Dokumentacja projektu - Programowanie w C 2

Politechnika Świętokrzyska w Kielcach

Grzegorz Bujak Jędrzej Cieślikiewicz Paweł Cieszkowski

Spis treści

Opis projektu

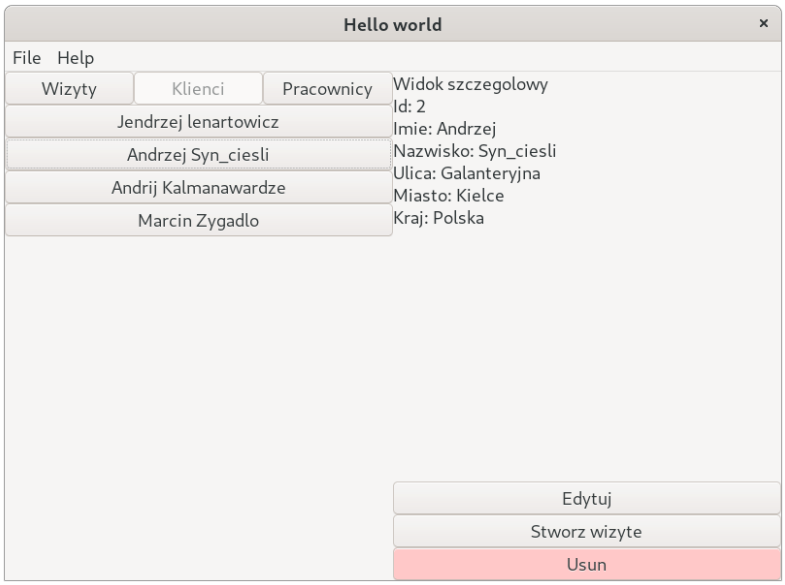
Przygotowaliśmy projekt o temacie “e-warsztat”. Przygotowany program ma ułatwiać organizację pracy w serwisie samochodowym. Wykonaliśmy to zadanie przygotowując program graficzny spełniający następujące funkcje:

- Baza klientów (lista klientów i danych o klientach)
- Baza pracowników
- Zapisywanie klientów na wizytę u konkretnego pracownika

Program został przygotowany przy użyciu biblioteki wxWidgets. Ta biblioteka umożliwia przygotowanie aplikacji graficznych działających na systemach operacyjnych Windows, MacOS oraz GNU/Linux.

Dokumentacja nie skupia się na poszczególnych klasach, lecz funkcjonalnościach programu - kilka razy wraca do opisywania tej samej klasy. Uważamy, że w ten sposób łatwiej zrozumieć implementację konkretnej funkcjonalności, która może przydać do innego projektu, bez zapoznawania się z całą strukturą aplikacji.

Interfejs graficzny



Rysunek 1: interfejs graficzny

Interfejs graficzny aplikacji przedstawiony na zdjęciu nr. 1 jest podzielony na dwie połowy. Lewa połowa przedstawia graficzną reprezentację bazy danych aplikacji.

Na szczycie lewego panelu znajdują się zakładki dla każdego typu danych przechowywanych w aplikacji. Poniżej paska z zakładkami dla każdej jednostki wybranego typu danych renderowany jest przycisk, którego naciśnięcie sprawia wyświetlenie szczegółowych danych o tej jednostce w prawym panelu.

Prawy panel zawiera duże pole tekstowe, w którym wyświetlane są szczegółowe informacje o przechowywanych danych. Ponadto, na spodzie prawego panelu znajdują się trzy przyciski służące do:

- edycji danych, których szczegóły wyświetlane są powyżej
- umówienie klienta na wizytę (przycisk aktywny tylko, gdy wybrane dane są typu klient)
- usunięcia wybranych danych z aplikacji

Implementacja

Klasa aplikacji (wxApp)

Protopyp klasy aplikacji (plik include/app.hpp):

```
#pragma once

#include <wx/wx.h>
#include "frame.hpp"

class MyApp : public wxApp {
    public:
    MyFrame* main_frame;
    virtual bool OnInit();
};
```

Implementacja (plik src/app.cpp):

```
#include <wx/wx.h>
#include "../include/app.hpp"
#include "../include/frame.hpp"

wxIMPLEMENT_APP(MyApp);

bool MyApp::OnInit() {
    main_frame = new MyFrame(
        "Hello world", {50, 50}, {800, 600});
    main_frame->Show(true);
    return true;
}
```

Klasa aplikacji jest typowa dla aplikacji korzystających z biblioteki wxWidgets. Jest to klasa (nazwana MyApp) dziedzicząca publicznie z klasy wxApp. Implementuje metodę OnInit, w której tworzy instancję klasy MyFrame i wywołuje jej metodę "Show".

Klasa MyFrame

```
#pragma once

#include <wx/wx.h>
#include "display_list.hpp"
#include "globals.hpp"

class MyFrame : public wxFrame {
    public:
    MyFrame(const wxString& title, const wxPoint& pos,
            const wxSize& size);
    private:
    void OnHello (wxCommandEvent& event);
    void OnExit  (wxCommandEvent& event);
    void OnAbout (wxCommandEvent& event);
    void OnSave  (wxCommandEvent& event);
    void OnOpen  (wxCommandEvent& event);
    void OnNew   (wxCommandEvent& event);

    wxDECLARE_EVENT_TABLE();
};

enum {
    ID_Hello = 1
};
```

Klasa MyFrame dziedziczy publicznie po klasie wxFrame. Metody, jakie zaimplementowaliśmy w tej klasie to konstruktor i kilka metod obsługujących wydarzenia.

- OnExit - funkcja obsługuje wybranie obcji exit w menubarze
- OnHello, OnAbout - funkcje obsługujące nazwane po sobie opcje w menubarze Wyświetlają informacje o programie
- OnSave, OnOpen - obsługują zapisywanie i wczytywanie danych do/z plików
- OnNew - funkcja obsługuje opcję stworzenia nowych danych

Implementacja zapisywania i wczytywania danych

Interfejs (klasa MyFrame)

```
void MyFrame::OnOpen(wxCommandEvent& event) {
    wxFileDialog openFileDialog(this, _("Open XYZ file"), "", "",
                                "XYZ files (*.xyz)|*.xyz", wxFD_OPEN|wxFD_FILE_MUST_EXIST);

    if (openFileDialog.ShowModal() == wxID_CANCEL)
        return;

    std::ifstream stream;
    stream.open(openFileDialog.GetPath());
    if (stream.fail()) {
        wxLogError("Cannot open file " + openFileDialog.GetPath());
        return;
    }

    g_datastore.load(stream);
    g_display_list->display(0);
}

void MyFrame::OnSave(wxCommandEvent& event) {
    wxFileDialog
        saveFileDialog(this, _("Save XYZ file"), "", "",
                       "XYZ files (*.xyz)|*.xyz", wxFD_SAVE|wxFD_OVERWRITE_PROMPT);
    if (saveFileDialog.ShowModal() == wxID_CANCEL)
        return;

    std::ofstream stream;
    stream.open(saveFileDialog.GetPath());
    if (stream.fail()) {
        wxLogError("Cannot open file " + saveFileDialog.GetPath());
        return;
    }

    g_datastore.save(stream);
}
```

W tym miejscu zaimplementowany jest tylko interfejs graficzny zapisywania i wczytywania danych. Prawdziwa implementacja znajduje się w klasie `datastore_t`.

Jest to typowa implementacja interfejsu wyboru pliku w bibliotece wxWidgets. Niewiele różni się od przykładowej implementacji znajdującej się na wikipedii biblioteki. Interfejs wyboru pliku nie jest przygotowany przez nas i różni się prawie całkowicie na różnych systemach operacyjnych.

"Prawdziwa" implementacja (klasa datastore_t)

Wszystkie informacje przechowywane w pamięci naszego programu znajdują się w klasie `datastore_t`. Jedna instancja tej klasy używana jest w całym programie, jako zmienna globalna `g_datastore`.

Klasa ta posiada `std::vector` dla każdego typu przechowywanych danych. Jej prototyp wygląda tak:

```
#pragma once

#ifndef DATA_STORE_HPP
```

```
#define DATA_STORE_HPP

#include <vector>
#include <iostream>
#include <wx/wx.h>

#include "data_classes.hpp"

class datastore_t {
    std::vector<customer_t> customers;
    std::vector<employee_t> employees;
    std::vector<appointment_t> appointments;

    int get_max_customer_id();
    int get_max_employee_id();
    int get_max_appointment_id();

    public:
    customer_t* get_customer(int);
    employee_t* get_employee(int);
    appointment_t* get_appointment(int);

    void add(employee_t);
    void add(customer_t);
    void add(appointment_t);

    void save(std::ostream& output);
    void load(std::istream& input);

    void delete_data(data_interface*);

    std::vector<customer_t>& get_customers();
    std::vector<employee_t>& get_employees();
    std::vector<appointment_t>& get_appointments();
};

#endif
```

`customer_t`, `employee_t` i `appointment_t` to struktury (klasy bez pól prywatnych). Ich implementacja nie jest w tym miejscu ważna. Ważne jest to, że dla każdej struktury przeciążono `operator<<` oraz `operator>>` dla `std::ostream` i `std::istream`, co jest wykorzystywane w metodach save i load klasy `datastore_t`.

Każda ze struktur dziedziczy tez po klasie `data_interface` co umożliwia pisanie bardziej ogólnych klas przedstawiających informacje o danych w interfejsie graficznym.

```
struct data_interface {
    virtual std::string to_str() = 0;
    virtual std::string display_name() = 0;
};
```

Metoda `to_str` zwraca `std::string` zawierający wszystkie informacje zawarte w strukturze. Ten ciąg znaków jest wyświetlany w interfejsie graficznym jako `wxStaticText`.

Metoda `display_name` zwraca krótki ciąg znaków służący do umieszczenia w `wxButton` dla każdego elementu danych przechowywanych w aplikacji.

Implementacja metod `save` i `load`

```
void datastore_t::save(std::ostream& output) {
    for (const customer_t& customer : customers) {
        output << "BEGIN CUSTOMER\n";
        output << customer << '\n';
    }
    for (const employee_t& employee : employees) {
        output << "BEGIN EMPLOYEE\n";
        output << employee << '\n';
    }
    for (const appointment_t& app : appointments) {
        output << "BEGIN APPOINTMENT\n";
        output << app << '\n';
    }
}

void datastore_t::load(std::istream& input) {
    customers.clear();
    employees.clear();
    appointments.clear(); // kasowanie wszystkich danych

    std::string current_line;
    while (input) {
        std::getline(input, current_line);
        if (current_line == "BEGIN CUSTOMER") {
            customer_t new_customer;
            input >> new_customer;
            customers.push_back(new_customer);
        } else if (current_line == "BEGIN EMPLOYEE") {
            employee_t new_employee;
            input >> new_employee;
            employees.push_back(new_employee);
        } else if (current_line == "BEGIN APPOINTMENT") {
            appointment_t app;
            input >> app;
            appointments.push_back(app);
        }
    }
}
```

Implementacja tych metod jest dość prosta. Plik, który powstaje w skutek zapisu danych wygląda tak:

```
BEGIN CUSTOMER
2 Andrzej Syn_ciesli Galanteryjna Kielce Polska
BEGIN EMPLOYEE
4 Maciej Buraczany Rewolucji_pazdziernikowej Minsk Bialorus 2600
BEGIN APPOINTMENT
0 0 4 14-Maj-2013 Wymiana_samochodu
```

Przy zapisie `customer_t` oraz `employee_t` pierwszym słowem jest id tego klienta lub pracownika. Wizyta, oprócz własnego, posiada też id klienta i pracownika, do którego klient jest umówiony. Zapisywanie id jako liczb całkowitych zamiast np. wskaźników na obiekty ułatwia usuwanie danych. Gdy pracownik o zapisanym id został usunięty, w interfejsie zostanie wyświetlone “nie można znaleźć osoby”.

Implementacja tworzenia nowych danych ("wyskakujące okienko")

Tworzenie nowych danych odbywa się po wciśnięciu opcji file > new w menubarze. Wydarzenie to obsługuje funkcja OnNew w klasie MyFrame.

```
void MyFrame::OnNew(wxCommandEvent& event) {
    new_data_window_t new_data_window {this};
    new_data_window.ShowModal();
}
```

`new_data_window` to wyskakujące okienko (klasa dziedziczy po `wxDialog`). Metoda `ShowModal` powoduje wyświetlenie okienka i zablokowanie możliwości interakcji z oknem głównym do czasu zamknięcia okna. W tym przypadku możliwe jest zainicjalizowanie zmiennej na stosie (bez słowa `new`), bo okno i tak zostanie zniszczone przed wyjściem z funkcji `OnNew`.

```
#define CHOICE_NUMBER 2

class new_data_window_t : public wxDialog {

    std::unordered_map<std::string, wxTextCtrl*> input_fields;
    event_functor_t on_type_selected;
    event_functor_t on_commit;

    const wxString choices[CHOICE_NUMBER] = {"Customer", "Employee"};
    customer_t customer;
    employee_t employee;

    data_collector_t collector; // wytłumaczone później

    wxBoxSizer* main_sizer;
    wxChoice* choice_widget;
    public:
    new_data_window_t(wxWindow*);
};
```

Okno to pozwana na wybór pomiędzy tworzeniem nowego pracownika a tworzeniem nowego klienta. Następnie w oknie wyświetlają się pola tekstowe dla każdego pola w strukturach `customer_t` lub `employee_t`. Implementacja wygląda tak:

```
new_data_window_t::new_data_window_t(wxWindow* parent) {
    this->Create(parent, wxNewId(), "Add new data");

    on_type_selected = [this] (wxEvent& event) {
        int selection = choice_widget->GetSelection();
        choice_widget->Disable();

        switch (selection) {
            case 0:
                add(collector, customer);
                break;
            case 1:
                add(collector, employee);
                break;
        }

        auto commit_button = new wxButton {this, wxNewId(), "commit"};
        main_sizer->Add(commit_button, wxSizerFlags{1}.Expand());

        SetSizerAndFit(main_sizer);
```

```
        this->Bind(wxEVT_BUTTON, this->on_commit);
    };

    on_commit = [this] (wxEvent& event) {
        if (!collector.collect()) return;
        int selection = choice_widget->GetSelection();
        switch (selection) {
            case 0:
                g_datastore.add(customer);
                break;
            case 1:
                g_datastore.add(employee);
                break;
        }
        this->Close(true);
    };

    main_sizer = new wxBoxSizer {wxVERTICAL};
    collector.set_sizer(main_sizer);
    choice_widget = new wxChoice {
                    this, wxNewId(),
                    wxDefaultPosition, wxDefaultSize,
                    CHOICE_NUMBER, this->choices };
    main_sizer->Add(choice_widget, wxSizerFlags{1});
    this->Bind(wxEVT_CHOICE, on_type_selected);

    SetSizerAndFit(main_sizer);
}
```

Klasa `data_collector_t`

Wyświetlanie pól tekstowych do wstawiania danych i zapis tych danych do struktur jest zrealizowany za pomocą klasy `data_collector_t`.

Ta klasa dostaje wskaźnik na `wxBoxSizer`, do którego będzie dodawała pola tekstowe. Następnie można do niej dodawać wskaźnik na dane do zebrania oraz opis pola tekstowego, który zostanie wyświetlony nad nim (`const char*`). Następnie, przy wywołaniu metody `collect`, sczytuje dane z pól tekstowych i zapisuje je w pamięci, na którą dostała wskaźnik. `collect` zwraca `false`, gdy nie udało się zinterpretować danych z pola tekstowego jako `int`.

```
class data_collector_t {
    wxBoxSizer* sizer = nullptr;
    std::vector<
        std::pair<wxTextCtrl*, std::string*>
    > str_vec;
    std::vector<
        std::pair<wxTextCtrl*, int*>
    > int_vec;
    wxTextCtrl* new_text_ctrl(const char*);
    public:
    void set_sizer(wxBoxSizer*);
    void add(std::string*, const char*);
    void add(int*, const char*);
    bool collect();
};
```

```
void add(data_collector_t&, customer_t&);
void add(data_collector_t&, employee_t&);
void add(data_collector_t&, appointment_t&);
```

funkcje `add` powyżej dodają każde pole struktur do kolektora za pomocą metody `add`.

```
void
data_collector_t::set_sizer(wxBoxSizer* sizer) {
    this->sizer = sizer;
}

void
data_collector_t::add(int* data, const char* label) {
    assert(this->sizer != nullptr);
    auto ctrl = new_text_ctrl(label);
    ctrl->SetValue(std::to_string(*data)); // kolektor ustawia
    // wartośc w polu tekstowym na taką, jaka znajduje się aktualnie w pamięci,
    // przez co można go używać również do edycji już istniejących danych
    int_vec.push_back(
        std::pair<wxTextCtrl*, int*>{ctrl, data}
    );
}

void
data_collector_t::add(std::string* data, const char* label) {
    assert(this->sizer != nullptr);
    auto ctrl = new_text_ctrl(label);
    ctrl->SetValue(*data); // to samo, co komentarz wyżej
    str_vec.push_back(
        std::pair<wxTextCtrl*, std::string*>{ctrl, data}
    );
}

wxTextCtrl*
data_collector_t::new_text_ctrl(const char* label) {
    auto ctrl = new wxTextCtrl {
        sizer->GetContainingWindow(),
        wxNewId()
    };
    sizer->Add(
        new wxStaticText{sizer->GetContainingWindow(), wxNewId(), label}
    );
    sizer->Add(ctrl, wxSizerFlags{1}.Expand());
    return ctrl;
}

bool
data_collector_t::collect() {
    assert(this->sizer != nullptr);
    bool ret = true;
    for (auto& i : str_vec) {
        *(i.second) = i.first->GetValue().ToStdString();
    }
    for (auto& i : int_vec) {
        int* dest = i.second;
        long tmp;
        if (!i.first->GetValue().ToLong(&tmp))
```

```
            ret = false;
        *dest = (int) tmp;
    }
    return ret;
}
```