Rozdział 6

Struktury

6.1 Wprowadzenie do programowania orientowanego obiektowo

Wprowadzenie do programowania orientowanego obiektowo zostanie wyjaśnione na praktycznym przykładzie. Załóżmy, że naszym zadaniem jest zaprogramowanie systemu bibliotecznego pozwalającego na zarządzanie wypożyczaniem, zwracaniem i rezerwowaniem książek. Oto jak działa nasz system:

- Każdy czytelnik ma swój unikatowy numer identyfikacyjny
- Każda książka ma numer dostępu
- Czytelnik może mieć wypożyczone maksymalnie 3 książki; może wypożyczać w dowolnym czasie
- Jeśli czytelnik A ma aktualnie wypożyczone 3 książki może zarezerwować książkę wypożyczoną przez osobę B. W chwili gdy książka zostanie zwrócona przez B jest ona przetrzymywana dla A i nie może zostać wypożyczona innym czytelnikom
- Czytelnik może zwrócić książkę w dowolnym czasie bez kar za opóźnienie

Zadanie można zrealizować za pomocą standardowych zmiennych i metodą programowania strukturalnego. Przy założeniu, że jest 1000 książek i tytuły oraz imię i nazwisko autorów nie przekraczają 100 znaków, dla każdej książki można zdefiniować:

- Tytuł (`string tytulKsiazki[1000][100]`)
- Autor (`string autor[1000][100]`)
- Cena w zł (`double cena[1000]`)
- Numer dostępu (`int nrDostepu[1000]`)

- Status wypożyczenia (`bool statusWypozyczenia[1000]`)
- Numer identyfikacyjny rezerwującego (jeśli jest) (`int nrIdentyfikacyjnyRezerwujacego[1000]`)

Podobnie dla każdego czytelnika można zdefiniować:

- Imię i nazwisko (`string imieNazwiskoCzyt[1000][100]`)
- Numer identyfikacyjny (`int nrIdentyfikacyjnyCzytelnika[1000]`)
- Liczba wypożyczonych książek (`int liczbaWypKsiazek[1000]`)
- Numery dostępu wypożyczonych książek (`int nrDostepuZarKsiazki[1000]`)

Teraz możemy zastanowić się jak to zaprogramować. Robimy założenie, że w bibliotece jest 1000 książek. Jednym ze sposobów mogłoby by być utworzenie różnych tablic o tysiącu elementów. Na przykład można by utworzyć 1000-elementową tablicę przechowującą tytuły książek. Każdy element tej tablicy zawiera tytuł innej książki. Podobnie byłoby z tablicą przechowującą imiona autorów. Pozostałe tablice też byłyby zwykłymi 1000-elementowymi tablicami.

Taki sposób implementacji naszego systemu oczywiście jest możliwy, jednak jest to niewygodne, że informacje o tej samej książce rozdzielamy na wiele różnych tablic zawierających różne typy zmiennych. Możemy zastanowić się jak dowiedzieć się wszystkich informacji o danej książce znając jej tytuł. Rozwiązaniem może być założenie, że informacje o konkretnej książce są przechowywane w elementach tablic o tym samy indeksie. Na przykład znając tytuł książki pod indeksem nr 10 moglibyśmy uzyskać pozostałe informacje jako: `tytulKsiazki[10]`, `autor[10]`, `cena[10]`, `nrDostepu[10]`, `statusWypozyczenia[10]`, `nrIdentyfikacyjnyRezerwujacego[10]`. Zauważmy jednak, że to działa tylko wtedy gdy znamy jaki jest indeks danej książki w tablicy. Jeśli go nie znamy, a chcemy dowiedzieć się jaki jest tytuł książki znając jej nr dostępu to najpierw musimy wykonać operację „znalezienia” indeksu w tablicy `nrDostepu[indeks]`, pod którym znajduje się znany kod dostępu. Znając indeks możemy następnie odczytać tytuł książki jako `tytul[indeks]` Widać, że jest to dodatkowa operacja wynikająca z implementacji informacji o jednym obiekcie w kilku różnych tablicach.

Lepszym sposobem jest utworzenie obiektu `Ksiazka`, który zawierałby wszystkie informacje o danej książce. Następnie można by utworzyć tablicę takich obiektów. Plusem tego rozwiązania jest to, że przeszukując tą tablicę i znajdując element o zadanym kodzie dostępu można od razu wyświetlić tytuł szukanej książki. Podobnie można utworzyć obiekt `Czytelnik`, a następnie tablicę takich obiektów.

C++ pozwala na tworzenie tzw. *struktur*, które mogą grupować w jeden zestaw zmienne o różnych typach. W nawiasach klamrowych znajdują się składniki struktury (mogą to być zmienne lub funkcje). Składają się one z typu zmiennej oraz nazwy. W przypadku struktury Książka i Czytelnik kod wyglądałby następująco:

```
struct Ksiazka
{
    string tytulKsiazki;
    string autor;
    double cena;
    int nrDostepu;
    bool statusWypozyczenia;
    int nrIdentyfikacyjnyRezerwujacego; // -1 jesli zaden czytelnik
                                        // nie zarezerwowal
};
```

Listing 6.1: Struktura Ksiazka

```
struct Czytelnik
{
    string imieNazwiskoCzyt;
    string adres;
    int nrIdentyfikacyjnyCzytelnika;
    int liczbaWypKsiazek;
    int nrDostepuZarKsiazki;   // -1 jesli czytelnik nie zglaszal
                               // zadnej rezerwacji
};
```

Listing 6.2: Struktura Czytelnik

Jak widać struktury są tworzone przez słowo kluczowe `struct`. Następnie jest nazwa struktury i jej definicja w nawiasach klamrowych. Nazwa struktury może być użyta w kodzie tak jak każda inna zmienna, np.

```
int mojaLiczba;
Ksiazka mojaKsiazka;
Czytelnik janKowalski, przemekBlasiak;
Czytelnik czytelnicy[10];
```

Listing 6.3: Tworzenie struktur w kodzie programu

Na powyższym prostym przykładzie możemy omówić podstawowe cechy metodologii programowania obiektowo orientowanego:

1. Identyfikacja obiektów występujących w systemie (książki, czytelnicy)
2. Postrzeganie systemu z punktu widzenia operacji na obiektach i wzajemnej interakcji pomiędzy obiektami. Np. Czytelnik wypożycza książkę; czytelnik rezerwuje książkę, itp.
3. Ukrywanie szczegółów nie istotnych dla przeprowadzenia danej operacji
4. Implementacja systemu ze skupieniem się na klasach i ich wzajemnych interakcjach

W powyższym opisie często występującym słowem jest *obiekt*. Obiekt posiada następujące cechy:

1. Zawiera informacje na temat obiektu. Informacje te mogą być:

- Informacje stałe - zwykle nie zmieniają się gdy obiekt oddziałuje z innymi obiektami (np. tytuł książki)
- Informacje stanu - mogą się zmieniać (np. status wypożyczenia książki)

2. Ma jednoznacznie określone granice. Jasno określone jakie informacje są częścią obiektu (np. Gdzie są przechowywane informacje kiedy czytelnik wypożycza książkę? nrIdentyfikacyjnyCzytelnika jest w obiekcie Czytelnik, nrDostepu jest obiekcie Ksiazka)

Idealnie gdy komunikacja między obiektami odbywa się przez jasno określone *interfejsy* obiektów. Osiąga się to przez ukrywanie niepotrzebnych szczegółów i tzw. enkapsulację o czym będzie później.

6.2 Proste operacje na strukturach

6.2.1 Dostęp do składników struktury

Przypomnijmy kod struktury Ksiazka i mamy zdefiniowany jeden obiekt o nazwie `mojaKsiazka`:

```
struct Ksiazka
{
    string tytulKsiazki;
    string autor;
    double cena;
    int nrDostepu;
    bool statusWypozyczenia;
    int nrIdentyfikacyjnyRezerwujacego;
};

Ksiazka mojaKsiazka;
```

Listing 6.4: Struktura Ksiazka

W celu dostępu do np. zmiennej `nrDostepu` C++ wprowadza tzw. „operator kropki” „ . ”. W kodzie wyglądałoby to następująco

```
mojaKsiazka.nrDostepu;
```

Listing 6.5: Dostęp do składnika struktury

Ponadto zmienne te możemy wykorzystywać jak każde inne zmienne wbudowane, np.:

```
cin >> mojaKsiazka.nrDostepu;
cout << mojaKsiazka.nrDostepu << endl;
mojaKsiazka.tytulKsiazki = "Potop";
cout << mojaKsiazka.tytulKsiazki << endl;
```

Listing 6.6: Przykładowe użycie składnika struktury w programie

6.2.2 Inicjalizacja struktur

Przypomnijmy deklarację struktury `Czytelnik`

```
struct Czytelnik
{
    string imieNazwiskoCzyt;
    string adres;
    int nrIdentyfikacyjnyCzytelnika;
    int liczbaWypKsiazek;
    int nrDostepuZarKsiazki;
};
```

Listing 6.7: Struktura Czytelnik

W celu inicjalizacji struktury możemy zastosować przykładowy kod:

```
Czytelnik czytNr1 = {"Przemek Blasiak", "Plac Grunwaldzki 9", 123,
    0, -1};
  cout << czytNr1.imieNazwiskoCzyt << endl;
  cout << czytNr1.adres << endl;
  cout << czytNr1.nrIdentyfikacyjnyCzytelnika << endl;
  cout << czytNr1.liczbaWypKsiazek << endl;
  cout << czytNr1.nrDostepuZarKsiazki << endl;
```

Listing 6.8: Przykład inicjalizacji struktur

Założyliśmy tutaj, że w chwili tworzenia danego obiektu typu `Czytelnik`, jego `liczbaWypKsiazek` i `nrDostepuZarKsiazek` są odpowiednio równe 0 i -1. Liczba -1 oznacza, że czytelnik nie ma zarezerwowanych żadnych książek. W przypadku struktury `Ksiazka` zakładamy, że w chwili inicjalizacji mamy `statusWypozyczenia = false` oraz `nrIdentyfikacyjnyRezerwujacego = -1`.

6.2.3 Kopiowanie struktur

Struktury możemy kopiować tak jak zmienne wbudowane. Przykład pokazano poniżej.

```
//- Kopiowanie struktur
  Czytelnik czytNr1 = {"Przemek Blasiak", "Plac Grunwaldzki 9", 123,
     0, -1};
  Czytelnik czytNr2 = {"Jan Kowalski", "Plac Dominikanski 99", 321,
    3, 123456789};
  cout << czytNr2.imieNazwiskoCzyt << endl;
  cout << czytNr2.adres << endl;
  cout << czytNr2.nrIdentyfikacyjnyCzytelnika << endl;
  cout << czytNr2.liczbaWypKsiazek << endl;
  cout << czytNr2.nrDostepuZarKsiazki << endl;

  czytNr2 = czytNr1;

  cout << czytNr2.imieNazwiskoCzyt << endl;
  cout << czytNr2.adres << endl;
  cout << czytNr2.nrIdentyfikacyjnyCzytelnika << endl;
  cout << czytNr2.liczbaWypKsiazek << endl;
```

```
16  cout << czytNr2.nrDostepuZarKsiazki << endl;
```

Listing 6.9: Przykład kopiowania struktur

Każdy składnik struktury `czytNr1` jest skopiowany do odpowiedniego składnika `czytNr2`.

6.3 Funkcje w strukturach

Oprócz właściwości takich jak np. wiek, imie, nazwisko, tytul, itp. struktury mogą zawierać też funkcje, które są zwane **metodami**. Przykładowo moglibyśmy dodać do struktury Czytelnik funkcję wyświetlającą imię i nazwisko. Jest to pokazane w listingu 6.10.

```
1  struct Czytelnik
2  {
3      string imieNazwiskoCzyt;
4      string adres;
5      int nrIdentyfikacyjnyCzytelnika;
6      int liczbaWypKsiazek;
7      int nrDostepuZarKsiazki;
8      void pokaz();
9  };
10
11 void Czytelnik::pokaz()
12 {
13     cout << imieNazwiskoCzyt << endl;
14 }
```

Listing 6.10: Struktura Czytelnik z dodaną metodą imieNazwisko()

Funkcje zwykle definiujemy poza obrębem struktury, często w osobnym pliku o nazwie o postaci *nazwaStruktury.cpp* (np. Czytelnik.cpp). W takim przypadku konieczne jest podanie zakresu funkcji, który w tym przypadku jest zakresem struktury Czytelnik. Robimy to przez podanie przed nazwą funkcji nazwy struktury i następnie operatora zakresu :: .

6.4 Konstruktory

Konstruktory to specjalne metody (funkcje) struktury o nazwie takiej jak nazwa struktury. Ich drugą cechą jest to, że nie zwracają niczego (nawet void). Służą one do tworzenia obiektów danej struktury, bowiem w momencie gdy definiujemy obiekt np. typu Czytelnik uruchamiany jest w tym momencie konstruktor, czyli funkcja "konstruująca"taki obiekt. Jeśli nie napiszemy własnej definicji konstruktora uruchomiony zostanie tzw. konstruktor domyślny. Konstruktory zwykle są używane do inicjalizacji pól danego obiektu. Obiekty mogą być tworzone w różny sposób, więc w myśl zasady o przeładowaniu nazw funkcji możemy mieć wiele konstruktorów, tylko muszą się one różnić ilością lub typem przyjmowanych argumentów. Konstruktor domyślny uruchamiany jest jako nie

przyjmujący żadnego argumentu. Przykład dwóch konstruktorów dla struktury Czytelnik pokazano w listingu 6.11.

```
struct Czytelnik
{
    string imieNazwiskoCzyt;
    string adres;
    int nrIdentyfikacyjnyCzytelnika;
    int liczbaWypKsiazek;
    int nrDostepuZarKsiazki;
    Czytelnik();
    Czytelnik(string in);
};

Czytelnik::Czytelnik()
{
    imieNazwiskoCzyt = "brak nazwy";
    adres = "brak adresu";
    nrIdentyfikacyjnyCzytelnika = -1;
    liczbaWypKsiazek = -1;
    nrDostepuZarKsiazki = -1;
}

Czytelnik::Czytelnik(string in)
{
    imieNazwiskoCzyt = in;
    adres = "brak adresu";
    nrIdentyfikacyjnyCzytelnika = -1;
    liczbaWypKsiazek = -1;
    nrDostepuZarKsiazki = -1;
}
```

Listing 6.11: Struktura Czytelnik z dodanymi dwoma konstruktorami

6.5 Destruktory

Destruktor to specjalna funkcja uruchamiana w momencie usuwania danego obiektu. Nic nie zwraca i nie przyjmuje argumentów. Jej nazwa jest taka sama jak nazwa struktury tylko poprzedzona znakiem tyldy. Np. destruktor dla struktury Czytelnik jest pokazany w listingu 6.12. W destruktorach często jest zwalniana pamięć alokowana wcześniej operatorem delete.

```
struct Czytelnik
{
    string imieNazwiskoCzyt;
    string adres;
    int nrIdentyfikacyjnyCzytelnika;
    int liczbaWypKsiazek;
    int nrDostepuZarKsiazki;
    Czytelnik();
    Czytelnik(string in);
    ~Czytelnik();
```

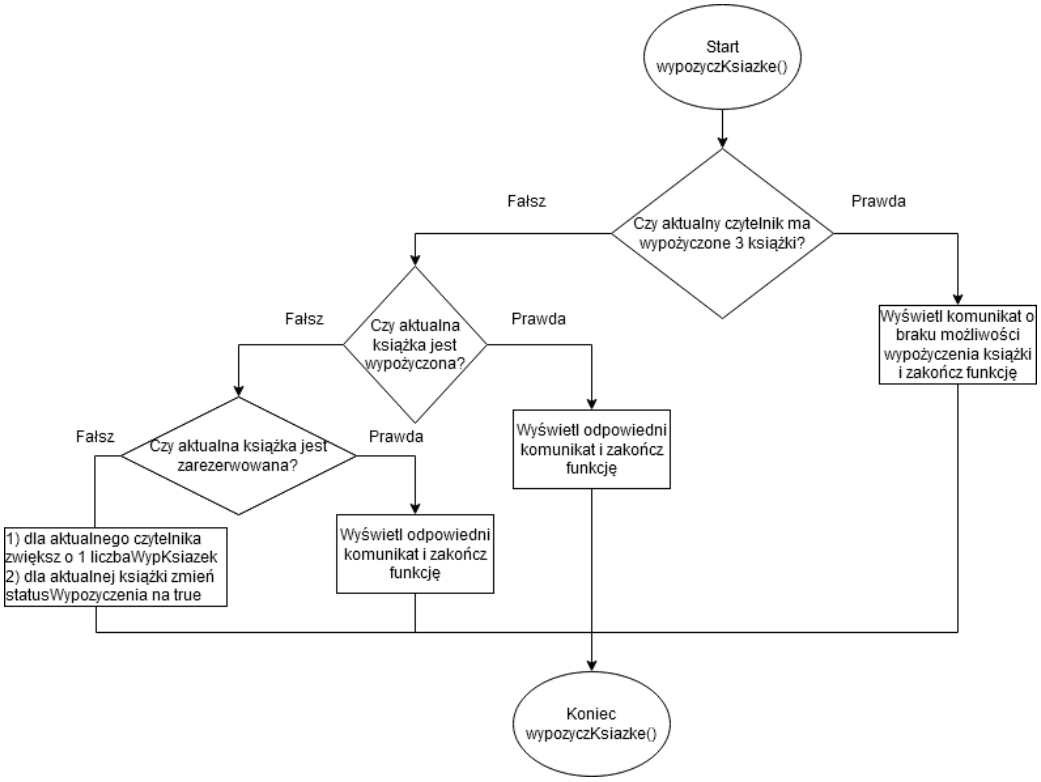
```
};

Czytelnik::~Czytelnik()
{
}
```

Listing 6.12: Struktura Czytelnik - destruktor

6.6 Programowanie z użyciem struktur

Teraz naszym zadaniem będzie implementacja „zachowań” naszych struktur. Jako pierwsze zachowanie zauważamy „wypożyczanie” książki przez czytelnika. Możemy to opisać jako: obiekt typu `Czytelnik` wypożycza książkę (obiekt typu `Ksiazka`). Przypuśćmy, że nasza funkcja będzie nazywać się `wypozyczKsiazke` oraz będzie wykonywać czynności wg schematu pokazanego na rys. 6.1.



Rys. 6.1 Algorytm funkcji `wypozyczKsiazke`

Zadanie nr 1

Zaimplementuj funkcję `void wypozyczKsiazke(Czytelnik& czytelnik, Ksiazka& ksiazka)`.

Ze względu na to, że po zakończeniu działania funkcji chcemy, aby zmienne `liczbaWypKsiazek` i `statusWypozyczenia` zostały uaktualnione, argumenty przekazywane do `wypozyczKsiazke` muszą być przekazywane przez referencję.

6.7 Struktury zawierające struktury

Załóżmy, że chcemy zaprogramować punkty i okręgi w przestrzeni dwuwymiarowej. Każdy punkt ma współrzędne x i y natomiast okręgi opisują środek (punkt) oraz promień. Definicja struktury `Punkt` byłaby następująca:

```
struct Punkt
{
    double x;
    double y;
};
```

Listing 6.13: Struktura `Punkt`

Natomiast kod dla struktury `Ograg` byłby

```
struct Okrag
{
    Punkt srodek;
    double promien;
};
```

Listing 6.14: Struktura `Okrag`

Załóżmy, że mamy zdefiniowany jeden obiekt typu `Okrag`. Poniższy kod prezentuje jak uzyskać dostęp do każdego składnika

```
Okrag mojOkrag;
mojOkrag.srodek.x = 1.2;
mojOkrag.srodek.y = 4.6
mojOkrag.promien = 5.0;
```

Listing 6.15: Dostęp do składników struktury zawierającej struktury

6.8 Struktury i wskaźniki

Przypomnijmy sobie proste operacje związane z wskaźnikami dla zmiennych wbudowanych

```
int a;
//utworzenie wskaznika do zmiennej typu int
int *wskA;
//wpisanie poczatku adresu zmiennej a do wskaznika wskA
wskA = &a;
//wpisanie wartosci do zmiennej a przez wskaznik wskA
*wskA = 20;
```

Listing 6.16: Podstawowe operacje ze wskaźnikiem

W dokładnie ten sam sposób możemy definiować wskaźniki do struktur i wykonywać z ich pomocą te same operacje

```
struct MojaStruktura
{
    double a;
    double b;
    char   c;
};

MojaStruktura ms;
//utworzenie wskaznika do zmiennej typu mojaStruktura
MojaStruktura *wsk;
//wpisanie poczatku adresu zmiennej ms do wskaznika wsk
wsk = &ms;
//wpisanie wartosci do zmiennej ms przez wskaznik wsk
*wsk = {20,400,'d'};
```

Listing 6.17: Podstawowe operacje ze wskaźnikiem

W celu dostępu do zmiennej `ms` przez wskaźnik `wsk` mamy dwie możliwości

```
//1. sposob
(*wsk).a = 10.5;
//2. sposob
wsk->x = 10.5;
```

Listing 6.18: Dostęp do składnika struktury przez wskaźnik

6.8.1 Struktury zawierające wskaźniki

Naszym zadaniem będzie zaprogramowanie prostego systemu kolejkowego dla taksówek. Tworzymy następujące struktury

```
struct Kierowca
{
  string imie;
  int  nr;
};
struct Taksowka
{
  int nr;
  Kierowca *kier;
};
```

Listing 6.19: Struktury `Kierowca` i `Taksowka`

W naszym programie chcemy, aby taksówka w kolejce miała informacje o następnej taksówce w kolejce. Można to osiągnąć przez drobną modyfikację struktury `Taksowka`

```
struct Taksowka
{
  int nr;
  Kierowca *kier;
  Taksowka *nastepna;
```

```
6 };
```

Listing 6.20: Modyfikacja struktury `Taksowka`

Jak widać w linijce nr 5 został dodany wskaźnik do `Taksowka`. Dzięki temu możliwe jest uzyskanie informacji jaka taksówka jest następna w kolejce. Oto przykładowy kod

```
1  int main()
2  {
3    Kierowca k1, k2;
4    Taksowka t1, t2;
5    k1 = {"Przemek", 25};
6    k2 = {"Jan", 54};
7    t1.nr = 100;
8    t1.kier = &k1;
9    t1.nastepna = nullptr;
10   t2.nr = 200;
11   t2.kier = &k2;
12   t2.nastepna = &t1;
13   std::cout << "Nastepny taksowkarz w kolejce to: " << (t2.nastepna)
       ->kier->imie << std::endl;
14 }
```

Listing 6.21: Przykładowy kod wykorzystujący wskaźnik wewnątrz struktury

6.9 Dynamiczna alokacja pamięci

Przypomnijmy sobie jak dokonywaliśmy dynamicznej alokacji pamięci dla zmiennych wbudowanych

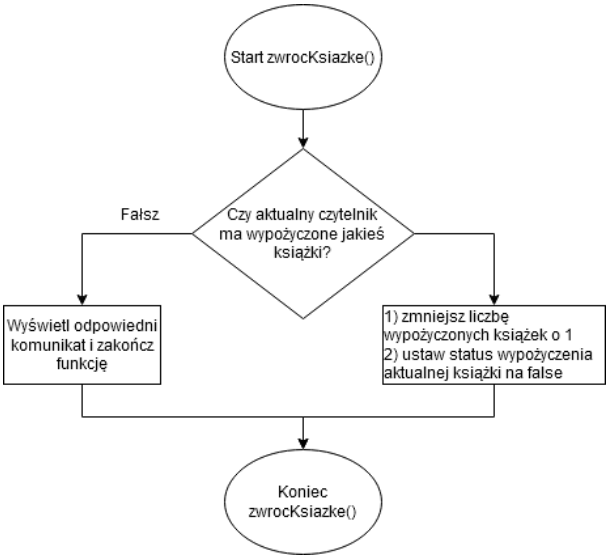
```
1 // dynamiczna alokacja pamieci dla pojedynczej zmiennej
2 int *mojWskaznikDoInt = new int;
3 // dynamiczna alokacja pamieci dla tablicy
4 int *mojWskaznikDoTablicyInt = new int [3];
5 // zwalnianie pamieci dla zmiennej wbudowanej
6 if (mojWskaznikDoInt != nullptr) delete mojWskaznikDoInt;
7 // zwalnianie pamieci dla tablicy
8 if (mojWskaznikDoTablicyInt != nullptr)
9     delete [] mojWskaznikDoTablicyInt;
```

Listing 6.22: Dynamiczna alokacja pamięci dla zmiennych wbudowanych

6.10 Zadania do rozwiązania

Zadania na ocenę 3,0

Zad. 1. Do zadania o systemie bibliotecznym przeprowadzonym na zajęciach dodaj funkcję `void zwrocKsiazke(Czytelnik& czytelnik, Ksiazka& ksiazka)`. Schemat funkcji przedstawiony jest na rys. 6.2. Następnie wywołaj tę funkcję w prostym programie.



Rys. 6.2 Algorytm funkcji `zwrocKsiazke`

Przedstawiona funkcja nie jest doskonała (np. sprawdź efekt podwójnego wywołania funkcji `zwrocKsiazke` dla tego samego czytelnika i książki), ale służy raczej zobrazowaniu używania struktur w programach.

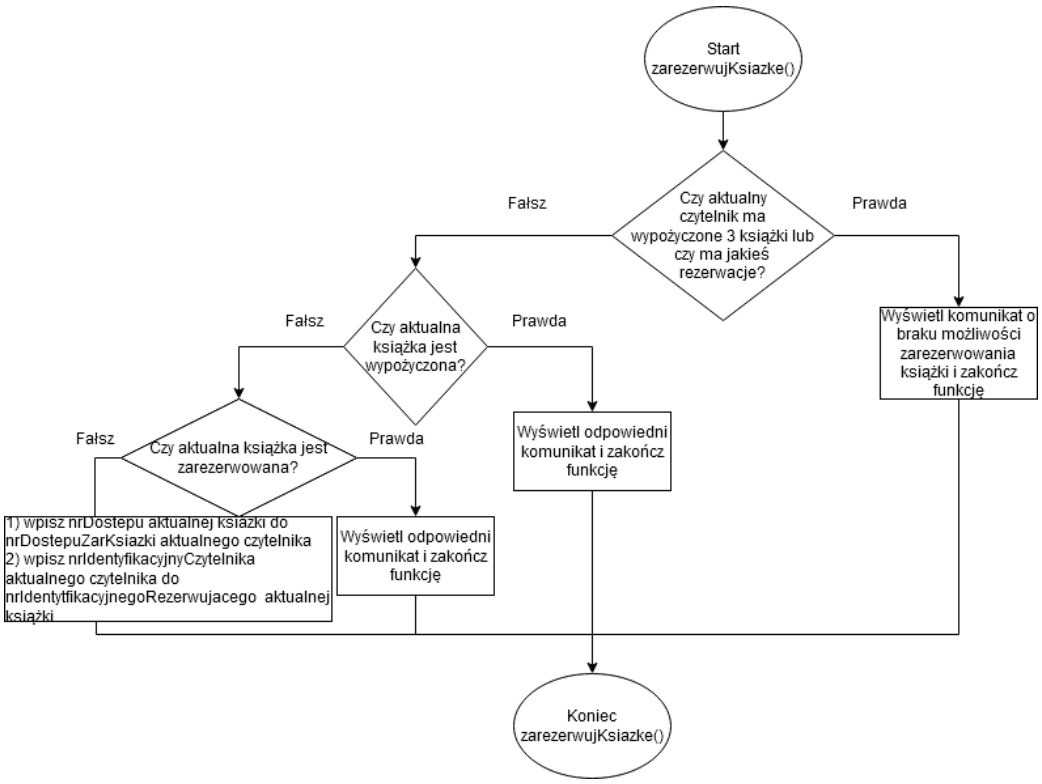
Zadania na ocenę 3,5-4,0

Zad. 2. Do zadania o systemie bibliotecznym przeprowadzonym na zajęciach dodaj funkcję `void zarezerwujKsiazke(Czytelnik& czytelnik, Ksiazka& ksiazka)`. Schemat funkcji przedstawiony jest na rys. 6.3. Następnie wywołaj tę funkcję w prostym programie.

Zadania na ocenę 4,5-5,0

Zad. 3. Zaimplementuj używając struktury:

1. obiekt typu *Punkt* mający:
   (a) nazwę punktu
   (b) 3 składowe współrzędnych kartezjańskich
   (c) konstruktor nie przyjmujący żadnych parametrów i ustawiający współrzędne na 0, nazwę na "bez nazwy"
   (d) konstruktor przyjmujący jako argumenty nazwę i 3 współrzędne
   (e) funkcję *pokaz* wyświetlającą składowe punktu na ekran
2. funkcję globalną o nazwie *odleglosc* obliczającą odległość między dwoma punktami



Rys. 6.3 Algorytm funkcji `zarezerwujKsiazke`

3. funkcję *czyTeSamePunkty* sprawdzającą czy dwa punkty są identyczne. (jak tak to zwraca 1 jak nie to 0). Dla liczb typu *double* niech sprawdza czy odległość między punktami jest mniejsza od zadanej małej liczby np. 1e-6

4. funkcję globalną *podajPunkty* wczytującą zadaną przez użytkownika liczbę punktów i zwracająca wynik w postaci wskaźnika do dynamicznej tablicy punktów. Funkcja powinna mieć zabezpieczenie, aby użytkownik nie podał dwóch takich samych punktów (użyj funkcji *czyTeSamePunkty*)

5. obiekt typu *Prostokat* mający:

   (a) cztery wierzchołki opisane za pomocą punktów (*Punkt*)

   (b) funkcje *pole* i *obwod* zwracające odpowiednio pole powierzchni i obwód prostokąta

   (c) konstruktor domyślny, przyjmujący 4 punkty, przyjmujący punkt początkowy i długości dwóch boków

6. program wczytujący dynamicznie cztery punty, które są umieszczane w tablicy - wykorzystaj funkcję *podajPunkty*. Następnie wyświetlana jest nazwa punktu, który jest najdalej oddalony od początku układu współrzędnych (funkcja *pokaz*).

Zadania nieobowiązkowe

Zad. 4. Lista jednokierunkowa oznacza, że można ją przebyć tylko w jednym kierunku od początku do końca.

Każdy element na takiej liście jest nazywany węzłem. Pojedynczy węzeł zawiera dane i wskaźnik do następnego węzła, co pomaga zachować strukturę listy.

Początek jest wskaźnikiem, który wskazuje na pierwszy węzeł listy, a tym samym pomaga nam poruszać się po liście. Ostatni węzeł wskazuje na *nullptr*, który pomaga nam określić, kiedy lista się kończy.

Zaimplementuj prostą listę jednokierunkową [6] określającą listę produktów w sklepie. Zadanie wykonaj w następujący sposób:

1. Utwórz strukturę *Produkt* zawierającą pola:
   (a) nazwa
   (b) nr produktu
   (c) data ważności
   (d) wskaźnik do następnego elementu
   (e) konstruktor domyślny ustawiający wskaźnik na nullptr
2. Utwórz strukturę *ListaProduktów* zawierającą:
   (a) wskaźnik do adresu początku listy (pierwszego elementu)
   (b) konstruktor domyślny ustawiający wskaźnik na nullptr
   (c) funkcję *dodaj* (element) - Możemy to zrobić wg następujących kroków:
      i. utworzenie dynamicznie nowego produktu
      ii. inicjalizacja pól w tym produkcie
      iii. sprawdzić czy to pierwszy element listy, jeśli tak to zrobić go początkiem listy
      iv. jeśli nie jest to pierwszy element to przejść na koniec listy i dla wskaźnika pokazującego na ostatni element ustawiamy go żeby pokazywał na ten nowy produkt i w nowym produkcie ustawiamy następny produkt jako nullptr
   (d) destruktor usuwający wskaźniki po dynamicznym alokowaniu pamięci
   (e) funkcję *usun* (element) - Tutaj należy pamiętać o kilku rzeczach:
      i. gdy usuwamy pierwszy element należy przesunąć wskaźnik pokazujący na początek listy na kolejny element
      ii. przy usuwaniu n-tego elementu trzeba zaprogramować, aby element n-1 pokazywał teraz na element n+1
      iii. przy usuwaniu ostatniego elementu ustawiamy wskaźnik elementu przedostatniego na nullptr

iv. należy zdecydować jak będzie wybierany element do usunięcia (np. przez podanie jako argumentu jego nazwy)

Algorytm usuwania elementu może wyglądać tak:

i. sprawdź czy to pierwszy element listy. Jak tak ustawiamy początek na drugi element i usuwamy stary element (delete)

ii. gdy nie jest to pierwszy element przesuwamy wskaźnik na element n-1

iii. sprawdzamy czy usuwamy ostatni element jeśli tak to zwalniamy pamięć i ustawiamy wskaźnik elementu przedostatniego na nullptr

iv. jeśli nie jest to ostatni element to ustawiamy wskaźnik na element n+1 i zwalniamy pamięć

(f) funkcję *sortuj* (elementy, np. wg daty ważności)

(g) funkcję *pokaz* (elementy)

3. w funkcji main utwórz listę obiektów, a następnie wczytaj kilka na listę za pomoca metody *dodaj*.

4. Napisz i wywołaj funkcję służąca do wyświetlania listy.

5. wywołaj funkcję *sortuj* i ponownie wyświetl listę.

6. usuń produkt z wnętrza listy (np. z okolicy jej środka) i wyświetl listę

Bibliografia

[1] cplusplus.com. Variables and types. https://www.cplusplus.com/doc/tutorial/variables/.

[2] cpp0x.pl/ Serwis programistyczny C++. Biblioteka <math.h>. https://cpp0x.pl/kursy/Kurs-C++/Dodatkowe-materialy/Biblioteka-math-h/322.

[3] cpp0x.pl/ Serwis programistyczny C++. Pseudolosowe liczby całkowite. https://cpp0x.pl/kursy/Kurs-C++/Poziom-2/Pseudolosowe-liczby-calkowite/290.

[4] GeeksforGeeks. Measure execution time of a function in c++. https://www.geeksforgeeks.org/measure-execution-time-function-cpp/.

[5] J. Grębosz. *Opus magnum C++11. Programowanie w języku C++*. Helion, 2019.

[6] programowanie;. Lista jednokierunkowa. https://www.p-programowanie.pl/cpp/lista-jednokierunkowa-c.