



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

<b>1</b>	<b>Cel ćwiczenia</b>	<b>2</b>
<b>2</b>	<b>Wprowadzenie</b>	<b>2</b>
2.1	Wskaźniki funkcyjne . . . . .	2
2.2	Wywołania zwrotne . . . . .	5
2.3	Wskaźnik funkcyjny jako pole struktury . . . . .	8
2.4	Funkcje o nieokreślonej liczbie argumentów . . . . .	13
<b>3</b>	<b>Program ćwiczenia</b>	<b>16</b>
<b>4</b>	<b>Dodatek</b>	<b>19</b>
4.1	Obsługa sygnałów systemowych . . . . .	19
4.2	Wzorzec projektowy obserwatora . . . . .	22

---

## 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 zadeklarować *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).

---

```
1 #include <stdio.h>
2
3 float multiply(float x, float y) {
4     return x * y;
5 }
6
7 int main() {
8     // Deklaracja lokalnego wskaźnika funkcyjnego
9     funcPtr
10    float (*funcPtr)(float, float);
11    // Inicjalizacja wskaźnika funkcyjnego
12    funcPtr = multiply;
13    // Wywołanie funkcji przez wskaźnik funkcyjny
14    printf("Multiplication result: %f", funcPtr(2.0,
15    3.5));
16    return 0;
17 }
```

Listing 1. Deklaracja i inicjalizacja wskaźnika funkcyjnego

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

```
1 // Wywołanie funkcji multiply()
2 multiply(2.0, 3.5);
3 // Wywołanie funkcji multiply() przez wskaźnik
4 funcPtr
5 float (*funcPtr)(float, float) = multiply;
6 funcPtr(2.0, 3.5);
7 // Ponowna inicjalizacja wskaźnika funcPtr funkcja
8 add()
9 funcPtr = add;
```

```
8 // Wywołanie funkcji add() przez wskaznik funcPtr
9 funcPtr(1.3, -0.9);
```

Wynika stąd wniosek, że nie jest możliwe rozróżnienie *wskaznika 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.

```
1 // Wskaznik na typ calkowity
2 int * ptr;
3 // Tablica liczb calkowitych
4 int tab[];
5 // Tablica wskaznikow na typ calkowity
6 int * tab[];
7 // Wskaznik na wskaznik na typ calkowity
8 int ** ptr;
9 // Funkcja pobierajaca wskaznik na typ calkowity i
   niezwracajaca nic
10 void func(int *);
11 // Wskaznik na funkcje pobierajaca wskaznik na typ
   calkowity i niezwracajaca nic
12 void (*ptr)(int *);
13 // Funkcja pobierajaca liczbe calkowita i zwracajaca
   wskaznik na typ zmiennoprzecinkowy
14 double * func(int);
15 // Wskaznik na funkcje pobierajaca liczbe calkowita
   i zwracajaca wskaznik na typ zmiennoprzecinkowy
16 double * (*ptr)(int);
17 // Funkcja niepobierajaca nic i zwracajaca wskaznik
   na funkcje pobierajaca liczbe calkowita i
   zwracajaca wskaznik na typ zmiennoprzecinkowy
```

```

18 double * (*func())(int)
19 // Tablica wskaźników na funkcje pobierających
    liczbe całkowita i zwracających wskaźnik 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 {
19     if (tab != NULL) {
20         for (unsigned int i = 0; i < tabSize; ++i)
21             std::cout << tab[i];
22         std::cout << std::endl;
23     }
24 }
25
26 void bubbleSort(int * tab, unsigned int n, bool (*
27     comparator)(int, int))
28 {
29     if (tab != NULL && comparator != NULL)
30         for (unsigned int i = 0; i < n - 1; ++i)
31             for (unsigned int j = 0; j < n - i - 1;
32                 ++j)
33                 // Wywołanie funkcji porównującej
34                 if (comparator(tab[j], tab[j + 1]))
35                     swap(tab + j, tab + j + 1);
36 }

```

```

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
   działania 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()*:

```
1 // Odbiór danych przesyłanych za pomocą interfejsu
   szeregowego
2 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'):

```
1 #define MAX_INSTRUCTION_LENGTH 8
2
3 typedef struct ParsedCommand {
4     char instruction[MAX_INSTRUCTION_LENGTH];
5     unsigned int argument;
6 } 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()*, zadeklarowanej 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, przekazywany 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()*, zadeklarowanej 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) +
5     1;
6     char buffer[commandLength];
7     strncpy(buffer, command, commandLength);
8     // Pobierz fragment lancucha znakowego do
9     wystapienia spacji
10    const char * delimiter = " ";
11    char * part = strtok(buffer, delimiter);
12    // Jezeli znaleziono spacje
13    if (part != NULL) {
14        // Skopiuj instrukcje do struktury
15        strncpy(parsedCommand.instruction, part,
16        sizeof(parsedCommand.instruction));
17        // Pobierz drugi fragment polecenia
18        part = strtok(NULL, delimiter);
```

```

16         // Dodaj arugment do struktury
17         parsedCommand.argument = atoi(part);
18     }
19     return parsedCommand;
20 }

```

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 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*.

---

```

1 void dispatch(ParsedCommand_t command) {
2     // Jednorazowe mapowanie instrukcji na wskaźniki
   funkcyjne
3     static const Task_t taskList[] = {
4         (Task_t){ "turnOn", turnOn },
5         (Task_t){ "turnOff", turnOff },
6         (Task_t){ "sleep", sleep } };
7
8     // Petla po wszystkich elementach tablicy
9     for (unsigned int i = 0; i < sizeof(taskList) /
   sizeof(Task_t); ++i) {
10        // Jezeli instrukcja z polecenia jest taka
   sama jak instrukcja w tablicy
11        if (strncmp(taskList[i].instruction, command
   .instruction, MAX_INSTRUCTION_LENGTH) == 0) {
12            // Wywolaj zmapowana funkcje
13            taskList[i].executor(command.argument);
14            // Kazda instrukcja wystepuje tylko raz
15            break;
16        }
17    }
18 }

```

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, zadeklarowanych 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 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*. Dodatkowo zmienna ta **nie może być tablicą**, inaczej działanie makra jest **niezdefiniowane**. 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 zainicjalizować zmienną *args* przez wywołanie *va\_start*.

```
1 #include <stdarg.h>
2 #include <stdio.h>
3
4 double average(unsigned int count, ...) {
5     // Typ danych uzywany przez makra va_start, va_end
6     , va_arg
7     va_list args;
8     // Przekazanie argumentu count poprzedzajacego
9     zmienna liste argumentow
10    va_start(args, count);
11    // Inicjalizacja zmiennej lokalnej do zera
12    double sum = 0;
13    for (unsigned int i = 0; i < count; ++i)
14        // Pobranie kolejnej wartosci ze zmiennej listy
15        argumentow (oczekiwany typ zmiennych to double)
16        sum += va_arg(args, double);
17    // Zakonczono iterowanie po zmiennej liscie
18    argumentow
19    va_end(args);
20    return sum / count;
21 }
22
23 int main() {
24     const unsigned int count = 5;
25     printf("%f", average(count, 1.3, 2.5, 0.7, 3.4,
26         -1.6));
27     return 0;
28 }
```

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



---

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

```
1 const unsigned int count = 3;
2 double (*funcPtr)(unsigned int, ...) = average;
3 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** stanowią **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 <sup>1</sup>
SIGKILL	9	Natychmiastowe przerwanie działania <sup>2</sup>
SIGSEGV	11	Błędne odwołanie do pamięci <sup>3</sup>
SIGTERM	15	Zakończenie działania <sup>4</sup>

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.

<sup>1</sup>wysyłane przez proces po wywołaniu funkcji *abort()* z biblioteki standardowej

<sup>2</sup>nie może być zignorowany przez proces; nie można zarejestrować własnej procedury obsługi; nie jest przeprowadzane sprzątanie zasobów

<sup>3</sup>*segmentation violation*

<sup>4</sup>można zarejestrować własną procedurę obsługi; może być zignorowany przez proces

```
1 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.

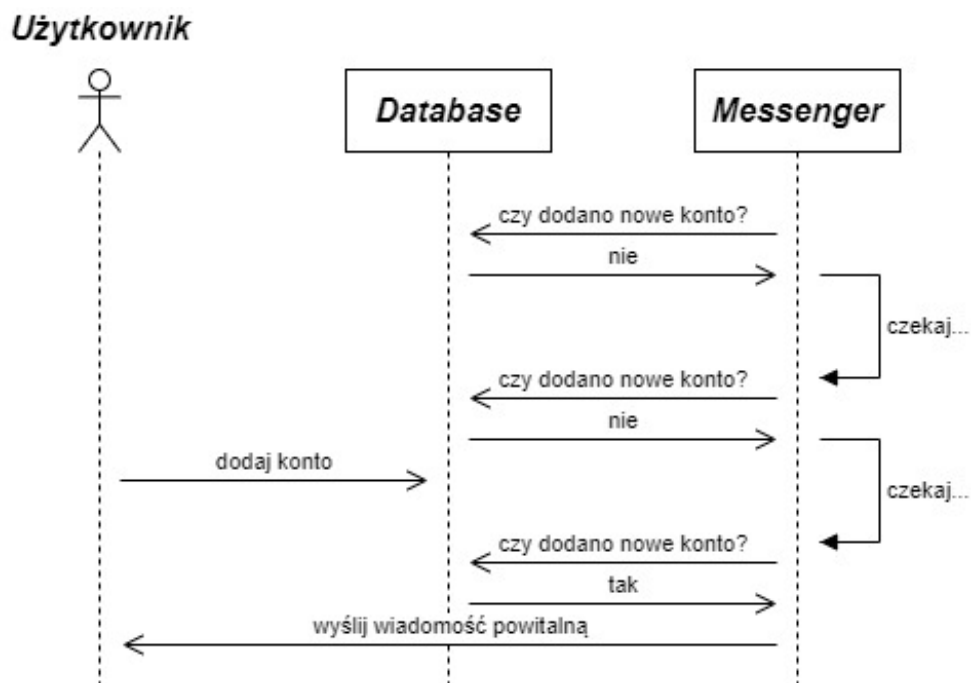
```
1 #include <signal.h>
2 #include <stdio.h>
3
4 void setUpHandler(int signalCode, void (*handler)(
5     int)) {
6     printf("Setting up handler for signal code %d\n",
7         signalCode);
8     if (signal(signalCode, handler) == SIG_ERR)
9         printf("Failed to set up handler\n");
10 }
11
12 void handle(int signalCode) {
13     printf("Received signal code %d\n", signalCode);
14 }
15
16 int main() {
17     setUpHandler(SIGINT, handle);
18     raise(SIGINT);
19     return 0;
20 }
```

Listing 10. Obsługa sygnałów systemowych

---

## 4.2. Wzorzec projektowy obserwatora

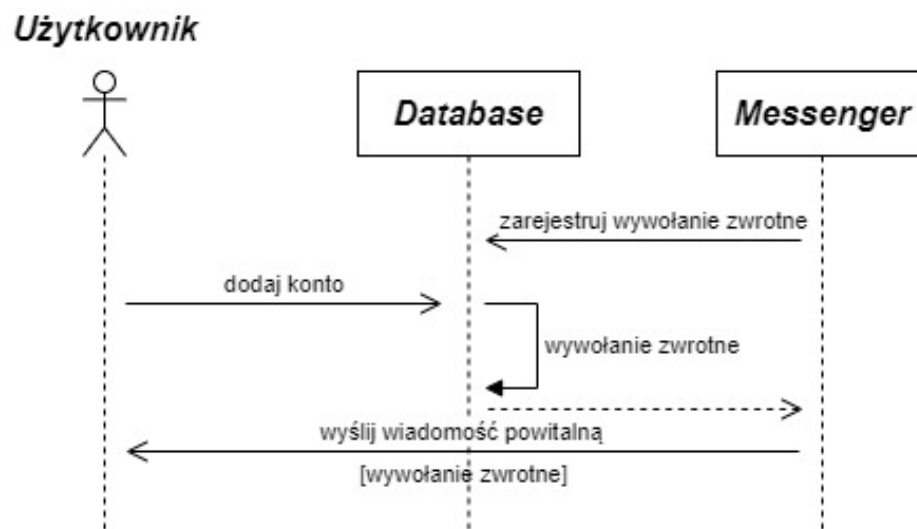
Inne, popularne zastosowanie *wywołań zwrotnych* może zostać zobrażowane następującym przykładem. Sklep internetowy wysyła wiadomość powitalną każdemu nowemu użytkownikowi. Komponent **Messenger** realizuje logikę odpowiedzialną za wysyłanie wiadomości e-mail, kiedy nowe konto użytkownika zostanie dodane 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 doбором 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>.