Politechnika Śląska w Gliwicach
Wydział Automatyki, Elektroniki i Informatyki

Podstawy Programowania Komputerów

Zadanie nr 7

Program do zarządzania bankiem

autor	Łukasz Ważny
prowadzący	dr inż. Jacek Widuch
rok akademicki	2018/2019
kierunek	informatyka
rodzaj studiów	SSI
semestr	1
termin laboratorium / ćwiczeń	wtorek, 13:45 – 15:15
grupa	1
sekcja	3
termin oddania sprawozdania	2018-01-25
data oddania sprawozdania	2018-01-25

1 Treść zadania

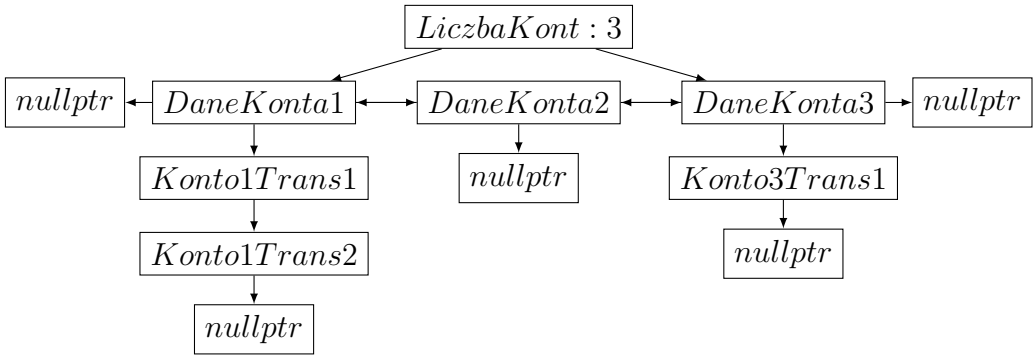
Napisać program do zarządzania bankiem. Program powinien umożliwić użytkownikowi założenie konta (przy zakładaniu konta należy podać dane osobowe, numer konta generowany jest automatycznie) oraz wykonywanie operacji na koncie: wpłatę dowolnej kwoty, wypłatę z konta (w programie ustalić limit debetu), uzyskanie wyciągu z konta za określony okres, likwidację konta (możliwa tylko wtedy jeżeli nie ma debetu na koncie). Użytkownik wykonuje operacje na swoim koncie po podaniu loginu i hasła. Dodatkowo umożliwić dowolnemu użytkownikowi (także temu, który nie posiada konta w banku) wpłatę dowolnej kwoty na konto innego użytkownika banku. Ponadto umożliwić wyświetlenie danych o wszystkich kontach (wykonuje pracownik banknu), wyświetlenie wykonanych operacji w danym okresie, wyświetlenie użytkowników, którzy mają debet na koncie.

2 Analiza zadania

Zagadnienie przedstawia problem wczytania danych z pliku do struktury dynamicznej, modyfikacji tych danych przez użytkownika (włączając w to dodanie nowych danych), wyświetlania aktualnych danych przez użytkownika oraz zapisania zmodyfikowanych danych do pliku (jeżeli jakakolwiek modyfikacja wystąpiła).

2.1 Struktury danych

W programie wykorzystano samoorganizującą się dwukierunkową listę jednokierunkowych list. Każdy element głównej listy przechowuje dane jednego konta bankowego, wskaźniki do poprzedniego i następnego elementu oraz wskaźnik do pierwszego elementu listy, której każdy element przechowuje dane pojedynczej transakcji wykonanej przez użytkownika konta oraz wskaźnik do następnego elementu listy. Każdy element głównej listy przechowuje również liczbę transakcji. Ponadto, za każdym razem, gdy następuje odwołanie do danego elementu głównej listy, element ten zamienia się miejscem z elementem znajdującym się o jedno miejsce bliżej początku listy. Wykorzystano również strukturę służącą do opisu listy, która zawiera wskaźnik do pierwszego elementu głównej listy, do ostatniego oraz liczbę elementów. Rysunek 1 przedstawia przykład wyżej opisanej struktury. Zdecydowałem się na użycie samoorganizującej się listy, ze względu na to, że użytkownik częściej wykonujący operacje, szybciej może się dostać do swojego konta. Jest ona dwukierunkowa, ponieważ w przypadku takiej listy łatwiej jest za-



Rysunek 1: Przykład dwukierunkowej listy kont bankowych jednokierunkowych list transakcji. Lista opisuje trzy konta bankowe, z czego użytkownik konta pierwszego wykonał od czasu założenia konta 3 transakcje; użytkownik drugiego - nie wykonał żadnych; a użytkownik trzeciego konta - jedną transakcję. Na samej górze znajduje się pole zawierające wskaźnik do pierwszego i ostatniego elementu oraz informacje o liczbie elementów.

mienić element miejscami z poprzednim. Do każdego elementu głównej listy „dołączyłem" listę jednokierunkową, ponieważ taka struktura w zupełności wystarcza do wykonywania operacji wyświetlenia wszystkich elementów w kolejności do pierwszego do ostatniego oraz do operacji dodawania elementu na koniec.

2.2 Algorytmy

Główne algorytmy programu są odpowiedzialne za umieszczanie nowych elementów w liście dwukierunkowej, jak i jednokierunkowej, samoorganizowanie się listy dwukierunkowej oraz wyszukiwanie odpowiedniego elementu listy za pomocą danego kryterium. Dokładny przebieg wymienionych wyżej algorytmów można wywnioskować z kodu funkcji przeprowadzających te operacje. Te funkcje to odpowiednio utworz_konto str.12, wplata_dowolnej_kwoty str.8, odwolanie str.9, istnieje_konto_login str.7.

3 Specyfikacja zewnętrzna

Program jest uruchamiany poprzez dwukrotne kliknięcie w plik wykonywalny `.exe`. Jeżeli tam, gdzie znajduje się program, nie ma pliku z danymi odnośnie kont i transakcji, to na wstępie wyświetla się nam powiadomienie o tym fakcie. Należy wtedy wcisnąć dowolny przycisk. Następnie wyświetla się menu. W przypadku, gdy plik z danymi występował w lokalizacji pliku

wykonywalnego; menu wyświetla się od razu. Z programu należy korzystać według poleceń wyświetlanych na bieżąco po każdej wykonanej czynności. W początkowym menu jest jednak jedna rzecz, która nie jest jawnie widoczna. Otóż po wpisaniu cyfry `4` i zatwierdzeniu wyboru klawiszem `enter` program poprosi nas o wpisanie loginu. Należy wpisać `wiewiórka`. Następnie należy wpisać hasło: `czas`. Po tych czynnościach wyświetli się specjalny panel dla pracownika banku. Należy z niego korzystać według instrukcji wyświetlonych na ekranie. Kolejnym istotnym faktem jest to, że gdy chce się wyjść z programu, powinno się w menu głównym wpisać `0`, ponieważ w przypadku wyjścia za pomocą „krzyżyka", dane się nie zapiszą.

4 Specyfikacja wewnętrzna

4.1 Typy zdefiniowane w programie

W programie zdefiniowano następujący typ:

```
string nazwa_miesiaca [12] =
{
    ''stycznia'',
    ''lutego'',
    ''marca'',
    ''kwietnia'',
    ''maja'',
    ''czerwca'',
    ''lipca'',
    ''sierpnia'',
    tr(''wrzesnia''),
    tr(''pazdziernika''),
    ''listopada'',
    ''grudnia''
};
```

Typ ten jest wykorzystywany do wyświetlania pełnych nazw miesięcy, zamiast tylko ich numerów.

Zdefiniowano także typ służący do przechowywania dat.

```
struct datum {
    int dzien;
    int miesiac;
    int rok;
};
```

Zdefiniowano także typ służący do przechowywania adresów.

```
struct adres {
    string ulica;
    string nr_domu;
    string miasto;
    string kraj;
};
```

Zdefiniowano także typ służący do zbudowania listy transakcji pojedynczego klienta banku.

```
struct transakcja {
    double kwota;                    //kwota transakcji (
        ujemna jezeli uzytkownik wyplacal lub puszczal do
         kogos przelew)
    datum data;                      //data wykonania
        transakcji
    transakcja* nastepna;  //wskaznik do nastepnej
        transakcji, nullptr jezeli transakcja jest
        ostatnia
};
typedef transakcja* wsk_transakcja;
```

Zdefiniowano także typ służący do zbudowania głównej listy kont bankowych.

```
struct konto {
    bool czy_konto_usuniete;              //informacja o
        tym czy uzytkownik usunal konto (1 - TAK, 0 - NIE
        )
    string login;
    string haslo;
    string imie;
    string drugie_imie;                           //jezeli
        ktos nie ma, to jest tam wpisane 0
    string trzecie_imie;                          //jezeli
        ktos nie ma, to jest tam wpisane 0
    string nazwisko;
    datum data_urodzenia;
    datum data_zalozenia_konta;
    string pesel;
    double limit_debetu;                    //wyrazony
        liczba dodatnia
```

```
13    string nr_konta;
14    double stan_konta;
15    adres adres_zamieszkania;
16    adres adres_korespondencji;     //jezeli taki sam jak
          adres zamieszkania, to w polu ,,ulica'' jest
          wpisane 0
17    wsk_transakcja transakcje;       //wskaznik do
          pierwszej transakcji wykonanej przez uzytkownika
          konta
18    int ile_transakcji;                              //
          informacja o tym, ile transakcji uzytkownik
          wykonal od czasu zalozenia konta
19    konto* poprzedni;                          //wskaznik do
          poprzedniego konta, nullptr jezeli konto jest
          pierwsze
20    konto* nastepny;                          //wskaznik do
          nastepnego konta, nullptr jezeli konto jest
          ostatnie
21 };
22 typedef konto* wsk_konto;
```

Zdefiniowano także typ służący do opisu głównej listy (str. 5)

```
1 struct opis_listy {
2    wsk_konto pierwsze;  //wskaznik do pierwszego
         elementu listy
3    wsk_konto ostatnie;   //wskaznik do ostatniego
         elementu listy
4    int liczba_kont;            //liczba wszystkich kont
5 };
```

4.2 Ogólna struktura programu

W funkcji głównej najpierw deklarowana jest struktura:

```
1 opis_listy konta;
```

Następnie wywoływana jest funkcja

```
1 odczytaj_dane_z_pliku(opis_listy & konta);
```

Funkcja ta odczytuje dane z pliku dane_kont.kurczak oraz plików transakcje_login .kurczak i umieszcza je w głównej strukturze danych wykorzystanej w pro-

gramie (patrz str. 2). Jeżeli pliki nie istnieją, to funkcja zainicjuje pustą strukturę. (patrz str. 14) Potem deklarowana jest zmienna

```
1 bool czy_byla_zmiana;
```

Której wartość ustawiana będzie jako **true**, kiedy nastąpi jakakolwiek zmiana, którą trzeba zapisać po wyjściu z programu. Następnie wyświetla się menu główne. Program prosi użytkownika o dokonanie odpowiedniego wyboru, po którego dokonaniu uruchamiane są odpowiednie funkcje, których dokładny opis znajduje się na str. 7. Po ukończeniu pracy z programem użytkownik wybiera 0, po czym, jeżeli wystąpiła jakakolwiek zmiana, którą należy zapisać, wywoływana jest funkcja

```
1 zapisz_dane_do_pliku(opis_listy & konta);
```

Funkcja ta zapisuje wszystkie dane do pliku dane_kont.kurczak oraz plików transakcje_login .kurczak (patrz str. 14). Na końcu wywoływana jest funkcja

```
1 zwolnij_pamiec(opis_listy & konta);
```

która, jak sama nazwa wskazuje, zwalnia pamięć przeznaczoną dla struktury opisanej na str.2.(patrz str. 13)

4.3 Szczegółowy opis implementacji funkcji

```
1 char *tr(char *str)
```

Funkcja umożliwia wyświetlanie polskich znaków. Parametrem jest ciąg znaków typu **string**. Funkcja zwraca ciąg znaków z podmienionymi polskimi znakami na odpowiednie znaki, dzięki czemu są one wyświetlane prawidłowo.

Funkcja istnieje_konto_login sprawdza czy istnieje konto o podanym loginie. Zwraca ona **true**, jeżeli konto zostało znalezione lub **false**, jeżeli nie. Funkcja została zaimplementowana w następujący sposób:

```
1 bool istnieje_konto_login(string login, opis_listy
    konta)
2 // login - login konta, ktore chcemy sprawdzic
3 // konta - opis listy
4 {
5     bool znaleziono = false;
6     wsk_konto pom;
7     pom = konta.pierwsze;
8     while (znaleziono == false && pom != nullptr)
```

```
9     {
10        if (pom->login == login && pom->
              czy_konto_usuniete == false)
11            znaleziono = true;
12        pom = pom->nastepny;
13    }
14    return znaleziono;
15 }
```

```
1 bool istnieje_konto_login_haslo(string login, string
    haslo, opis_listy konta)
```

Funkcja działa analogicznie do funkcji istnieje_konto_login . Różnica polega na tym, że posiada ona dodatkowy parametr haslo - hasło konta, które chcemy sprawdzić oraz sprawdza czy istnieje konto o podanym loginie i haśle (login i hasło muszą należeć do tego samego konta). Zwraca to samo, co poprzednia funkcja.

```
1 void ustaw_wskaznik_na_konto(wsk_konto & pom,
    opis_listy konta, string login)
```

Funkcja ustawia podany wskaźnik na konto o podanym loginie. Będzie ona wielokrotnie używana, w celu zlokalizowania miejsca w pamięci, gdzie znajduje się konto o podanym loginie. Jako parametry przyjmuje pom - wskaźnik, który chcemy ustawić na dane konto, login - login konta, na które chcemy ustawić wskaźnik oraz konta - opis listy.

```
1 void wplata_dowolnej_kwoty(string login, opis_listy
    konta, double kwota)
2 {
3     wsk_konto pom;
4
5     ustaw_wskaznik_na_konto(pom, konta, login);
6
7     if (pom == nullptr)
8     {
9         cout << tr("Blad! Takie konto nie istnieje.") <<
              endl;
10    }
11    else
12    {
13        pom->stan_konta += kwota;
```

```
14          pom->ile_transakcji++;
15
16          wsk_transakcja nowa = new transakcja;
17          nowa->kwota = kwota;
18          nowa->data.dzien = obecny_dzien();
19          nowa->data.miesiac = obecny_miesiac();
20          nowa->data.rok = obecny_rok();
21          nowa->nastepna = nullptr;
22
23          if (pom->transakcje == nullptr)
24          {
25              pom->transakcje = nowa;
26          }
27          else
28          {
29              wsk_transakcja pom1;
30              pom1 = pom->transakcje;
31              while (pom1->nastepna != nullptr)
32              {
33                  pom1 = pom1->nastepna;
34              }
35              pom1->nastepna = nowa;
36          }
37      }
38  }
```

Funkcja ta dodaje na koniec listy transakcji danego konta transakcje z datą kiedy była ona wykonywana oraz kwotą równą parametrowi kwota. Zwiększa ona również stan konta o parametr kwota. Ta funkcja jest miejscem, gdzie alokowana jest pamięć dla nowych transakcji. Jako parametry przyjmuje ona login - login konta, na którym wykonywana jest operacja, konta - opis listy oraz kwota - kwota transakcji.

```
1  void odwolanie(string login, opis_listy & konta)
2  {
3      wsk_konto pom;
4      ustaw_wskaznik_na_konto(pom, konta, login);
5      if (pom != nullptr && pom->poprzedni != nullptr)
6      {
7          wsk_konto pom_poprzedni = pom->poprzedni;
8          wsk_konto pom_poprzedni_poprzedni = pom_poprzedni
               ->poprzedni;
```

```
9         wsk_konto pom_nastepny = pom->nastepny;
10
11        pom->poprzedni = pom_poprzedni_poprzedni;
12        pom->nastepny = pom_poprzedni;
13
14        pom_poprzedni->poprzedni = pom;
15        pom_poprzedni->nastepny = pom_nastepny;
16
17        if (pom_poprzedni_poprzedni != nullptr)
18        {
19            pom_poprzedni_poprzedni->nastepny = pom;
20        }
21        else
22        {
23            konta.pierwsze = pom;
24        }
25
26        if (pom_nastepny != nullptr)
27        {
28            pom_nastepny->poprzedni = pom_poprzedni;
29        }
30        else
31        {
32            konta.ostatnie = pom_poprzedni;
33        }
34    }
35 }
```

Funkcja ta zamienia miejscami konto, do którego nastąpiło jakieś odwołanie, z kontem znajdującym się o 1 miejsce w stronę początku listy. Jako parametry przyjmuje ona login - login konta, do którego nastąpiło odwołanie, konta - opis listy.

```
1 string jaki_login(string nr_konta, opis_listy konta)
```

Funkcja zwraca login konta o podanym numerze, jeżeli takie konto nie istnieje, zwraca ona ''0''. Jako parametry przyjmuje: nr_konta - numer konta oraz konta - opis listy.

```
1 bool czy_debet_przekroczony(string login, opis_listy
    konta, double kwota)
```

Funkcja ta sprawdza, czy po wykonaniu transakcji na podaną kwotę, debet na koncie o podanym loginie zostanie przekroczony. Zwraca ona **true**, jeżeli tak się stanie. W przeciwnym wypadku - **false**. Przyjmuje jako parametry: login - login konta, konta - opis listy oraz kwota - kwota transakcji.

```
1 bool data_jest_wieksza(datum ta_ktora_ma_byc_wieksza,
    datum ta_od_ktorej_ma_byc_wieksza)
```

Funkcja jako parametry przyjmuje ta_ktora_ma_byc_wieksza - data transakcji oraz ta_od_ktorej_ma_byc_wieksza - data od kiedy mają być wyświetlane transakcje. Sprawdza ona, czy data danej transakcji jest młodsza od daty, od której chcemy mieć wyświetlane transakcje. Zwraca **true**, jeżeli tak jest lub **false**, jeżeli tak nie jest. Funkcja ta jest używana w funkcji wyświetlającej transakcje w podanym okresie.

```
1 bool data_jest_mniejsza(datum ta_ktora_ma_byc_mniejsza,
     datum ta_od_ktorej_ma_byc_mniejsza)
```

Funkcja działa analogicznie do poprzedniej funkcji. Z tym, że sprawdza ona czy data danej transakcji jest starsza od daty, do której chcemy mieć wyświetlane transakcje.

```
1 void wyciag_jeden_klient(string login, opis_listy konta
    , datum poczatek, datum koniec)
```

Funkcja wyświetla wszystkie transakcje danego klienta w podanym przedziale czasowym. Jej parametry, to login - login klienta, którego transakcje chcemy wyświetlić, konta - opis listy oraz poczatek, koniec - odpowiednio: data, od kiedy chcemy mieć wyciąg; data, do kiedy chcemy mieć wyciąg.

```
1 void dane_klienta(string login, opis_listy konta)
```

Funkcja wyświetla wszystkie dane klienta. Jako parametry przyjmuje login - login klienta, którego dane chcemy wyświetlić oraz konta - opis listy.

```
1 void zmiana_wybranej_danej(string &login, opis_listy
    konta, int ktora_dana)
```

Funkcja ta umożliwia klientowi zmianę wybranej danej, dotyczącej swojego konta. Najpierw prosi ona o podanie nowej wartości danej, a następnie przypisuje tą nową wartość do odpowiedniej zmiennej. Jako parametry przyjmuje: login - login klienta, który chce zmienić swoją daną, konta - opis listy oraz ktora_dana - wartość dzięki której funkcja się dowiaduje, którą daną należy zmienić odpowiednio: 0 - login, 1 - hasło, 2 - imię, 3 - drugie imię, 4 - trzecie imię, 5 - nazwisko, 6 - data urodzenia, 7 - pesel, 8 - limit debetu, 9 - adres

zamieszkania, 10 - adres korespondencji. Funkcja ta będzie wykorzystywana w poniższej funkcji:

```
1 void zmiana_danych(string & login, opis_listy konta)
```

Funkcja ta prosi użytkownika o wybranie, jaką daną chce zmienić, a następnie wywołuje funkcję zmiana_wybranej_danej (patrz str. 11) z wartością parametru ktora_dana, odpowiadającą danej, którą użytkownik chce zmienić. Jej parametry to: login - login klienta, który chce zmienić swoją daną oraz konta - opis listy.

```
1  void utworz_konto(string login, string haslo,
       opis_listy &konta)
2  {
3      wsk_konto nowy = new konto;
4      nowy->login = login;
5      nowy->haslo = haslo;
6      nowy->czy_konto_usuniete = false;
7      nowy->data_zalozenia_konta.dzien = obecny_dzien();
8      nowy->data_zalozenia_konta.miesiac = obecny_miesiac
           ();
9      nowy->data_zalozenia_konta.rok = obecny_rok();
10     nowy->nr_konta = numer_konta();
11     nowy->stan_konta = 0;
12     nowy->transakcje = nullptr;
13     nowy->ile_transakcji = 0;
14     nowy->nastepny = nullptr;
15     if (konta.pierwsze == nullptr)
16     {
17         konta.pierwsze = nowy;
18         nowy->poprzedni = nullptr;
19     }
20     else
21     {
22         konta.ostatnie->nastepny = nowy;
23         nowy->poprzedni = konta.ostatnie;
24     }
25     konta.ostatnie = nowy;
26     konta.liczba_kont++;
27 }
```

Funkcja ta jest miejscem, w którym alokowana jest pamięć dla nowego konta bankowego. Tworzy ona nowe konto o podanym loginie i haśle na końcu listy.

Przypisuje do konta datę, kiedy zostało ono założone, przypisuje automatycznie wygenerowany numer konta. Przypisuje do zmiennej czy_konto_usuniete wartość **false**. Przypisuje do zmiennej stan_konta wartość 0 oraz przypisuje do wskaźnika transakcje wartość nullptr. Jako parametry przyjmuje login - login do nowego konta, haslo - hasło do nowego konta oraz konta - opis listy.

```
1  void zwolnij_pamiec(opis_listy &konta)
2  {
3      wsk_konto pom = konta.pierwsze;
4      while (pom != nullptr)
5      {
6          if (pom->transakcje != nullptr)
7          {
8              wsk_transakcja poprzednia = pom->transakcje;
9              wsk_transakcja nast = poprzednia->nastepna;
10             delete poprzednia;
11             while (nast != nullptr)
12             {
13                 poprzednia = nast;
14                 nast = nast->nastepna;
15                 delete poprzednia;
16             }
17         }
18         if (pom->nastepny == nullptr)
19         {
20             delete pom;
21             pom = nullptr;
22         }
23         else
24         {
25             pom = pom->nastepny;
26             delete pom->poprzedni;
27         }
28     }
29 }
```

Funkcja ta jest miejscem, w którym zwalniana jest cała pamięć zaalokowana dla głównej struktury (patrz str. 2). Jako parametr przyjmuje tylko konta - opis listy.

```
1  string numer_konta()
```

Funkcja ta losuje ostatnie 16 cyfr numeru konta bankowego, następnie zwraca łańcuch znaków w postaci 6510100002 + 16 wylosowanych znaków.

```
1 int obecny_dzien()
```

Funkcja zwraca numer dnia z przedziału od 1 do 31, który jest w momencie jej wywołania.

```
1 int obecny_miesiac()
```

Funkcja zwraca numer miesiąca z przedziału od 0 do 11, który jest w momencie jej wywołania.

```
1 int obecny_rok()
```

Funkcja zwraca numer roku, który jest w momencie jej wywołania.

```
1 void odczytaj_dane_z_pliku(opis_listy &konta)
```

Funkcja ta jest miejscem, w którym dane z plików są odczytywane oraz w którym dla odczytanych danych alokowana jest pamięć. Funkcja sprawdza, czy plik z danymi dane_kont.kurczak istnieje. Jeżeli nie, to tworzy pustą strukturę (patrz str. 2), nie posiadającą żadnych kont. Jeżeli istnieje to odczytuje odpowiednie dane z pliku dane_kont.kurczak, alokuje dla nich pamięć i tworzy odpowiednią listę. Ponadto dla każdego konta sprawdza ona, czy istnieje plik o nazwie transakcje_login .kurczak, gdzie login, to login rozpatrywanego konta oraz jeżeli taki plik nie istnieje, to ustawia wartość wskaźnika transakcje znajdującego się w rozpatrywanym koncie na nullptr, a jeżeli ten plik istnieje, to wczytuje dane z tego pliku, alokuje dla nich pamięć i tworzy odpowiednią listę transakcji rozpatrywanego konta. Jej jedyny parametr to konta - opis listy.

```
1 void zapisz_dane_do_pliku(opis_listy &konta)
```

Funkcja ta usuwa starą zawartość plików z danymi (dane_kont.kurczak oraz transakcje_login .kurczak) (jeżeli istniały (jeżeli nie istniały, to tworzy nowe)) oraz zapisuje do niej aktualne dane. Jej jedyny parametr to konta - opis listy.

5 Testowanie

Program został przetestowany z różnymi danymi wejściowymi. Za każdym razem, gdy dane były prawidłowe, program odpowiadał poprawnie. Gdy dane były nieprawidłowe, lecz były odpowiednim typem danych (np. **double**, gdy należało wpisać dane typu **double**); system zabezpieczeniowy programu

działał poprawnie, wyświetlając użytkownikowi informacje, że dane były niepoprawne i prosząc go o wpisanie danych jeszcze raz. Natomiast, gdy wprowadzane dane były innego typu niż powinny być (np. **char**, gdy wymagany był **int**), to program reagował w bliżej niezidentyfikowany sposób.

6 Wnioski

Program do zarządzania bankiem był dla mnie nie lada wyzwaniem, tym bardziej, że nie uczyłem się programowania w liceum. Dzięki temu projektowi zdołałem się nauczyć wielu wspaniałych rzeczy, jak chociażby obsługa plików binarnych, operacje na strukturach dynamicznych, czy odpowiednie wykorzystywanie funkcji. Pisanie tego projektu było też dla mnie świetnym przygotowaniem do zbliżającego się egzaminu oraz przyszłej pracy.