Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 11. Zastosowania wskaźników funkcyjnych

Zagadnienia do opracowania:

- wskaźniki funkcyjne i ich zastosowania
- wywołania zwrotne
- leniwa inicjalizacja zmiennych
- funkcje o nieokreślonej liczbie argumentów

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie ze wskaźnikami funkcyjnymi w językach *C/C++* oraz możliwością ich zastosowania w praktyce.

2. Wprowadzenie

2.1. Wskaźniki funkcyjne

Funkcje, tak samo jak zmienne czy stałe, posiadają swój unikalny adres w pamięci komputera. Położone są w segmencie ***text (code)***. W związku z tym, możliwe jest przypisanie adresu funkcji do zmiennej wskaźnikowej. Taki wskaźnik nazywa się wówczas ***wskaźnikiem funkcyjnym***. Składnia ***wskaźników funkcyjnych*** jest następująca:

typ_zwracany (*nazwa_wskaźnika)(lista_argumentów)

Wspomniany w Ćw. 9. *adres powrotu z funkcji* jest również wskaźnikiem, co prawda nie funkcyjnym, ale wskazuje na pewien fragment obszaru pamięci ***text***. Przykład deklaracji i inicjalizacji ***wskaźnika funkcyjnego*** przedstawiono na listingu 1. ***Wskaźnik funkcyjny funcPtr*** może przechowywać adres funkcji zwracającej typ *float* i przyjmującej dwa argumenty typu *float*. Taką funkcją jest ***multiply()***. Przypisanie adresu funkcji do ***wskaźnika funkcyjnego*** przebiega analogicznie, jak do klasycznego wskaźnika (***funcPtr = multiply***), z tą różnicą, że do pobrania adresu funkcji nie korzysta się z ***operatora adresu*** &. **Nazwa funkcji stanowi wskaźnik na nią samą**, podobnie jak miało to miejsce w przypadku tablic. Warto zauważyć, że aby zdeklarować ***wskaźnik funkcyjny*** do funkcji o określonej sygnaturze, wystarczy zmienić nazwę funkcji (tu: *multiply*) na wyrażenie ***(*nazwa_wskaźnika)*** (tu: *(*funcPtr)*). Rozmiar ***wskaźnika funkcyjnego*** jest taki sam jak rozmiar dowolnego innego typu wskaźnikowego, czyli równy długości adresu w bajtach (zależy od architektury komputera).

```c
#include <stdio.h>

float multiply(float x, float y) {
    return x * y;
}

int main() {
    // Deklaracja lokalnego wskaznika funkcyjnego
    funcPtr
    float (*funcPtr)(float, float);
    // Inicjalizacja wskaznika funkcyjnego
    funcPtr = multiply;
    // Wywolanie funkcji przez wskaznik funkcyjny
    printf("Multiplication result: %f", funcPtr(2.0,
    3.5));
    return 0;
}
```

Listing 1. Deklaracja i inicjalizacja wskaźnika funkcyjnego

Składnia wywołania funkcji, której adres przechowuje ***wskaźnik funkcyjny***, jest identyczna jak składnia bezpośredniego wywołania funkcji:

```c
// Wywolanie funkcji multiply()
multiply(2.0, 3.5);
// Wywolanie funkcji multiply() przez wskaznik
   funcPtr
float (*funcPtr)(float, float) = multiply;
funcPtr(2.0, 3.5);
// Ponowna inicjalizacja wskaznika funcPtr funkcja
   add()
funcPtr = add;
```

```
// Wywolanie funkcji add() przez wskaznik funcPtr
funcPtr(1.3, -0.9);
```

Wynika stąd wniosek, że nie jest możliwe rozróżnienie ***wskaźnika funkcyjnego*** od funkcji na podstawie samej instrukcji wywołania. Możliwe jest to dopiero po sprawdzeniu odpowiadającej deklaracji. Przykłady deklaracji różnych wskaźników i funkcji wykorzystujących wskaźniki przedstawiono na listingu 2.

```
// Wskaznik na typ calkowity
int * ptr;
// Tablica liczb calkowitych
int tab[];
// Tablica wskaznikow na typ calkowity
int * tab[];
// Wskaznik na wskaznik na typ calkowity
int ** ptr;
// Funkcja pobierajaca wskaznik na typ calkowity i
   niezwracajaca nic
void func(int *);
// Wskaznik na funkcje pobierajaca wskaznik na typ
   calkowity i niezwracajaca nic
void (*ptr)(int *);
// Funkcja pobierajaca liczbe calkowita i zwracajaca
    wskaznik na typ zmiennoprzecinkowy
double * func(int);
// Wskaznik na funkcje pobierajaca liczbe calkowita
   i zwracajaca wskaznik na typ zmiennoprzecinkowy
double * (*ptr)(int);
// Funkcja niepobierajaca nic i zwracajaca wskaznik
   na funkcje pobierajaca liczbe calkowita i
   zwracajaca wskaznik na typ zmiennoprzecinkowy
```

```
18 double * (*func())(int)
19 // Tablica wskaznikow na funkcje pobierajacych
     liczbe calkowita i zwracajacych wskaznik na typ
     zmiennoprzecinkowy
20 double * (*ptr[])(int);
```

Listing 2. Przykłady deklaracji wskaźników i funkcji operujących na wskaźnikach

2.2. Wywołania zwrotne

Wskaźniki funkcyjne znajdują zastosowanie jako argumenty funkcji. Przykładem mogą być funkcje implementujące algorytmy sortowania, jak np. *bubbleSort()*. Algorytm sortowania bąbelkowego z poprzedniego ćwiczenia sortuje elementy tablicy w kolejności rosnącej. Aby zmienić porządek sortowania wystarczy zmienić instrukcję porównującą dwa sąsiadujące elementy tablicy ($tab[j] > tab[j+1]$ na $tab[j] < tab[j+1]$). Powielanie całego kodu funkcji w celu zmiany pojedynczego operatora porównania jest złym rozwiązaniem (większa ilość kodu do utrzymania, konieczność modyfikacji tej samej funkcjonalności w dwóch miejscach, większy rozmiar pliku wykonywalnego). Rozwiązaniem może być przekazanie wskaźnika na funkcję realizującą operację porównania elementów tablicy, jako argument funkcji sortującej (listing 3.).

```
1 #include <iostream>
2
3 bool isLess(int x, int y) {
4     return x < y;
5 }
6
7 bool isGreater(int x, int y) {
8     return x > y;
```

```
9  }
10
11 void swap(int * x, int * y) {
12     int tmp = *x;
13     *x = *y;
14     *y = tmp;
15 }
16
17 void printTab(const int * tab, unsigned int tabSize)
       {
18     if (tab != NULL) {
19         for (unsigned int i = 0; i < tabSize; ++i)
20             std::cout << tab[i];
21         std::cout << std::endl;
22     }
23 }
24
25 void bubbleSort(int * tab, unsigned int n, bool (*
       comparator)(int, int))
26 {
27     if (tab != NULL && comparator != NULL)
28         for (unsigned int i = 0; i < n - 1; ++i)
29             for (unsigned int j = 0; j < n - i - 1;
       ++j)
30                 // Wywolanie funkcji porownujacej
31                 if (comparator(tab[j], tab[j + 1]))
       {
32                     swap(tab + j, tab + j + 1);
33                 }
34 }
35
```

```
36 int main() {
37     const unsigned int tabSize = 10;
38     int tab[tabSize] = {4, -7, 1, 0, 2, 1, 8, -3,
   -2, 5};
39     // Sortowanie w kolejnosci malejacej
40     bubbleSort(tab, tabSize, isLess);
41     printTab(tab, tabSize);
42     // Sortowanie w kolejnosci rosnacej
43     bubbleSort(tab, tabSize, isGreater);
44     printTab(tab, tabSize);
45     return 0;
46 }
```

Listing 3. Zastosowanie wskaźnika funkcyjnego w algorytmie sortowania bąbelkowego

Wywołanie ***wskaźnika funkcyjnego*** przekazanego jako argument funkcji (jak w przypadku wskaźnika ***comparator*** z listingu 3.) nazywa się ***wywołaniem zwrotnym*** (*ang. callback*). Warto zwrócić uwagę na różnicę w przekazaniu do funkcji adresu innej funkcji, a przekazaniu rezultatu działania innej funkcji:

```
1 // Przekazanie do funkcji bubbleSort() adresu
     funkcji isLess()
2 bubbleSort(tab, tabSize, isLess);
3 // Przekazanie do funkcji bubbleSort() wyniku
     dzialania funkcji isLess() (zmiennej typu bool)
4 bubbleSort(tab, tabSize, isLess());
```

Mechanizm ***wywołań zwrotnych*** jest bardzo popularny w językach ***wysokiego poziomu***; w szczególności znalazł zastosowanie w **programowaniu asynchronicznym**. Jednakże zagadnienie to znacznie wykracza poza zakres kursu.

2.3. Wskaźnik funkcyjny jako pole struktury

Wskaźnik funkcyjny może być również polem struktury. Otrzymuje się wówczas namiastkę ***klas*** w języku *C* – ***wskaźniki funkcyjne*** pełnią rolę analogiczną do ***metod*** znanych z języka *C++*. Jako przykład takiego zastosowania ***wskaźników funkcyjnych*** omówiona zostanie obsługa poleceń tekstowych, popularna w przypadku mikrokontrolerów. Do wyprowadzeń **GPIO** (*ang. General Purpose Input Output*) mikrokontrolera można podłączyć np. układ diod elektroluminescencyjnych (LED). Świecenie lub wygaszanie konkretnej diody może służyć wizualnej sygnalizacji użytkownikowi gotowości mikrokontrolera do odbioru danych za pomocą protokołu bezprzewodowego (jak *Bluetooth* czy *ZigBee*) lub zakończenia transmisji. Na listingu 4. przedstawiono przykłady deklaracji funkcji realizujących opisane operacje. Funkcje ***void turnOn(unsigned int gpio)*** i ***void turnOff(unsigned int gpio)*** wystawiają odpowiednio stan wysoki lub niski na wyprowadzeniu **GPIO** mikrokontrolera o zadanym numerze (powodując świecenie lub wygaszenie podłączonej do niego diody). Funkcja ***void sleep(unsigned int seconds)*** powoduje przejście mikrokontrolera w stan uśpienia przez określony czas (w sekundach), np. w celu oszczędzania energii w stanie bezczynności. Warto zwrócić uwagę, że wymienione funkcje mają zgodne sygnatury.

```
1 void turnOn(unsigned int gpio);
2 void turnOff(unsigned int gpio);
3 void sleep(unsigned int seconds);
```

Listing 4. Deklaracje funkcji realizujących założone operacje przez mikrokontroler

Użytkownik za pomocą interfejsu szeregowego (np. *UART*, *SPI* czy *I2C*) przesyła do mikrokontrolera polecenia tekstowe, które odpowiadają funkcjom z listingu 4. Pojedyncze polecenie składa się z instrukcji oraz jej argumentu, rozdzielonych spacją, np.:

- ”turnOn 5” – zaświeć diodę podpiętą do wyprowadzenia nr 5;
- ”turnOff 2” – zgaś diodę podpiętą do wyprowadzenia nr 2;
- ”sleep 60” – przejdź w stan uśpienia na 60 sekund.

Odbiór przesyłanych danych realizuje funkcja ***receive()***:

```
// Odbior danych przeslanych za pomoca interfejsu szeregowego
const char * receive();
```

Zadaniem mikrokontrolera jest *parsowanie* poleceń i wykonywanie odpowiadających im operacji. Pierwszym etapem realizacji zadania jest implementacja funkcji *parsującej* polecenia – rozdzielającej instrukcję ***instruction*** (”turnOn”, ”turnOff”, ”sleep”) od argumentu ***argument*** (5, 2, 60, ...). Sparsowane polecenia można przechować w strukturze ***ParsedCommand***. Na potrzeby przykładu ograniczono długość instrukcji do 7 znaków (+1 dla znaku końca łańcucha ’\0’):

```
#define MAX_INSTRUCTION_LENGTH 8

typedef struct ParsedCommand {
    char instruction[MAX_INSTRUCTION_LENGTH];
    unsigned int argument;
} ParsedCommand_t;
```

Wówczas funkcja parsująca przyjmie postać, jak na listingu 5. Rozdzielenie łańcucha znakowego, zawierającego polecenie, na instrukcję oraz argument, odbywa się z wykorzystaniem funkcji ***strtok()***, zdeklarowanej w nagłówku ***string.h*** biblioteki standardowej. Pobiera ona łańcuch znakowy, który ma zostać podzielony oraz, jako drugi argument, łańcuch zawierający separatory (*ang. delimiters*) (tu: spację). Ponieważ funkcja operuje na oryginale

łańcucha i modyfikuje go (*char **), przed przekazaniem polecenia do funkcji ***strtok()*** wykonywana jest lokalna kopia łańcucha do bufora ***buffer***. Funkcja ***strtok()*** zwraca wartość ***NULL*** w przypadku nieodnalezienia żadnego z zadanych separatorów w przekazanym łańcuchu znakowym. W przeciwnym razie zwracany jest ciąg znaków poprzedzających pierwszy napotkany separator. Aby otrzymać kolejne fragmenty łańcucha (występujące pomiędzy kolejnymi separatorami) należy ponownie wywoływać funkcję ***strtok()*** z pierwszym argumentem ustawionym na wartość ***NULL***. Oba uzyskane w ten sposób fragmenty kopiowane są do pól struktury ***ParsedCommand***. W przypadku argumentu instrukcji przeprowadzana jest konwersja z literału łańcuchowego do liczby całkowitej, z wykorzystaniem funkcji ***atoi()***, zdeklarowanej w nagłówku ***stdlib.h*** biblioteki standardowej.

```
1  ParsedCommand_t parseCommand(const char * command) {
2      ParsedCommand_t parsedCommand;
3      // Wykonaj lokalna kopie instrukcji
4      unsigned int commandLength = strlen(command) +
       1;
5      char * buffer = (char *) calloc(commandLength,
       sizeof(char));
6      strncpy(buffer, command, commandLength);
7      // Pobierz fragment lancucha znakowego do
       wystapienia spacji
8      const char * delimiter = " ";
9      char * part = strtok(buffer, delimiter);
10     // Jezeli znaleziono spacje
11     if (part != NULL) {
12         // Skopiuj instrukcje do struktury
13         strncpy(parsedCommand.instruction, part,
       sizeof(parsedCommand.instruction));
14         // Pobierz drugi fragment polecenia
```

```
15          part = strtok(NULL, delimiter);
16          // Dodaj arugment do struktury
17          parsedCommand.argument = atoi(part);
18      }
19      free(buffer);
20      return parsedCommand;
21  }
```

Listing 5. Funkcja parsująca polecenie

Kolejnym etapem jest implementacja funkcji ***dispatch()***, której zadaniem jest wywołanie odpowiedniej funkcji mikrokontrolera. W pierwszym kroku funkcja mapuje instrukcje na odpowiadające im ***wskaźniki funkcyjne***. W tym celu można wprowadzić pomocniczą strukturę ***Task***. ***Wskaźnik funkcyjny executor*** posiada typ zgodny z nagłówkami funkcji ***turnOn()***, ***turnOff()*** i ***sleep()***:

```
1  typedef struct Task {
2      char instruction[MAX_INSTRUCTION_LENGTH];
3      void (*executor)(unsigned int);
4  } Task_t;
```

Implementacja funkcji ***dispatch()*** została przedstawiona na listingu 6. Na początku tworzona jest tablica ***taskList***, która inicjalizowana jest mapowaniami instrukcji na ***wskaźniki funkcyjne***. Słowo kluczowe ***static*** powoduje, że tablica ***taskList*** będzie widoczna między kolejnymi wywołaniami funkcji, i co ważniejsze, zostanie utworzona i zainicjalizowana tylko raz (optymalizacja czasu wykonania programu), podczas pierwszego wywołania funkcji ***dispatch()***. Jest to tak zwana ***leniwa inicjalizacja zmiennej*** (*ang. lazy initialization*). Następnie, w pętli, porównywana jest instrukcja przekazana jako argument funkcji z instrukcją przechowywaną na danej pozycji tablicy. Jeżeli łańcuchy znakowe są identyczne, wywoływana jest odpowiednia funkcja realizująca jedno z zadań mikrokontrolera. Instrukcja ***break*** przerywa

pętlę, ponieważ każda instrukcja występuje w tablicy tylko raz – nie ma sensu dalsze iterowanie po elementach tablicy ***taskList***.

```
void dispatch(ParsedCommand_t command) {
    // Jednorazowe mapowanie instrukcji na wskazniki funkcyjne
    static const Task_t taskList[] = {
        (Task_t){"turnOn", turnOn},
        (Task_t){"turnOff", turnOff},
        (Task_t){"sleep", sleep}};

    // Petla po wszystkich elementach tablicy
    for (unsigned int i = 0; i < sizeof(taskList) / sizeof(Task_t); ++i) {
        // Jezeli instrukcja z polecenia jest taka sama jak instrukcja w tablicy
        if (strncmp(taskList[i].instruction, command.instruction, MAX_INSTRUCTION_LENGTH) == 0) {
            // Wywolaj zmapowana funkcje
            taskList[i].executor(command.argument);
            // Kazda instrukcja wystepuje tylko raz
            break;
        }
    }
}
```

Listing 6. Funkcja mapująca instrukcje na wskaźniki funkcyjne

Ostatecznie obsługa poleceń tekstowych przez mikrokontroler sprowadza się do odpowiedniego złożenia funkcji ***receive()***, ***parseCommand()*** oraz ***dispatch()***:

```
1     dispatch(parseCommand(receive()));
```

2.4. Funkcje o nieokreślonej liczbie argumentów

Funkcje o nieokreślonej liczbie argumentów (*ang. variadic functions*) to funkcje, których ***zmienna lista argumentów*** (*ang. variadic arguments*) może być reprezentowana zarówno przez zero, jeden czy więcej argumentów, z którymi wywołana będzie funkcja. Przykładem takich funkcji są *printf()* i *scanf()*, których nagłówki przedstawiono na listingu 7.

```
1 int printf(const char * format, ...);
2 int scanf(const char * format, ...);
```

Listing 7. Nagłówki funkcji wejścia/wyjścia w języku *C*

Dostęp do ***zmiennej listy argumentów*** jest realizowany z wykorzystaniem czterech makr, zdeklarowanych w nagłówku ***stdarg.h*** biblioteki standardowej:

- ***va_start*** – umożliwiającej dostęp do ***zmiennej listy argumentów***;
- ***va_arg*** – pobierającej następny argument ze ***zmiennej listy argumentów***;
- ***va_copy*** [*od standardu C99*] – wykonującej kopię ***zmiennej listy argumentów***;
- ***va_end*** – kończącej iterowanie po ***zmiennej liście argumentów***.

Przykład funkcji ***average()*** liczącej średnią arytmetyczną dla dowolnej liczby argumentów przedstawiono na listingu 8. Praca ze ***zmienną listą argumentów*** rozpoczyna się od wywołania makra ***va_start***. Inicjalizuje ono zmienną typu ***va_list***. Jest to typ danych przechowujący informacje potrzebne do operowania na ***zmiennej liście argumentów*** przez makra ***va_arg***

oraz ***va_end***. Jako drugi argument makro ***va_start*** przyjmuje (nazwany) argument funkcji **bezpośrednio poprzedzający *zmienną listę argumentów*** (tu: ***count***). Kopie argumentów wywołania funkcji przechowywane są na ***stosie***. Makro ***va_start*** operuje na wskaźnikach, aby określić adres początku ***zmiennej listy argumentów***, stąd wymagane jest przekazanie zmiennej **bezpośrednio poprzedzającej *zmienną listę argumentów***. Kolejne wartości ze ***zmiennej listy argumentów*** pobierane są za pomocą makra ***va_arg*** przyjmującego dwa argumenty: zmienną typu ***va_list***, zainicjalizowaną wcześniej przez wywołanie ***va_start***, oraz ***deskryptor typu*** (tu: *double*). Jeżeli typ zmiennej na ***zmiennej liście argumentów*** jest niezgodny z przekazanym deskryptorem, to działanie makra jest **niezdefiniowane** (w przypadku funkcji ***average()*** oczekuje się, że wszystkie argumenty następujące po ***count*** będą typu *double*). Deskryptor typu jest wymagany do przeprowadzenia prawidłowej arytmetyki wskaźników na ***zmiennej liście argumentów***. Należy mieć na uwadze, że **nie ma możliwości dedukcji liczby argumentów przekazanych do funkcji na podstawie samej *zmiennej listy argumentów***. Wywołanie makra ***va_arg*** większą liczbę razy niż wynosi rozmiar ***zmiennej listy argumentów*** skutkuje **niezdefiniowanym zachowaniem**. Klasycznym rozwiązaniem jest przekazanie rozmiaru ***zmiennej listy argumentów*** jako pierwszy argument funkcji (jak w przypadku ***average()***). Rozmiar ***zmiennej listy argumentów printf()*** i ***scanf()*** określany jest na podstawie liczby ***specyfikatorów formatu %*** występujących w ***ciągu formatującym*** (patrz: Ćw. 4). Po zakończeniu pracy ze ***zmienną listą argumentów*** należy wywołać makro ***va_end*** zwalniające zmienną typu ***va_list***, zainicjalizowaną wcześniej przez wywołanie makra ***va_start*** (lub ***va_copy***). Aby iterować po ***zmiennej liście argumentów*** po wywołaniu makra **va_end** należy ponownie zaincjalizować zmienną ***args*** przez wywołanie ***va_start***.

```c
#include <stdarg.h>
#include <stdio.h>

double average(unsigned int count, ...) {
  // Typ danych uzywany przez makra va_start, va_end, va_arg
  va_list args;
  // Przekazanie argumentu count poprzedzajacego zmienna liste argumentow
  va_start(args, count);
  // Inicjalizacja zmiennej lokalnej do zera
  double sum = 0;
  for (unsigned int i = 0; i < count; ++i)
    // Pobranie kolejnej wartosci ze zmiennej listy argumentow (oczekiwany typ zmiennych to double)
    sum += va_arg(args, double);
  // Zakonczono iterowanie po zmiennej liscie argumentow
  va_end(args);
  return sum / count;
}

int main() {
  const unsigned int count = 5;
  printf("%f", average(count, 1.3, 2.5, 0.7, 3.4, -1.6));
  return 0;
}
```

Listing 8. Funkcja licząca średnią arytmetyczną z wykorzystaniem zmiennej listy argumentów

Możliwe jest również tworzenie ***wskaźników funkcyjnych*** na ***funkcje o nieokreślonej liczbie argumentów***:

```
const unsigned int count = 3;
double (*funcPtr)(unsigned int, ...) = average;
printf("%f", funcPtr(count, -0.4, 1.2, 4.5));
```

3. Program ćwiczenia

Plik nagłówkowy ***list.h*** zawiera deklaracje czterech funkcji implementujących listę:

- ***Node_t * createList(unsigned int nodeCount, ...)*** – tworzącej nową listę o liczbie ***nodeCount*** węzłów. Wartości (*head*) kolejnych węzłów przekazywane są za pomocą zmiennej listy argumentów (typu *int*);
- ***void printList(Node_t * root)*** – wypisującej kolejne wartości listy rozpoczynającej się węzłem o adresie ***root***;
- ***void push(Node_t * root, int value)*** – dodającej nowy węzeł o wartości ***value*** na koniec listy rozpoczynającej się węzłem o adresie ***root***;
- ***void removeIf(Node_t ** root, Predicate predicate, int toCompare)*** – usuwającej pierwszy węzeł listy spełniający warunek ***predicate***. ***Predicate*** to alias na typ wskaźnika funkcyjnego ***bool (*)(int, int)***. Funkcja usuwa pierwszy węzeł, dla którego ***predicate*** zwróci wartość ***true***, porównując wartość węzła (*head*) z wartością ***toCompare***.

Plik nagłówkowy ***comparators.h*** zawiera deklaracje trzech funkcji zgodnych z typem wskaźnika funkcyjnego ***Predicate***:

- ***bool isEqual(int lhs, int rhs)*** – sprawdzającej czy wartości ***lhs*** i ***rhs*** są równe;

- ***bool isGreater(int lhs, int rhs)*** – sprawdzającej czy wartość ***lhs*** jest większa niż ***rhs***;
- ***bool isLess(int lhs, int rhs)*** – sprawdzającej czy wartość ***lhs*** jest mniejsza niż ***rhs***.

W pliku źródłowym ***list.c*** zdefiniowano funkcje ***push()*** i ***printList()*** oraz pomocniczą (statyczną) funkcję *createNode()*.

Zadanie 1. Celem zadania jest implementacja funkcji ***createList()***. Do implementacji funkcji ***createList()*** należy wykorzystać wcześniej zdefiniowane funkcje ***createNode()*** i ***push()***. Definicję funkcji należy zamieścić w pliku ***list.c***. Plik ***main.c*** zawiera przykład użycia zaimplementowanej listy, testujący działanie funkcji ***createList()***. Cały program powinien zostać napisany w języku *C*.

Zadanie 2. Celem zadania jest rozszerzenie kodu z **Zadania 1.** o implementację funkcji ***removeIf()*** (w pliku źródłowym ***list.c***), a także ***isEqual()***, ***isGreater()*** i ***isLess()*** (w pliku źródłowym ***comparators.c***). Plik ***main.c*** zawiera przykład użycia zaimplementowanej listy, testujący działanie funkcji ***removeIf()***. Cały program powinien zostać napisany w języku *C*.

Zadanie 3. W plikach ***parser.h*** i ***parser.c*** zawarto implementację parsera poleceń z listingu 5. Celem zadania jest rozszerzenie kodu z **Zadania 2.** o przekazywanie warunków ***Predicate*** do funkcji ***removeIf()*** za pomocą poleceń tekstowych (argumentów funkcji *main()*). Plik ***main.c*** został w tym celu odpowiednio zmodyfikowany. Plik nagłówkowy ***dispatcher.h*** zawiera deklarację funkcji ***void dispatch(Node_t ** root, ParsedCommand_t parsedCommand)***, której zadaniem jest zmapowanie instrukcji na odpowiadające im funkcje porównujące (***isEqual()***, ***isGreater()***, ***isLess()***), a następnie wywołanie funkcji ***removeIf()*** na przekazanej do funkcji liście (***root***) z określonymi w strukturze ***parsedCommand*** argumentami.

Implementację funkcji ***dispatch()*** należy umieścić w pliku źródłowym ***dispatcher.c***. Przykładowe wywołanie skompilowanej aplikacji może wyglądać następująco: **app.exe "isEqual 2" "isLess 10" "isGreater 4"**

4. Dodatek

4.1. Obsługa sygnałów systemowych

Sygnały (*ang. signals*) to jeden z mechanizmów ***komunikacji międzyprocesowej*** (*inter-process communication, IPC*). ***Procesem*** nazywa się pojedynczą instancję wykonywanej aplikacji w systemie operacyjnym. System operacyjny przydziela nowo utworzonemu ***procesowi*** jego pamięć (i wirtualną przestrzeń adresową), pliki (np. *standardowe wejście/wyjście*), czas procesora oraz indywidualny identyfikator ***PID*** (*ang. process identifier*). Sam ***proces*** może zawierać jeden lub więcej ***wątków*** (*ang. threads*), czyli fragmentów programu wykonywanych współbieżnie (równolegle). ***Wątki*** współdzielą przestrzeń adresową procesu (ale każdy ***wątek*** posiada oddzielną pamięć ***stosu***). ***Komunikacja międzyprocesowa*** polega na wymianie danych między ***procesami*** systemu operacyjnego, tj. między różnymi instancjami aplikacji komputerowych. Wśród technik ***IPC*** można wyróżnić m.in.:

- gniazda (*ang. sockets*);
- pliki (*ang. files*);
- kolejki komunikatów (*ang. message queues*);
- potoki (*ang. pipes*);
- pamięć współdzieloną (*ang. shared memory*);
- ***sygnały*** (*ang. signals*).

Sygnały stanową **asynchroniczną** metodę komunikacji zdefiniowaną w standardzie **POSIX** (*ang. Portable Operating System Interface*) i są powszechnie wykorzystywane w rodzinie systemów operacyjnych ***Unix***. Znajdują również zastosowanie w ***systemach wbudowanych*** (*ang. embedded systems*) ze względu na relatywnie małą złożoność obliczeniową i pamięciową obsługi

sygnałów. W momencie przesłania ***sygnału*** do ***procesu*** system operacyjny przerywa normalne wykonanie programu, wymuszając obsługę ***sygnału***. ***Proces*** przechodzi do podprogramu obsługi ***sygnału*** (domyślnego lub wcześniej zarejestrowanego dla konkretnego ***sygnału***). W tabeli 1. zebrano najczęściej wykorzystywane ***sygnały*** standardu **POSIX**.

Tabela 1. Najpopularniejsze sygnały standardu POSIX [1]

Sygnał	**Wartość**	**Komentarz**
SIGINT	2	Przerwanie (z klawiatury)
SIGILL	4	Błędna instrukcja
SIGABRT	6	Przerwanie działania [1]
SIGKILL	9	Natychmiastowe przerwanie działania [2]
SIGSEGV	11	Błędne odwołanie do pamięci [3]
SIGTERM	15	Zakończenie działania [4]

Aby zaimplementować własny ***podprogram obsługi sygnału*** (*ang. signal handler*) należy skorzystać z funkcji ***signal()***, zdeklarowanej w nagłówku ***signal.h*** (*csignal*). Deklarację funkcji przedstawiono na listingu 9. Funkcja ***signal()*** przyjmuje dwa argumenty: numer sygnału ***sig*** (patrz: tabela 1.) oraz ***wskaźnik funkcyjny*** na ***podprogram obsługi sygnału*** – ***handler***. W przypadku powodzenia funkcja zwraca wskaźnik na poprzednio przypisany ***podprogram obsługi sygnału*** (każdy ***sygnał*** posiada swój domyślny mechanizm obsługi) albo wartość ***SIG_ERR*** w przeciwnym wypadku.

[1] wysyłane przez proces po wywołaniu funkcji *abort()* z bilbioteki standardowej

[2] nie może być zignorowany przez proces; nie można zarejestrować własnej procedury obsługi; nie jest przeprowadzane sprzątanie zasobów

[3] *segmentation violation*

[4] można zarejestrować własną procedurę obsługi; może być zignorowany przez proces

```c
void (*signal(int sig, void (*handler)(int)))(int);
```

Listing 9. Nagłówek funkcji *signal()*

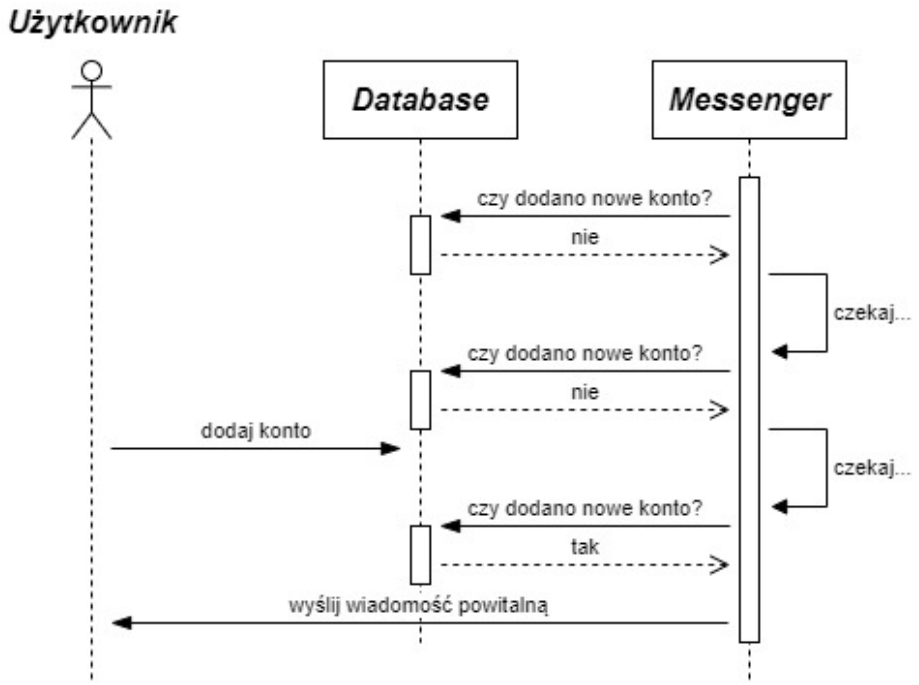
Przykładową obsługę sygnału przedstawiono na listingu 10. Wysłanie ***sygnału*** o określonej wartości można przeprowadzić korzystając z funkcji ***raise()***. ***Sygnał SIGINT*** można wysłać do ***procesu*** również z klawiatury (kombinacja *Ctrl+C*). Obsługa tego ***sygnału*** może być przydatna, np. jeżeli chcemy wybudzić konkretny ***wątek*** programu.

```c
#include <signal.h>
#include <stdio.h>

void setUpHandler(int signalCode, void (*handler)(
   int)) {
  printf("Setting up handler for signal code %d\n",
   signalCode);
  if (signal(signalCode, handler) ==  SIG_ERR)
    printf("Failed to set up handler\n");
}

void handle(int signalCode) {
  printf("Received signal code %d\n", signalCode);
}

int main() {
  setUpHandler(SIGINT, handle);
  raise(SIGINT);
  return 0;
}
```

Listing 10. Obsługa sygnałów systemowych

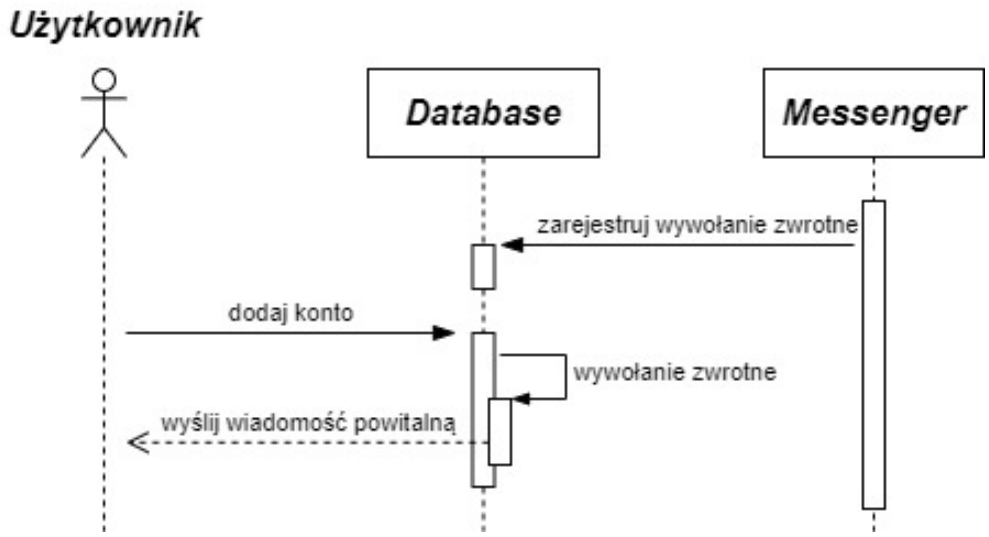
4.2. Wzorzec projektowy obserwatora

Inne, popularne zastosowanie ***wywołań zwrotnych*** może zostać zobrazowane następującym przykładem. Sklep internetowy wysyła wiadomość powitalną każdemu nowemu użytkownikowi. Komponent ***Messenger*** realizuje logikę odpowiedzialną za wysyłanie wiadomości e–mail, kiedy nowe konto użytkownika zostanie dodane do bazy danych (komponent ***Database***) przez ***użytkownika***. Często początkujący programiści implementują opisany system stosując cykliczne ***odpytywanie*** (*ang. polling*) bazy danych o to czy zostało dodane nowe konto (rys. 4.1). Pojawia się problem z doborem częstotliwości odpytywania. Jeżeli komponent ***Messenger*** będzie odpytywać bazę danych bardzo często (np. co sekundę), to użytkownik nie zauważy opóźnienia między założeniem konta, a otrzymaniem wiadomości powitalnej. Jednakże, wysyłane co sekundę zapytanie będzie musiało być każdorazowo obsłużone przez bazę danych, powodując opóźnienie w obsłudze zakolejkowanych zapytań od innych komponentów – powstaje *wąskie gardło* (*ang. bottleneck*) systemu. Jeżeli wiadomość będzie wysyłana z mniejszą częstotliwością (np. co godzinę), baza danych nie będzie krytycznie obciążona ilością przetwarzanych zapytań, ale użytkownik zauważy znaczne opóźnienie w otrzymaniu wiadomości powitalnej.



Rys. 4.1. Odpytywanie (*polling*)

Lepszym rozwiązaniem byłoby zarejestrowanie ***wywołania zwrotnego*** w komponencie ***Database***, tzn. przekazanie przez komponent ***Messenger*** bazie danych wskaźnika na funkcję realizującą wysłanie wiadomości powitalnej. Gdy użytkownik założy nowe konto, ***Database*** wykonuje wywołanie zwrotne, co skutkuje przesłaniem wiadomości e–mail. Sam komponent ***Messenger*** nie musi otrzymać informacji o tym, że zostało założone nowe konto. Takie rozwiązanie w inżynierii oprogramowania nosi nazwę ***wzorca projektowego obserwatora*** (***Messenger*** jest obserwatorem ***Database***).



Rys. 4.2. Wzorzec projektowy obserwatora

Literatura

[1] *SIGNAL(7). Linux Programmer's Manual.* URL: `http://man7.org/linux/man-pages/man7/signal.7.html`.