Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 5. Funkcje i debugowanie

Zagadnienia do opracowania:

- dekorowanie nazw funkcji
- przeciążanie funkcji
- funkcje rekurencyjne
- makra
- debugger i jego zadania

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest doskonalenie umiejętności implementacji zaawansowanych funkcji w językach *C/C++* oraz zapoznanie z procedurą debugowania kodu. Program laboratorium obejmuje zagadnienia przeciążania funkcji w języku *C++*, implementacji funkcji rekurencyjnych, a także analizy kodu z wykorzystaniem debuggera i oceny jego złożoności obliczeniowej.

2. Wprowadzenie

2.1. Dekorowanie nazw

Każda zmienna i funkcja w języku *C* i *C++* posiada swoją unikatową nazwę (identyfikator). Jest to warunek niezbędny w procesie wiązania symboli przez konsolidator. Nazwa, do której odwołuje się konsolidator, nie jest równoważna z nazwą, którą stosujemy w kodzie (dlatego też możliwe jest np. nadanie tej samej nazwy dla zmiennej globalnej i lokalnej). Kompilator przeprowadza niejawne generowanie unikatowych nazw zmiennych i funkcji na podstawie ich identyfikatorów oraz typów danych. Mechanizm ten nosi nazwę ***dekorowania nazw*** (*ang. name mangling*). Sam schemat dekorowania nazw nie jest sprecyzowany przez standard języka i jest różny, w zależności od implementacji kompilatora. Kompilatory języka *C* z reguły nie przeprowadzają dekorowania nazw, jeżeli nie zachodzi taka konieczność (np. w przypadku przesłaniania zmiennych). Kompilatory języka *C++* z kolei domyślnie dekorują nazwy wszystkich zmiennych i funkcji, zawierając w nich informacje o zagnieżdżającej ***przestrzeni nazw*** czy ***klasie*** [3]. Z tego względu kod skompilowany za pomocą kompilatora języka *C++* (np. kod bibliotek programistycznych) nie może być konsolidowany z modułami języka *C*. Możliwe jest powstrzymanie kompilatora przed przeprowadzeniem procesu dekorowania wybranych symboli za pomocą specyfikatora ***extern "C"***:

```
// Te symbole zostana udekorowane
int add(int x, int y);
int mul(int x, int y);

const float pi = 3.14f;
static unsigned int counter = 0;

// Te symbole nie zostana udekorowane
extern "C" {
  int sub(int x, int y);
  double div(double x, double y);

  float var;
};
```

Listing 1. Powstrzymanie kompilatora języka *C++* przed automatycznym dekorowaniem nazw

Udekorowane nazwy można zobaczyć wykorzystując np. program ***nm***, rozwijany w ramach pakietu *GNU*. Wywołanie polecenia ***nm*** na pliku wykonywalnym z poziomu konsoli poskutkuje wypisaniem wszystkich symboli, które znajdowały się w plikach obiektowych (**.o**). Przykładowo, dla funkcji *int sum(int x, int y)* udekorowana nazwa po kompilacji kompilatorem *g++* może mieć postać *_Z3sumii*. Aby wyświetlić symbole w formie nieudekorowanej można posłużyć się flagą $--$***demangle***.

2.2. Przeciążanie funkcji

W języku *C* nie jest możliwe zdefiniowanie dwóch funkcji o tej samej nazwie, nawet jeżeli posiadają **różne sygnatury** (nagłówek funkcji bez typu zwracanego). Zostanie to zauważone przez kompilator, który zgłosi błąd

redefinicji symbolu. Inaczej jest w przypadku języka *C++*, który wprowadził mechanizm ***przeciążania funkcji i operatorów***. Przeciążanie funkcji (*ang. function overloading*) polega na deklarowaniu wielu funkcji o tej samej nazwie, ale różnej sygnaturze, tj. różniących się listą przyjmowanych argumentów. Mechanizm przeciążania funkcji jest przykładem ***polimorfizmu statycznego*** (polimorfizmu czasu kompilacji). Polimorfizm (inaczej: *wielopostaciowość*) przejawia się w tym, że dana nazwa funkcji odwołuje się do różnych implementacji, w zależności od kontekstu wywołania. Rozpoznanie wywołania właściwej definicji obywa się na etapie kompilacji. Przykład przedstawiono na listingu 2. *Uwaga: aby skompilować program z listingu 2. należy posłużyć się kompilatorem języka C++, np. kompilatorem* ***g++***.

```
1  #include <iostream>
2
3  int sum(int x, int y) {
4      std::cout << "Summing integers " << x << " and "
       << y << std::endl;
5      return x + y;
6  }
7
8  float sum(float x, float y) {
9      std::cout << "Summing floating point numbers "
      << x << " and " << y << std::endl;
10     return x + y;
11 }
12
13 int main() {
14     // Summing integers 1 and 2
15     std::cout << "Sum: " << sum(1, 2) << std::endl;
16
17     // Summing floating point numbers 2.0 and 5.0
```

```
18      std::cout << "Sum: " << sum(2.0f, 5.0f) << std::
    endl;
19      return 0;
20  }
```

Listing 2. Przeciążanie funkcji w języku *C++*

Niemożliwe jest przeciążanie funkcji różniących się jedynie typem zwracanym. W takim przypadku kompilator zgłosi błąd redefinicji.

```
1  int getValue();
2  // Blad kompilacji
3  double getValue();
```

Dzięki mechanizmowi dekorowania nazw, każda funkcja po procesie kompilacji posiada swoją unikatową nazwę. Dlatego mimo wielokrotnego wykorzystywania tej samej nazwy funkcji w kodzie źródłowym w ramach przeciążania funkcji, z punktu widzenia konsolidatora funkcje te mają różne identyfikatory (są to różne symbole).

Analogicznie, jak w przypadku funkcji, można również przeciążać operatory (patrz: tabela 2 [Ćw. 3]). W języku *C++* dozwolone jest przeciążanie wszystkich operatorów za wyjątkiem:

- *operatora ::*
- *operatora .*
- *operatora .**
- *operatora ?:*
- *operatora sizeof*
- *operatorów rzutowania static_cast, dynamic_cast, const_cast, reinterpret_cast*

Przykładem przeciążonego operatora jest *operator«*, wykorzystany na listingu 2. W języku *C++* możliwe jest przeciążanie operatorów dla ***klas***, czyli pewnych abstrakcyjnych typów danych, definiowanych przez programistę. Przykładem klasy jest ***std::ostream***, której ***obiektem***, czyli instancją (realizacją, wcieleniem), jest ***std::cout***. Przeciążenia operatorów dla typów podstawowych są dostarczane przez kompilator.

Jeżeli typ argumentu wywołania funkcji nie jest zgodny z żadną z przeciążonych sygnatur, ale możliwa jest konwersja typu na typ zgodny, to kompilator przeprowadzi niejawne rzutowanie typu zmiennej. W przeciwnym wypadku zostanie zgłoszony błąd kompilacji:

```
1  enum class Colour {
2    BLUE,
3    RED,
4    YELLOW
5  };
6
7  int add(int x, int y) {
8    return x + y;
9  }
10
11 long add(long x, long y) {
12   return x + y;
13 }
14
15 double add(double x, double y) {
16   return x + y;
17 }
18
19 int main() {
20   // Zostanie wywolana funkcja int add(int x, int y)
```

```
21    std::cout << add(2, 3) << std::endl;
22    // Zostanie wywolana funkcja long add(long x, long
        y)
23    std::cout << add(5l, 6l) << std::endl;
24    // Zostanie wywolana funkcja double add(double x,
       double y) (niejawne rzutowanie float -> double)
25    std::cout << add(0.35f, 2.71f) << std::endl;
26    // Blad kompilacji, niejednoznaczna operacja
       rzutowania (unsigned int -> int/long/double)
27    std::cout << add(1u, 4u) << std::endl;
28    // Blad kompilacji, nieokreslona operacja
       rzutowania (Colour -> int/long/double)
29    std::cout << add(Colour::BLUE, Colour::YELLOW) <<
       std::endl;
30    return 0;
31  }
```

2.3. Domyślne argumenty funkcji

Wykorzystując przeciążanie nazw można implementować **funkcje wywoływane z domyślnymi argumentami**, delegując wywołanie do przeciążonej funkcji. Przez domyślny argument określa się argument, z którym zostanie wywołana funkcja, jeżeli nie dostarczymy innego. Na listingu 3. przedstawiono przykład takiego rozwiązania dla funkcji *initializeValue(int)*, przypisującej zadaną wartość do zmiennej *value*. Przeciążająca funkcja bezparametrowa *initializeValue()*, wywołuje przeciążoną funkcję z domyślnym argumentem równym 2. Taka implementacja funkcji z domyślnymi argumentami umożliwia zastąpienie **dowolnej liczby argumentów funkcji, na dowolnej pozycji na liście argumentów** przez domyślne wartości.

```cpp
#include <iostream>

int value;

void initializeValue(int initializer) {
  value = initializer;
}

void initializeValue() {
  initializeValue(2);
}

int main() {
  initializeValue();
  std::cout << "Value = " << value;
  return 0;
}
```

Listing 3. Implementacja funkcji z domyślnym argumentem, wykorzystując przeciążanie funkcji

Język *C++* posiada również konkurencyjny sposób implementacji domyślnych argumentów funkcji. Został on przedstawiony na listingu 4. Taki zapis jest krótszy niż implementacja wykorzystująca przeciążanie funkcji, jednakże ma swoje ograniczenia. **Jeżeli określony argument ma posiadać wartość domyślną (zapisywaną przez = *wartość*), to każdy następujący po nim argument również musi mieć zdefiniowaną wartość domyślną.**

```cpp
#include <iostream>

int value;

void initializeValue(int initializer = 2) {
  value = initializer;
}

int main() {
  initializeValue();
  std::cout << "Value = " << value;
  return 0;
}
```

Listing 4. Implementacja funkcji z domyślnym argumentem

Przy takim podejściu niemożliwe jest przeciążenie funkcji z pustą listą argumentów. Kompilacja programu z listingu 5. zakończy się błędem niejednoznacznego odwołania do funkcji *initializeValue()* (*call of overloaded 'initializeValue()' is ambiguous*).

```cpp
#include <iostream>

int value;

void initializeValue(int initializer = 2) {
  value = initializer;
}

void initializeValue() {
  value = 3;
}
```

```
12
13 int main() {
14   initializeValue();
15   std::cout << "Value = " << value;
16   return 0;
17 }
```

Listing 5. Niejednoznaczne odwołanie do przeciążonej funkcji z domyślnym argumentem

2.4. Rekurencja

Rekurencja (zwana również *rekursją*) to mechanizm, w którym **funkcja wywołuje samą siebie**. Jest to podejście często wykorzystywane w inżynierii oprogramowania, umożliwiające uproszczenie niektórych zagadnień.

Przykład:

Silnia liczby n może być zdefiniowana w sposób iteracyjny:

$$n! = \begin{cases} \prod_{k=1}^{n} k, & gdy\ n \geqslant 1 \\ 1, & gdy\ n = 0 \end{cases}$$

lub rekurencyjny:

$$n! = \begin{cases} n \cdot (n-1)!, & gdy\ n \geqslant 1 \\ 1, & gdy\ n = 0 \end{cases}$$

Na listingu 6. przedstawiono implementację funkcji, realizującej matematyczną operację silni, w podejściu iteracyjnym *iterFactorial()* oraz rekurencyjnym *recurFactorial()*. Funkcja *recurFactorial()* wywołuje samą siebie z wartością przekazanego argumentu pomniejszoną o 1, aż do wywołania *recurFactorial(0)*. Przykładowo, funkcja wywołana z argumentem 3 (*recurFactorial(3)*) wywoła dodatkowo 3 swoje kopie (kolejno z argumentami 2, 1 oraz 0). Wyniki działania poszczególnych kopii będą zbierane od końca i mnożone

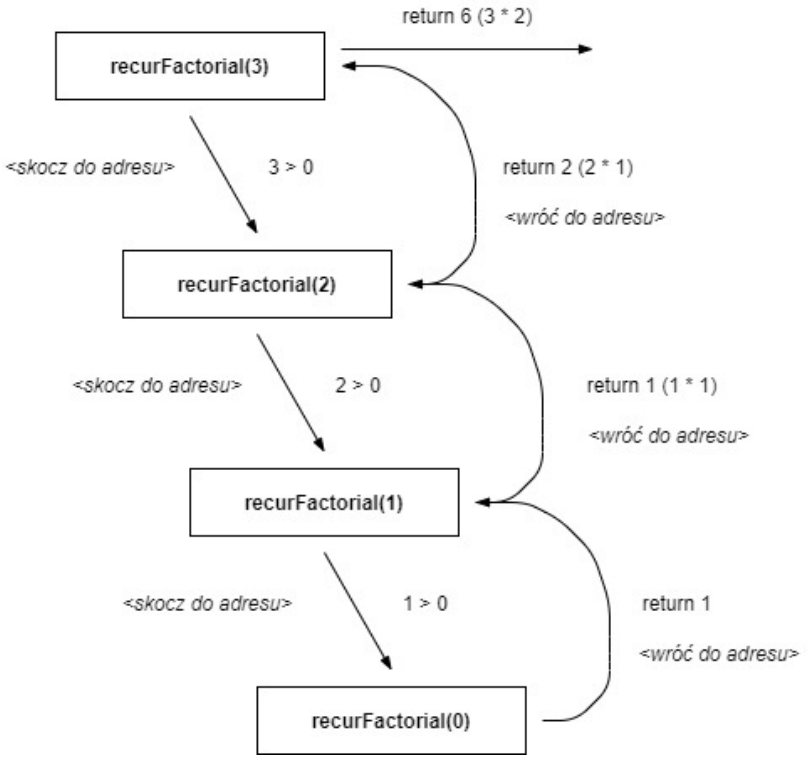
przez wartość argumentu przekazaną do funkcji wywołującej. Schemat rekurencyjnych wywołań funkcji *recurFactorial()* został przedstawiony na rys. 2.1.

```
unsigned int iterFactorial(unsigned int n) {
    unsigned int result = 1;
    while (n > 1) {
        result *= n;
        --n;
    }
    return result;
}

unsigned int recurFactorial(unsigned int n) {
    if (n == 0)
        return 1;
    else
        return n * recurFactorial(n - 1);
}
```

Listing 6. Implementacja silni w podejściu iteracyjnym i rekurencyjnym

Skompilowany kod funkcji (jej ciało) przechowywany jest w segmencie pamięci ***text*** (zwanym również ***code***). Przy każdym wywołaniu funkcji na ***stosie*** odkładany jest **adres powrotu z funkcji**. Oznacza to, że przed skokiem do kodu funkcji, program zapamiętuje adres (lokalizację w pamięci), pod którą ma powrócić po obsłużeniu wywołania funkcji. Zostało to schematycznie zaznaczone na rys. 2.1 – każdorazowo po wywołaniu swojej kopii, funkcja *recurFactorial()* powraca do funkcji wywołującej. Wynika z tego bardzo ważny wniosek. **Liczba rekurencyjnych wywołań funkcji nie jest nieskończona. Górnym ograniczeniem jest rozmiar *stosu*.** Należy również pamiętać, aby prawidłowo zdefiniować ***warunek zakończenia rekurencji*** (tu: n == 0). W przeciwnym razie funkcja może wywoływać



Rys. 2.1. Rekurencyjne wywołanie funkcji

swoje kopie tak długo, aż zapełni całą dostępną pamięć stosu, a program zostanie przerwany z komunikatem o przekroczeniu dopuszczalnego rozmiaru pamięci.

Rekurencja może uprościć niektóre problemy programistyczne, ale należy mieć na uwadze idące za tym konsekwencje. Poza problemem nieograniczonej rekurencji, niepożądane efekty uboczne stosowania rekurencji, jakie można napotkać, to między innymi: wielokrotne wykonywanie tych samych operacji (np. ciąg Fibonacciego), nieoczywistość kodu i utrudnione wprowadzanie modyfikacji czy wydłużenie czasu działania programu (wynikające ze skoków i powrotów z adresu funkcji). Przykładami algorytmów, które mogą być realizowane za pomocą rekurencji są: algorytm Euklidesa (poszukiwanie największego wspólnego dzielnika), algorytmy sortowania, algorytmy przeliczania systemów liczbowych, algorytm potęgowania, algorytm silni, itp.

2.5. Makra

Zarówno w języku *C* jak i *C++* można spotkać się z pojęciem ***makra***. ***Makro*** to określony fragment kodu, który ***preprocesor*** umieści w miejscu zdefiniowanej nazwy makra. Za pomocą makr można definiować pewne ***stałe numeryczne***, ale również instrukcje, które mają zostać wykonane (takie makra mogą przyjmować argumenty na wzór ***funkcji***). Makra definiowane są z wykorzystaniem dyrektywy preprocesora ***#define*** (listing 7). Warto zwrócić uwagę, że po definicji makra **nie stawia się średnika**.

```c
#include <stdio.h>

// Makrodefinicja stalej
#define MY_LUCKY_NUMBER 666

// Makro funkcyjne
#define print_lucky_number() printf("%d\n",
   MY_LUCKY_NUMBER)

#define sum(x, y) (x + y)

int main() {
    // Wyswietla 666
    print_lucky_number();

    // Wyswietla 5
    printf("sum: %d\n", sum(2, 3));
    return 0;
}
```

Listing 7. Definiowanie makr

Wywołanie makra jest szybsze niż wywołanie funkcji, ponieważ implementacja makra jest wstawiana przez preprocesor w miejscu wywołania - nie są wykonywane instrukcje skoku i powrotu z adresu funkcji. Jednakże, makra powinny być stosowane zamiast funkcji **jedynie w szczególnych przypadkach**. Wynika to z ich mniejszego bezpieczeństwa i utrudnionego debugowania kodu. Przykładem łatwego do przeoczenia błędu jest pominięcie nawiasu w definicji makra ***sum()*** z listingu 7. Wykorzystanie makra w wyrażeniu $sum(2,3) * 2$ poskutkuje zastąpieniem go przez preprocesor, jeszcze przed kompilacją programu, wyrażeniem o postaci $2 + 3 * 2$. Zgodnie z priorytetem operatorów, w pierwszej kolejności zostanie wykonana instrukcja mnożenia, a dopiero później dodawanie. Taka sytuacja nie wystąpiłaby w przypadku zastosowania funkcji (listing 8).

```
#include <stdio.h>

// Usunieto nawiasy z definicji makra
#define sum(x, y) x + y

int sumFunc(int x, int y) {
    return x + y;
}

int main() {
    // Wyswietla 8 zamiast 10
    printf("sum: %d\n", sum(2, 3) * 2);

    // Wyswietla 10
    printf("sum: %d\n", sumFunc(2, 3) * 2);
    return 0;
}
```

Listing 8. Błędna definicja makra

2.6. Debugger

Debugger to program służący do analizy innych programów **w czasie ich wykonania**. ***Debugowanie*** (dynamiczna analiza z wykorzystaniem debuggera) umożliwia śledzenie kolejności wykonania instrukcji w kodzie, wartości przyjmowanych przez poszczególne zmienne czy zatrzymywanie wykonania programu po wykonaniu określonych instrukcji. Upraszcza to proces wykrywania błędów w napisanym kodzie aplikacji. Przykładem debuggera jest program ***GDB***, rozwijany w ramach pakietu *GNU*. Większość współczesnych zintegrowanych środowisk programistycznych (*IDE*) posiada również wbudowane debuggery, ułatwiające implementację aplikacji. ***GDB*** jest sterowany za pomocą poleceń konsolowych. Podstawowe polecenia zostały zebrane w tabeli 1. Pełna lista poleceń programu ***GDB*** zostanie wyświetlona po wywołaniu ***debuggera*** z flagą **--help**.

Aby uruchomić aplikację za pomocą ***debuggera*** należy podczas kompilacji programu wygenerować ***symbole debugowe***. Stanowią one informację o adresach zmiennych, stałych i funkcji w pamięci komputera, typach zmiennych, nazwach plików i liniach kodu, w których zostały zdefiniowane określone symbole. Umożliwia to ustawianie tzw. ***punktów przerwania*** (*ang. breakpoints*) w określonych liniach kodu, w celu zatrzymania działania aplikacji w zdefiniowanych stanach. Aby wygenerować ***symbole debugowe*** można posłużyć się flagą **-g** kompilatora **gcc** (lub **g++**):

gcc -g main.c

Należy jednak mieć na uwadze, aby generować ***symbole debugowe*** tylko dla celu ***debugowania aplikacji***, ponieważ mogą one **istotnie zwiększyć rozmiar pliku wykonywalnego**.

Tabela 1. Podstawowe polecenia debuggera *GDB* [2]

Polecenie	**Opis**
r	uruchom działanie aplikacji do wystąpienia punktu przerwania lub końca programu
b *func*	ustaw punkt przerwania na początku funkcji *func*
b *file.c:func*	ustaw punkt przerwania na początku funkcji *func* w pliku źródłowym *file.c*
b *N*	ustaw punkt przerwania na *N-tej* linii kodu aktualnie wykonywanego pliku źródłowego
b *file.c:N*	ustaw punkt przerwania na *N-tej* linii kodu pliku źródłowego *file.c*
d *N*	usuń *N-ty* punkt przerwania
i b	wypisz wszystkie punkty przerwania
c	wznów działanie aplikacji do wystąpienia kolejnego punktu przerwania lub końca programu
f	wykonaj bieżącą funkcję do końca
s	wykonaj kolejną linię kodu
s *N*	wykonaj kolejne *N* linii kodu
n *N*	wykonaj kolejne *N* linii kodu nie licząc linii kodu w ciałach wywoływanych funkcji
p *var*	wypisz aktualną wartość zmiennej *var*
set *var=val*	przypisz zmiennej *var* wartość *val*
bt	wyświetl *stack trace*
q	zakończ działanie debuggera

Aby uruchomić aplikację w ***debuggerze GDB*** należy wywołać w konsoli polecenie:

gdb *nazwa_pliku_wykonywalnego*

Zostanie wyświetlona informacja o poprawnie zaczytanych symbolach, a ***debugger*** będzie oczekiwał na pierwsze polecenie (zostanie otwarta konsola ***(gdb)***). Aby zakończyć ***sesję debugową*** należy wykonać polecenie ***q*** (*quit*).

Przykład: Aplikacja składa się z trzech plików: *main.c*, *factorial.h* oraz *factorial.c*. Plik *factorial.h* zawiera deklarację funkcji *factorial()*, wyznaczającej wartość silni zadanej liczby naturalnej n:

```
#pragma once

unsigned int factorial(unsigned int n);
```

Definicja funkcji *factorial()* została umieszczona w pliku źródłowym *factorial.c*:

```
#include "factorial.h"

unsigned int factorial(unsigned int n) {
    unsigned int result = 1;
    while (n > 1) {
        result *= n;
        --n
    }
    return result;
}
```

Funkcja *factorial()* jest wywoływana z ciała funkcji *main()* (plik *main.c*):

```
#include <stdio.h>
#include "factorial.h"

int main() {
    printf("Factorial of 5: %d", factorial(5));
    return 0;
}
```

Aby prześledzić działanie aplikacji posługując się programem ***GDB*** należy w pierwszej kolejności skompilować kod:

gcc -g main.c factorial.c -o app

Wygenerowany zostanie plik wykonywalny *app*, zawierający ***symbole debugowe*** (na tym etapie można również skompilować aplikację bez ***symbolów debugowych*** i porównać rozmiar obu plików wykonywalnych). Aby rozpocząć ***sesję debugową*** należy wykonać polecenie:

gdb app

Aby ustawić ***punkt przerwania*** na początku funkcji *factorial()* można posłużyć się poleceniem:

b factorial.c:factorial

Debugger informuje, że ***punkt przerwania*** o numerze 1 został ustawiony w linii czwartej pliku źródłowego *factorial.c*:

Breakpoint 1 at 0x4004b9: file factorial.c, line 4

Polecenie **r** uruchomi aplikację i zatrzyma jej wykonanie na ustawionym ***punkcie przerwania***:

Breakpoint 1, factorial (n=5) at factorial.c:4
4 *unsigned int result = 1;*

Polecenie **s** spowoduje przejście do kolejnej linii kodu:

5 *while (n > 1) {*

Aby prześledzić wartość zmiennej *result* po dwóch iteracjach pętli można ustawić ***punkt przerwania*** na linii nr 6 (*result *= n;*):

b 6,

usunąć ***punkt przerwania*** z linii 4 (ustawiony na początku sesji):

d 1,

a następnie wykorzystać trzykrotnie polecenie **c**. Polecenie **p result** wyświetli aktualną wartość zmiennej *result*:

$$\$1 = 20$$

Wyświetlony wynik jest zgodny z oczekiwaniami, ponieważ po dwóch iteracjach pętli wartość zmiennej *result* powinna wynosić $1 \cdot 5 \cdot 4 = 20$. Można usunąć ***punkt przerwania*** (**d 2**) i wznowić działanie aplikacji (**c**). Wyświetlona zostanie informacja, że program zakończył działanie poprawnie, np.:

Continuing.
[Inferior 1 (process 9853) exited normally]

Polecenie **q** zakończy działanie ***debuggera***.

2.7. Pomiar czasu wykonania funkcji

W celu pomiaru czasu wykonania bloku kodu w języku *C* można posłużyć się funkcją ***clock()***, zdeklarowaną w pliku nagłówkowym ***time.h***. Funkcja zwraca liczbę cykli procesora, która minęła od uruchomienia aplikacji, w postaci zmiennej typu ***clock_t***[1]. Wartość ta może być przeliczona na sekundy po podzieleniu przez makro ***CLOCKS_PER_SEC***. Przykład pomiaru czasu wykonania funkcji *func()* przedstawiono na listingu 9.

```
1 #include <stdio.h>
2 #include <time.h>
3
4 void func() {
5     for(int i = 0; i < 100000; ++i)
6     printf("Iteration count: %d\n", i);
7 }
```

[1] typ arytmetyczny reprezentujący liczbę cykli procesora

```
8
9  int main() {
10     clock_t begin = clock();
11     func();
12     clock_t end = clock();
13
14     // Czas wykonania w milisekundach
15     double duration = (double)(end - begin) /
       CLOCKS_PER_SEC * 1000;
16     printf("Function executed in %.2f ms", duration)
       ;
17     return 0;
18 }
```

Listing 9. Pomiar czasu wykonania funkcji z wykorzystaniem funkcji *clock()*

W języku *C++* biblioteka ***chrono*** udostępnia narzędzia umożliwiające pomiar czasu wykonania funkcji (dostępna od standardu *C++*11). W tym celu należy zapisać dwa punkty w czasie – przed wywołaniem i po wywołaniu funkcji, wykorzystując funkcję ***std::chrono::high_resolution_clock::now()***, a następnie zrzutować różnicę punktów na odpowiednią jednostkę czasu. Przykład pomiaru czasu wykonania funkcji *func()* przedstawiono na listingu 10. Słowo kluczowe ***auto*** od standardu *C++*11 umożliwia automatyczną dedukcję typu zmiennej (przez kompilator).

```cpp
#include <iostream>
#include <chrono>

void func() {
    for(int i = 0; i < 100000; ++i)
    std::cout << "Iteration count: " << i << std::
   endl;
}

int main() {
    auto begin = std::chrono::high_resolution_clock
   ::now();
    func();
    auto end = std::chrono::high_resolution_clock::
   now();

    // Czas wykonania w milisekundach
    auto duration = std::chrono::duration_cast<std::
   chrono::milliseconds>(end - begin).count();
    std::cout << "Function executed in: " <<
   duration << " ms";
    return 0;
}
```

Listing 10. Pomiar czasu wykonania funkcji z wykorzystaniem biblioteki *chrono*

3. Program ćwiczenia

Zadanie 1. W pliku nagłówkowym ***multiply.h*** zamieszczono deklarację funkcji ***multiply(int, int)***, wykonującej mnożenie dwóch liczb całkowitych. Definicja tej funkcji znajduje się w pliku źródłowym ***multiply.cpp***. Przeciąż funkcję ***multiply()***, aby możliwe było wykonywanie operacji mnożenia na liczbach zmiennoprzecinkowych. Obie funkcje wywołaj wewnątrz funkcji ***main()*** (plik ***main.cpp***). Wykorzystując program ***nm*** (pakiet *GNU*) sprawdź, jak wyglądają udekorowane nazwy funkcji ***multiply(int, int)*** oraz ***multiply(float, float)***. Co się stanie, gdy deklarację funkcji ***multiply(int, int)*** zagnieździ się w bloku ***extern "C"***?

Zadanie 2. W pliku nagłówkowym ***fib.h*** zamieszczono deklarację funkcji ***fibonacci(unsigned int)***, obliczającą *n-ty* wyraz ciągu Fibonacciego. Ciąg ten może być określony rekurencyjne w następujący sposób:

$$F(n) = \begin{cases} 0, & gdy\ n = 0 \\ 1, & gdy\ n = 1 \\ F(n-1) + F(n-2), & gdy\ n > 1 \end{cases}$$

Rekurencyjna implementacja funkcji ***fibonacci(unsigned int)*** została umieszczona w pliku źródłowym ***fib.cpp***. Prześledź działanie funkcji za pomocą debuggera ***GDB*** i znajdź błąd zawarty w implementacji rekurencyjnej, a następnie zaimplementuj poprawnie ten algorytm w sposób iteracyjny.

Zadanie 3. W pliku nagłówkowym ***gcd.h*** zamieszczono deklaracje funkcji ***iterGcd(int, int)*** oraz ***recurGcd(int, int)***, wyznaczających największy wspólny dzielnik dwóch liczb, w podejściu odpowiednio iteracyjnym i rekurencyjnym. Zaimplementuj obie funkcje (definicje umieść w pliku ***gcd.cpp***), a następnie wywołaj obie funkcje wewnątrz funkcji ***main()*** (plik ***main.cpp***) z tym samym zestawem parametrów i porównaj otrzymane wyniki. Porównaj

czas wykonywania obu funkcji wykorzystując bibliotekę ***chrono*** języka *C++* lub funkcję ***clock()*** języka *C*.

4. Dodatek

4.1. Rekurencja ogonowa

Szczególną odmianą rekurencji jest ***rekurencja ogonowa*** (*ang. tail recursion*). Warunek kwalifikacji rekurencji jako ***rekurencji ogonowej*** określa, że ostatnią operacją wykonywaną przez funkcję ma być rekurencyjne wywołanie samej siebie lub zwrócenie ostatecznego wyniku. Stosowana jest w celu zwiększenia wydajności algorytmów rekurencyjnych i zmniejszenia zajętości stosu przez adresy powrotu z funkcji. Algorytm zaimplementowany za pomocą rekurencji ogonowej może być w łatwy sposób przepisany na podejście iteracyjne. Przykład implementacji silni z wykorzystaniem rekurencji ogonowej przedstawiono na listingu 11.

```
1 int factorial(int n, int currentValue = 1) {
2     if (n == 0)
3         return currentValue;
4     else
5         return factorial(n - 1, currentValue * n);
6 }
7
```

Listing 11. Implementacja silni za pomocą rekurencji ogonowej

Zadanie (dla chętnych) Prześledź kolejne rekurencyjne wywołania funkcji *factorial()* z listingu 11. i porównaj ze schematem wywołań funkcji *recurFactorial()* z rys. 2.1. (listing 6).

Zadanie (dla chętnych) Czy rekurencyjna implementacja funkcji *recurGcd()* z zadania 3. jest przykładem rekurencji ogonowej?

4.2. Specyfikator *inline*

Czasami w przypadku trywialnych funkcji może wystąpić sytuacja, że czas przełączania kontekstu (skok do kodu funkcji i powrót do funkcji wywołującej) jest dłuższy niż czas wykonania samego ciała wywoływanej funkcji. Język *C++* umożliwia optymalizację takich funkcji przez zastosowanie specyfikatora ***inline***. W czasie kompilacji podmieniane są wywołania funkcji *inline* bezpośrednio na kod tych funkcji. W ten sposób pomijane są instrukcje skoku i powrotu z adresu funkcji. Należy mieć na uwadze, że specyfikator ***inline*** stanowi jedynie sugestię dla kompilatora. To, że określimy funkcję jako *inline* nie oznacza, że kompilator przeprowadzi operację podstawienia, jeśli uzna ją za nieoptymalną. Analogicznie, niezastosowanie specyfikatora *inline* nie oznacza, że dla wyższych poziomów optymalizacji, kompilator sam nie przeprowadzi operacji podstawienia kodu funkcji w miejscach jej wywołań. Wadami stosowania funkcji *inline* jest między innymi zwiększony rozmiar pliku binarnego (co może wykluczać zastosowanie specyfikatora *inline* w przypadku niektórych architektur mikrokontrolerów o małym rozmiarze pamięci) czy zwiększenie czasu kompilacji. Warunkiem deklarowania funkcji za pomocą specyfikatora *inline* jest **nierozdzielanie deklaracji i definicji**, tzn. definicja funkcji stanowi jednocześnie jej deklarację. Przykład deklaracji funkcji *inline* przedstawiono na listingu 12.

```
1 inline int square(int x) {
2     return x * x;
3 }
```

Listing 12. Przykład zastosowania specyfikatora *inline*

4.3. Automatyczna dedukcja typu zmiennej

Począwszy od standardu *C++11* do języka został wprowadzony mechanizm ***automatycznej dedukcji typów zmiennych***, a od standardu *C++14* może być on stosowany również w odniesieniu do typu zwracanego z funkcji. Mechanizm ten pozwala na ***definiowanie*** zmiennych z użyciem słowa kluczowego ***auto***, delegując odpowiedzialność za dedukcję właściwego typu zmiennych do kompilatora. Typ zmiennej jest określany automatycznie podczas inicjalizacji zmiennej (na podstawie typu argumentu przypisania):

```
// Od C++14
// Typem zwracanym jest float
auto mul(float a, float b) {
    return a * b;
}

// Od C++11
// Zmienna x jest typu int
auto x = 2 + 5;
// Zmienna y jest typu float
y = mul(2.3f, 7.1f);
// Zmienna text jest typu const char *
auto text = "Ala ma kota";
// Zmienna ref jest typu const int &
const auto & ref = x;
// Zmienna timePoint jest typu std::chrono::
   time_point<std::chrono::high_resolution_clock>
auto timePoint = std::chrono::high_resolution_clock
   ::now();
```

Listing 13. Automatyczna dedukcja typu zmiennej

Deklaracja zapowiadająca zmiennej z użyciem słowa kluczowego ***auto*** zakończy się błędem kompilacji (*error: declaration of (...) has no initializer*):

```
1 // Blad kompilacji, niemozliwa automatyczna dedukcja
      typu (zmienna niezainicjalizowana)
2 auto x;
```

Przed wprowadzeniem standardu *C++11* słowo kluczowe ***auto*** stanowiło jeden ze specyfikatorów czasu życia i rodzaju wiązania zmiennych (obok innych słów kluczowych, jak ***register***, ***static***, ***extern*** czy ***thread_local***) i odnosiło się do **zmiennych automatycznych**. Ponieważ deklarowanie zmiennych z użyciem specyfikatora ***auto*** było dozwolone wyłącznie wewnątrz bloków (lub listy parametrów funkcji), to specyfikator ***auto*** pełnił rolę wyłącznie „dekoracyjną" (jego umieszczenie w kodzie nie wpływało w żaden sposób na deklarowane zmienne). Standard *C++11* zmienił znaczenie słowa kluczowego ***auto*** przez realizację mechanizmu ***automatycznej dedukcji typu zmiennych***.

Nadmierne wykorzystywanie mechanizmu ***automatycznej dedukcji typu zmiennych*** może prowadzić do istotnego zmniejszenia czytelności kodu. Dobrą praktyką jest niestosowanie słowa kluczowego ***auto*** w przypadku typów podstawowych (*int*, *float*, *char*, *unsigned long*, itp.), a ograniczenie go do dedukcji skomplikowanych typów złożonych. Przykładami mogą być:

- *std::unique_ptr<T, std::function<void(T*)>>*;
- *std::unordered_map<std::string, std::vector<int>>*;
- *std::chrono::time_point<std::chrono::high_resolution_clock>*.

Jeżeli wyłącznym celem stosowania ***automatycznej dedukcji typu*** jest zmniejszenie liczby znaków potrzebnych do zdeklarowania zmiennej, to alternatywną metodą może być mechanizm ***aliasów typów*** [więcej o aliasach typów w Ćw. 9].

4.4. Złożoność obliczeniowa

Istotnym parametrem opisującym implementację danego algorytmu jest czas wykonania. Informacja ta wydaje się mało przydatna, jeżeli czas wykonania jest zależny od szybkości maszyny, na której uruchamiany jest program. W celu uniezależnienia tego parametru od budowy komputera, wprowadzono pojęcie ***złożoności obliczeniowej***, wiążącej liczbę przeprowadzanych ***operacji dominujących*** (***czasowa złożoność obliczeniowa***) lub wymagany rozmiar pamięci (***pamięciowa złożoność obliczeniowa***) od rozmiaru danych wejściowych. ***Operacjami dominującymi (elementarnymi)*** nazywamy operacje, których liczba jest proporcjonalna do liczby wszystkich wykonanych operacji jednostkowych w dowolnej implementacji algorytmu. Z reguły operacje elementarne stanowią największy odsetek w całościowym czasie wykonania algorytmu. Wykonanie jednej operacji dominującej uznawane jest za **jednostkę złożoności czasowej**. Złożoność obliczeniowa może być przedstawiona w postaci funkcji rozmiaru danych wejściowych $f(n)$, jak np.:

$$f(n) = n^3 - 4n^2 + 12,\ n > 0,$$

gdzie n – rozmiar danych wejściowych w bitach.

Precyzyjne wyznaczenie złożoności obliczeniowej algorytmu jest często trudne i czasochłonne. Najpopularniejszą metodą przedstawiania przybliżonej wartości ***czasowej złożoności obliczeniowej*** algorytmów jest tzw. ***notacja dużego O***. Jest to pesymistyczne oszacowanie złożoności obliczeniowej, tj. przy założeniu najbardziej skomplikowanego przypadku ułożenia danych. Złożoność obliczeniowa w notacji dużego O jest równa najbardziej znaczącemu członowi funkcji $f(n)$, z pominięciem stałego współczynnika. Stąd, dla powyższego przykładu złożoność obliczeniowa w notacji dużego O wynosić będzie:

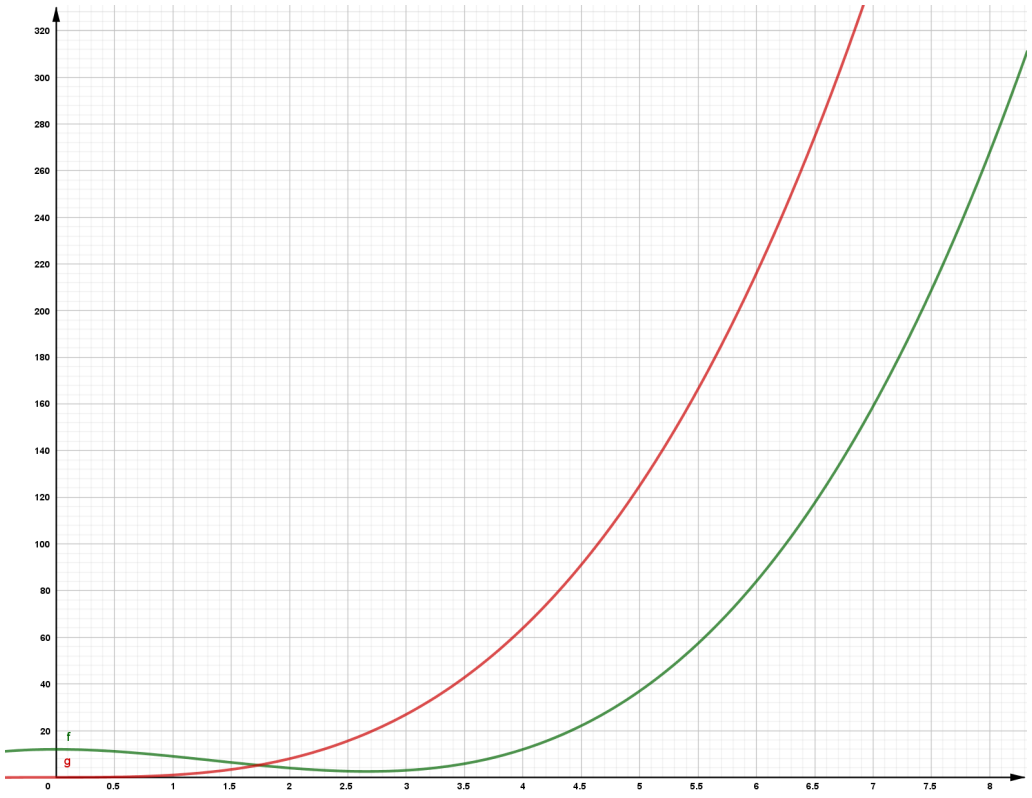
$$f(n) = n^3,$$

co skrótowo zapisywane jest jako $O(n^3)$. Oczywiście notacja dużego O stanowi pewne oszacowanie złożoności obliczeniowej. Dlatego też dla małych

liczb $\boldsymbol{n}$ wartość oszacowana będzie mniejsza niż złożoność rzeczywista. Jednakże dla pewnego dostatecznie dużego $\boldsymbol{n}$ człon dominujący (tu: n^3) staje się na tyle duży, że pozostałe człony mogą zostać pominięte. Formalna definicja $O(g(n))$ stanowi, że istnieje takie n_0, dla którego każde $n > n_0$ spełnia nierówność $f(n) \leqslant c \cdot g(n)$, dla każdego $c > 0$.

$$O(g(n)) = \{f(n) : \exists_{c>0}\ \exists_{n_0 \in \mathbb{N}}\ \forall_{n \geqslant n_0}\ f(n) \leqslant c \cdot g(n)\}$$

Na rys. 4.1. przedstawiono wykresy funkcji $f(n) = n^3 - 4n^2 + 12$ (zielona) oraz $g(n) = n^3$ (czerwona). Dla $n_0 = 2$ spełniona jest zależność $g(n) > f(n)$, stąd $g(n)$ ogranicza z góry $f(n)$.



Rys. 4.1. Złożoność obliczeniowa rzeczywista (f) i oszacowanie z góry (g) – *notacja dużego O*

Najczęściej spotykane złożoności obliczeniowe algorytmów w kolejności rosnącej:

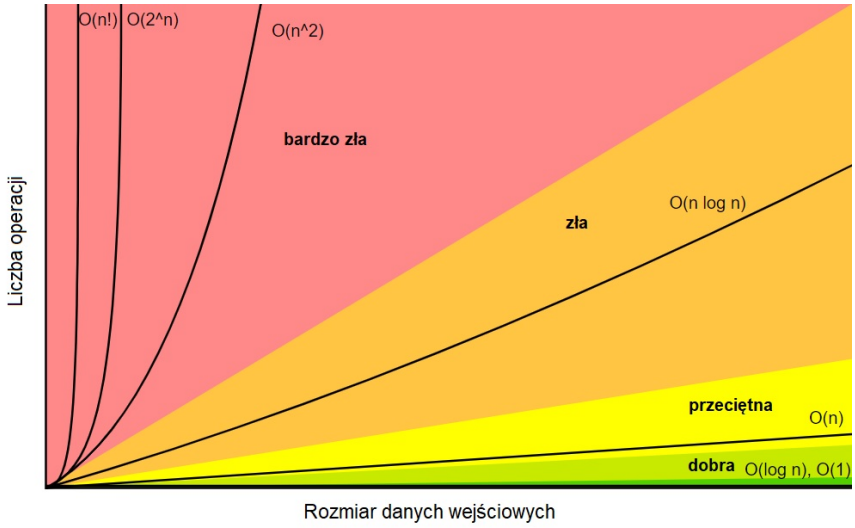
- $O(1)$ – złożoność stała
- $O(log(n))$ – złożoność logarytmiczna
- $O(n)$ – złożoność liniowa
- $O(n\ log(n))$ – złożoność liniowo-logarytmiczna
- $O(n^2)$ – złożoność kwadratowa
- $O(n^k),\ k \in \mathbb{N}_+$ – złożoność wielomianowa
- $O(k^n),\ k \in \mathbb{N}_+$ – złożoność wykładnicza
- $O(n!)$ – złożoność rzędu silnia

Porównanie charakterystyk dla wybranych złożoności obliczeniowych przedstawiono na rys. 4.2. Czasowo-wydajne programy charakteryzują się jak najmniejszą złożonością obliczeniową (najlepiej logarytmiczną lub stałą).

Na listingu 14. przedstawiono przykład algorytmu o liniowej złożoności obliczeniowej $O(n)$. Oblicza on sumę wszystkich liczb naturalnych nie większych niż $\boldsymbol{n}$. Pętla *for* iteruje $\boldsymbol{n}$ razy, za każdym razem zwiększając wynik o aktualną wartość iteratora. Zatem czas wykonania algrytmu jest wprost proporcjonalny do rozmiaru danych wejściowych.

```
1 int sumInRange(int n) {
2     int result = 0;
3     for (int i = 1; i <= n; ++i)
4         result += i;
5     return result;
6 }
```

Listing 14. Algorytm o złożoności $O(n)$



Rys. 4.2. Porównanie charakterystyk wybranych złożoności obliczeniowych [1]

4.5. Standardy języków *C/C++*

Język *C* powstał w roku 1972 jako następca języka *B* do celu implementacji jądra systemu operacyjnego ***Unix*** (1973 r.). Jego projektantem był **Dennis Ritchie**, pracujący wówczas w Bell Labs. W latach '80 ubiegłego wieku język *C* zyskał znaczną popularność jako język programowania systemów operacyjnych, w związku z czym ***Amerykański Narodowy Instytut Normalizacyjny*** (*ang. American National Standards Institute, ANSI*) powołał komitet, mający za zadanie opracowanie pierwszego standardu języka *C*. W 1989 roku zatwierdzony został standard **ANSI X3.159-1989 "Programming Language C"**, znany również jako ***ANSI C*** lub ***C89***. ***ANSI C*** zostało również zatwierdzone przez ***Międzynarodową Organizację Normalizacyjną*** (*ang. International Organization for Standardization, ISO*) jako standard **ISO/IEC 9899:1990**, zwany potocznie standardem ***C90*** [5]. Najnowsza wersja standardu języka *C* została opublikowana przez ***ISO*** w 2018 roku – ***ISO/IEC 9899:2018*** (***C18***).

W roku 1985 **Bjarne Stroustrup** opublikował książkę ***The C++ Pro-***

gramming Language, stanowiącą referencję dla opracowanego przez niego nowego języka programowania [4]. Książka była wielokrotnie wznawiana w kolejnych edycjach wraz z rozwojem języka. Język *C++* stanowił rozszerzenie języka *C* o mechanizmy ***programowania obiektowego*** (język *C++* początkowo był roboczo nazywany *językiem C z klasami*). Podobnie jak język *C*, również *C++* jest standaryzowany przez ***ISO***. Do tej pory zatwierdzone zostało sześć standardów języka *C++* (zestawione w tabeli 2). Jednym z najważniejszych założeń komitetu standaryzacyjnego jest ***wsteczna kompatybilność*** (z niewielkimi odstępstwami) zarówno z poprzednimi standardami języka *C++*, jak również z językiem *C*.

Tabela 2. Standardy ISO języka *C++*

Wersja	**Rok**	**Kluczowe cechy**
ISO/IEC 14882:1998 (*C++98*)	1998	pierwszy oficjalny standard języka *C++*
ISO/IEC 14882:2003 (*C++03*)	2003	naprawiono błędy biblioteki standardowej z wersji *C++98*
ISO/IEC 14882:2011 (*C++11*)	2011	semantyka przenoszenia, inteligentne wskaźniki, uogólnione wyrażenia stałe, wskaźnik *nullptr*, klasy wyliczeniowe, składnia jednolitej inicjalizacji, listy inicjalizacyjne, automatyczna dedukcja typu, jawne operatory konwertujące, pętla *for* po zakresie, deklaracje *default* i *delete*, słowa kluczowe *override* i *final*, wyrażenia lambda, statyczne asercje, referencja adapterowa, krotki, lokalna pamięć wątku, szablony ze zmienną listą parametrów, biblioteka *std::chrono*
ISO/IEC 14882:2014 (*C++14*)	2014	automatyczna dedukcja zwracanego typu, generyczne wyrażenia lambda, szablony zmiennych, standardowe literały znakowe, separatory cyfr
ISO/IEC 14882:2017 (*C++17*)	2017	*std::string_view*, *std::optional*, *std::any*, *std::variant*, biblioteka systemu plików, algorytmy równoległe *Standardowej Biblioteki Szablonów*, *std::byte*, literały heksadecymalnych liczb zmiennoprzecinkowych, wyrażenie *if* czasu kompilacji, zmienne *inline*, dedukcja argumentów klasy szablonowej (*CTAD*)
ISO/IEC 14882:2020 (*C++20*)	2020	koncepty, trójwynikowy operator porównania, warunkowy specyfikator *explicit*, słowa kluczowe *consteval* i *constinit*, korutyny, atomowe inteligentne wskaźniki, *std::span*, zakresy (*ang. ranges*)

Literatura

[1] *Big-O Algorithm Complexity Cheat Sheet*. URL: https://www.bigocheatsheet.com/.

[2] *GDB Tutorial. Commands*. URL: http://www.gdbtutorial.com/tutorial/commands.

[3] *Name mangling in C*. URL: https://gustedt.wordpress.com/2011/06/24/name-mangling-in-c/.

[4] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840.

[5] *The Origin of ANSI C and ISO C*. URL: https://blog.ansi.org/2017/09/origin-ansi-c-iso-c/#gref.