i++){
      _gr_options.adopt(_ci_options[i]);
      _ci_options[i].setValue(i == 0);
      _ci_options[i].setAnchor({ 0, 0, 1, 0 });
      _ci_options[i].setOffset({
            { 10.f, 30.f + i * 30},
            { -20.f, 18.f } });
      _ci_options[i].setText(
            i == 0 ? L"dodatne masline" :
            i == 1 ? L"dadatan sir" :
            i == 2 ? L"punjena kora" : L"N/A");
      _ci_options[i].setVisible();
}

//
```

```
	// TEXT INPUT ADDRESS SETUP
	adopt(_ti_address);
	_ti_address.setAnchor({ 0.0f, 0.f, 1.f, 0.f });
	_ti_address.setOffset({ { 10, 210 }, {-20, 30} });
	_ti_address.setHint(L"Adresa poručioca");
	_ti_address.setVisible();

	//
	// SUBMIT BUTTON SETUP
	adopt(_btn_submit);
	_btn_submit.setAnchor({ 0, 1, 1, 1 });
	_btn_submit.setOffset({ {10, -50}, { -20, 40} });
	_btn_submit.setText(L"Naruči");
	_btn_submit.onClick().subscribe(
		callback<void, control&, eventData&>::
		make<pizzaForm, &pizzaForm::onSubmit>(this));
	_btn_submit.setVisible();

	//
	// Pump
	plt::message msg;
	while (isValid()) {
		msg.awaitPull();
		msg.dispatch();
	}

}
```

Видимо да прво подешавамо параметре прозора – ослушкиваче његових догађаја, његов стил, границе и наслов. Након што смо све подесили позивамо *setVisible()* да бисмо га учинили видљивим кориснику.

Након тога иницијализујемо све елементе корисничког интерфејса: прво их присвајамо прозору (*adopt(*)*), онда им подешавамо величину(усидрење и офсет), изглед, текст итд. И на крају их приказујемо(исто функцијом **.setVisible()*).

На крају предајемо контролу програма пумпи порука. Ово у суштини морамо да урадимо, да би апликација коректно обрађивала поруке графичких елемената – слично функцији *mainLoop()* у *Tkinter* библиотеци.

За цртање форме користимо чланску функцију *onPaintPizzaForm*, она је овако дефинисана:

```
	void pizzaForm::onPaintPizzaForm(control& sender, paintData& e) {
	auto br =
gui::defaults::getTheme()>getBackground(gui::theme::select::normal);
	auto sz = sender.getClientSize();
	if (br != nullptr)
		e.getPaint().fillRect({ {}, sz }, *br);
}
```

У првој линији добављамо референцу на нормалну позадинску четкицу аутоматске теме. У наредној добављамо величину контроле коју фарбамо(у овом случају пица форма). На крају фарбамо површину форме добављеном четкицом, ако није нул.

```
	void pizzaForm::onSizingPizzaForm(control& sender, sizingData& e) {
	gui::control::limitSizingWidth(e, { 640, 1280 });
	gui::control::limitSizingHeight(e, { 360, 720 });}
```

При промени величине форме, примењујемо помоћне функције за ограничавање величине прозора (*limitSizingWidth* и *limitSizingHeight*), тако да величина прозора не одступа превише од предвиђене величине. Те помоћне функције, узимају дводимензионални вектор, чије *x* и *y* вредности означавају минимум и максимум ширине, односно висине контроле. Они узимају у обзир која ивица контроле се користи за ширење односно скупљање прозора.

Валидације имена(исто и за презиме) је имплементирана на пратећи начин:

```
	void pizzaForm::validateName(
	control& sender,
	gui::textInput::validationData&) {

	gui::textInput::validateNotDigit(sender, e);
	gui::textInput::validateNotWhitespace(sender, e);
}
```

Валидацију у овом случају вршимо помоћним функцијама *textInput* контроле *validateNotDigit* и *validateNotWhitespace*, које ће отказати упис текста у поље, ако садржи цифру, односно било коју врсту размака.

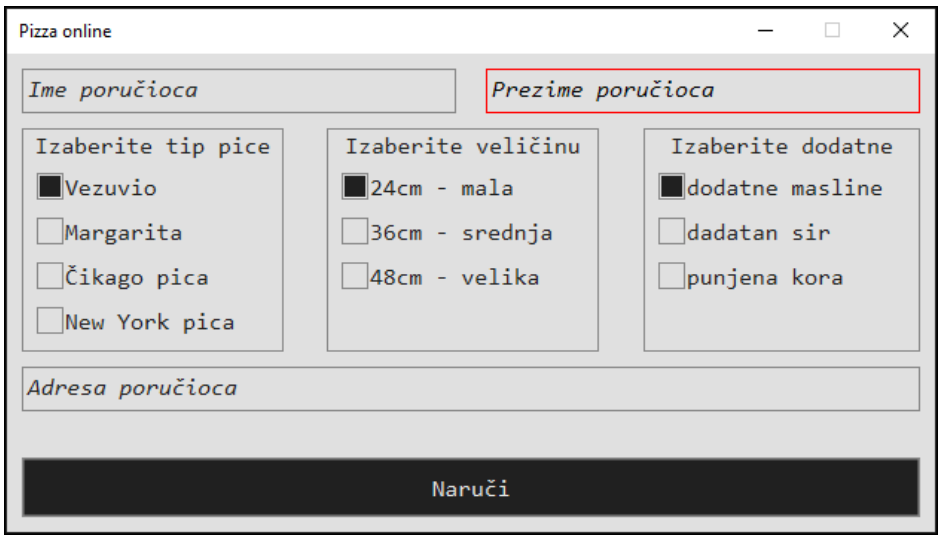
При притиску на дугме програм записује у фајл информације о поруџбини. Имплементација те функције:

```
	void pizzaForm::onSubmit(control& sender, eventData& e) {
	FILE* out = nullptr;
	_wfopen_s(&out, L"zapisnik.txt", L"a");
	if (out == nullptr)
		return;
	auto ret = _setmode(_fileno(out), _O_WTEXT);
	fwprintf(
		out,
		L"Narudzbina:\n\tIME: %ls\n\tPREZIME: %ls\n\tADRESA: %ls\n\t",
		_ti_name.getText().c_str(),
		_ti_lastname.getText().c_str(),
		_ti_address.getText().c_str());
	gui::checkInput* pizza_type = nullptr;
	for (auto& p : _ci_type) {
		if (p.getValue()) {
			pizza_type = &p;
			break;
		}
	}
	if (pizza_type != nullptr)
	fwprintf(out, L"TYPE: %ls\n\t", pizza_type->getText().c_str());
	gui::checkInput* pizza_size = nullptr;
	for (auto& p : _ci_size) {
		if (p.getValue()) {
			pizza_size = &p;
			break;
		}
	}
	if (pizza_size != nullptr)
		fwprintf(out, L"SIZE: %ls\n\t", pizza_size->getText().c_str());
	if (pizza_size != nullptr)
		fwprintf(out, L"OPT: ");
	for (auto& p : _ci_options) {
		if (p.getValue())
```

```
                fwprintf(out, L"%ls, ", p.getText().c_str());
        }
        fwprintf(out, L"\n");
        fclose(out);
}
```

Функција прво отвара фајл и припрема га за уписивање текста. Овде користим Ц верзију фајлова, уместо фајл стримова у Ц++, јер знам како да их учитим да користе широке карактере. Након тога, ако је фајл успешно отворен, редом уписујемо унесене информације са форме у тај фајл, и на крају га затварамо.

Као резултат овог кода, након позива старт функције отвара се прозор који изгледа као на наредној слици (слика 5.1). Ако промените тему(у овом случају је коришћена светла тема) можете добити другачији изглед(на пример, исто уграђена тамна тема).



Слика 5.1 Изглед демо апликације.

6. ЗАКЉУЧАК

У реализацији овог пројекта, коришћене су две технологије (поред стандардне библиотеке Ц++-а): *windows* библиотека, и *Gdiplus* библиотека. Радно окружење које сам користио за развој овог пројекта је *Visual studio 2022.*

Windows библиотека је коришћена за отварање, затварање и манипулисање прозорима; као и за увођење свих класа које припадају *cauldron-platform* пројекту, док је *Gdiplus* библиотека коришћена за цртање по прозорима.

У наставку овог пројекта бих заменио *Gdiplus* библиотеку са модернијом графичком библиотеком, попут *Direct3D* библиотеком, и знатно проширио број стандардних контрола присутних у библиотеци(нпр. Скролабилну контролу, *listBox, dropDown*). Поред тога, додао бих више функционалности порукама. Тренутно само можете да их добавите, и да их извршите. Волео бих да им додам начин да некако можете да прочитате њихов садржај и да их одбаците, по потреби.

7. ИНДЕКС ПОЈМОВА

8. ЛИТЕРАТУРА

[1] Слободанка Ђенић, Јелена Митић и Светлана Штрбац, *Програмирање на језику "C" и основи програмирања на језику "C++"*, VIŠER, 2019.

[2] Слободанка Ђенић, *Програмски језици C и C++*, VIŠER, 2020.

[3] Др Перица Штрбац, *Објектно програмирање 1*, VIŠER, 2019.

[4] Боје, трансформација боја 2022. Септембар
https://www.rapidtables.com/convert/color

[5] Матрице, рад са матрицама 2022. Септембар
https://en.wikipedia.org/wiki/Transformation_matrix

[6] Прозори на *windows*-у 2022. Септембар
https://learn.microsoft.com/en-us/windows/win32/api/winuser

[7] Синхронизација на *windows*-у 2022. Септембар
https://learn.microsoft.com/en-us/windows/win32/api/synchapi

[8] Тредовање на *windows*-у 2022. Септембар
https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi

[9] ГДИ плус 2022. Септембар
https://learn.microsoft.com/en-us/windows/win32/gdiplus

9. ПРИЛОЗИ

[1] **Kod biblioteke** 2022. Септембар
https://github.com/djordjermus/cauldron

[2] **Слика коришћена у слици 3.12** 2022. Септембар
https://cwexplorationphotography.com

10. ИЗЈАВА О АКАДЕМСКОЈ ЧЕСТИТОСТИ

ИЗЈАВА О АКАДЕМСКОЈ ЧЕСТИТОСТИ

Студент (име, име једног родитеља и презиме): **Ђорђе, Милан Рмуш**

Број индекса: **РТ-14/19**

Под пуном моралном, материјалном, дисциплинском и кривичном одговорношћу изјављујем да је завршни рад, под насловом:

Имплементација Ц++ библиотеке за графички интерфејс

1. резултат сопственог истраживачког рада;
2. да овај рад, ни у целини, нити у деловима, нисам пријављиво/ла на другим високошколским установама;
3. да нисам повредио/ла ауторска права, нити злоупотребио/ла интелектуалну својину других лица;
4. да сам рад и мишљења других аутора које сам користио/ла у овом раду назначио/ла или цитирао/ла у складу са Упутством;
5. да су сви радови и мишљења других аутора наведени у списку литературе/референци који је саставни део овог рада, пописани у складу са Упутством;
6. да сам свестан/свесна да је плагијат коришћење туђих радова у било ком облику (као цитата, прафраза, слика, табела, дијаграма, дизајна, планова, фотографија, филма, музике, формула, вебсајтова, компјутерских програма и сл.) без навођења аутора или представљање туђих ауторских дела као мојих, кажњиво по закону (Закон о ауторском и сродним правима), као и других закона и одговарајућих аката Високе школе електротехнике и рачунарства струковних студија у Београду;
7. да је електронска верзија овог рада идентична штампаном примерку овог рада и да пристајем на његово објављивање под условима прописаним актима Високе школе електротехнике и рачунарства струковних студија у Београду;
8. да сам свестан/свесна последица уколико се докаже да је овај рад плагијат.

У Београду, __. __. 202_. године

Својеручни потпис студента

_____