

Sprawozdanie z zajęć: „Podstawy Programowania” z dnia 27.10.2022r.

## Laboratorium 3

**Imię i nazwisko:** Patryk Kozłowski / Informatyka Techniczna

**Temat:** Zmienne i operatory w C

**Cel:** Opanowanie tworzenia prostych programów w C realizujących tworzenie napisów.

**Opis:**

1. Utworzyłem folder lab3

```
umarly-poeta@new-host projects/pp (master) » mkdir lab3
```

2. utworzyłem plik hello.c i napisałem go zgodnie z wykładem

```
1  #include <stdio.h>
2
3  int main()
4  {
5      // hello world
6      printf("Hello World!\n");
7      /*
8       * cos tam cos tam
9       * */
10
11     printf("*****\n");
12     printf(" *   *\n");
13     printf(" *   *\n");
14     printf(" *   *\n");
15     printf("*****\n");
16
17     printf("  *  \n");
18     printf(" *** \n");
19     printf("*****\n");
20     return 0;
21 }
22
```

3. Skompilowałem plik źródłowy i wykonałem go w terminalu, następnie zmodyfikowałem, aby rysował kwadrat i trójkąt złożony ze znaków ASCII

```

umarly-poeta@new-host pp/lab3 (master) » bash script.bash
Rozpoczynam kompilację programu...
Kompiluję hello.c z opcjami -std=c11 -O2 -march=native...
Kompilacja zakończona sukcesem.
Uruchamiam program...
Hello World!
*****
* *
* *
* *
*****
*
**
*****
Kompiluję hello.c z opcjami -std=c11 -O2 -march=native -g...
Kompilacja zakończona sukcesem.
Uruchamiam program w trybie debugowania...
Reading symbols from ./hello...
Breakpoint 1 at 0x401040: file hello.c, line 6.
Starting program: /home/umarly-poeta/projects/pp/lab3/hello

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main () at hello.c:6
6      printf("Hello World!\n");
0x00007ffffe350ba in __GI_IO_puts (str=str@entry=0x402010 "Hello World!")
    at ioputs.c:35
35     size_t len = strlen (str);
List of classes of commands:

aliases — User-defined aliases of other commands.
breakpoints — Making program stop at certain points.
data — Examining data.
files — Specifying and examining files.
internals — Maintenance commands.
obscure — Obscure features.
running — Running the program.
stack — Examining the stack.
status — Status inquiries.
support — Support facilities.
text-user-interface — TUI is the GDB text based interface.
tracepoints — Tracing of program execution without stopping the program.
user-defined — User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
—Type <RET> for more, q to quit, c to continue without paging—c
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
A debugging session is active.

```

4. Napisałem prosty skrypt w bashu, który kompiluje plik źródłowy z różnymi flagami, dodałem również komentarze i polecenia wypisujące tekst w terminalu (echo), następnie dodałem automatyczne odpalenie debuggera z breakpointami i helpem, aby pomóc zaznajomić się z gdb

```
$ script.bash
You, 19 minutes ago | 1 author (You)
1  #!/bin/bash
2
3  # Kompilacja programu z wybranymi opcjami
4  echo "Rozpaczynam kompilację programu..."
5
6  # Opcje kompilacji
7  CFLAGS="-std=c11 -O2 -march=native"
8
9  # Nazwa pliku źródłowego i wynikowego
10 SOURCE="hello.c"
11 OUTPUT="hello"
12
13 # Kompilacja
14 echo "Kompiluję $SOURCE z opcjami $CFLAGS..."
15 gcc $CFLAGS -o $OUTPUT $SOURCE
16
17 # Sprawdzenie, czy kompilacja zakończyła się sukcesem
18 if [ $? -eq 0 ]; then
19     echo "Kompilacja zakończona sukcesem."
20     echo "Uruchamiam program..."
21     ./$OUTPUT
22 else
23     echo "Błąd podczas kompilacji."
24 fi
25
26 # Kompilacja z opcją -g (debugowanie)
27 CFLAGS_DEBUG="-std=c11 -O2 -march=native -g"
28 echo "Kompiluję $SOURCE z opcjami $CFLAGS_DEBUG..."
29 gcc $CFLAGS_DEBUG -o $OUTPUT $SOURCE
30
31 # Sprawdzenie, czy kompilacja zakończyła się sukcesem
32 if [ $? -eq 0 ]; then
33     echo "Kompilacja zakończona sukcesem."
34     echo "Uruchamiam program w trybie debugowania..."
35     gdb -q -ex "break main" -ex "run" -ex "step" -ex "help" -ex "quit" ./$OUTPUT
36 else
37     echo "Błąd podczas kompilacji."
38 fi
39
```

Sprawozdanie z zajęć: „Podstawy Programowania” z dnia 27.10.2022r.

## Laboratorium 4

**Imię i nazwisko:** Patryk Kozłowski / Informatyka Techniczna

**Temat:** Zmienne i operatory w C

**Cel:** Opanowanie tworzenia zmiennych, używania operatorów oraz ogólnej arytmetyki i działań na nich przeprowadzone.

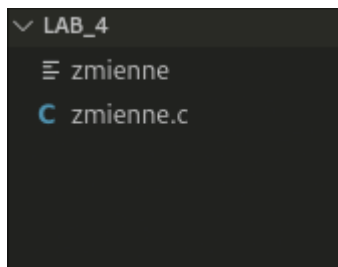
**Opis:**

1. Utworzyłem folder lab\_4



```
[umarly-poeta new-host-2] - [~/projects/pp] - [pią paź 18, 08:29]
[($)]> ls
.  .git  lab_4
[umarly-poeta new-host-2] - [~/projects/pp] - [pią paź 18, 08:29]
[($)]> mkdir lab_4
```

2. Skopiowałem zmienne.c do folderu



```
▼ LAB_4
  ≡ zmienne
  C  zmienne.c
```

3. Skompilowałem plik źródłowy i wykonałem go w terminalu

```

~({})> ./zmienne

wartość zmiennej n = 7, wartość zmiennej znak_do_testowania = a
42 10000000000000000 5.320000 5.231324 d 0

bledne uzynie
* 10000000000000000 100 5.320000 0 5.231324
n = 7, (2 * n + 3) = 17 ≠ (2 * (n + 3)) = 20

n = 7, wynik podstawienia: 'p = n++;' - 7, wynik podstawienia: 'q = ++n;' - 8

n = 1574, n/7 = 224, reszta - n%7 = 6, n = (n/7)*7 + (n%7) = 1574
4 % 3 = 1 z powrotem to 2
r = 7, stała SIEDEM = 7
piec to 5, osiem to 8
liczby zmiennoprzecinkowe:
float - f = 1.0f/3.0f = 0.333333 (dokładnie: 0.333333343267441)
float - g = 1.0f/3.0 = 0.333333 (dokładnie: 0.333333343267441)
double - d = 1.0/3.0 = 0.333333 (dokładnie: 0.333333333333333)
double - e = 1.0f/3.0 = 0.333333 (dokładnie: 0.333333333333333)
(liczba całkowita s = 1/3 = 0 - dzielenie całkowite)
(liczba całkowita t = 1.0/3.0 = 0 - obcięcie przy konwersji)

float - f = 1.0f/3.0f = 3.333333e-01 (dokładnie: 3.333333432674408e-01)
double - d = 1.0/3.0 = 3.333333e-01 (dokładnie: 3.333333333333333e-01)
normalnie: 4.213112 a dokładnie 4.213112354278564
normalnie: 2.123123 a dokładnie 2.123123123000000
Błąd dla z05: 0.000000
Błąd dla z06: 0.000000

Wartość logiczna wyrażenia: 1574<6 wynosi 0 (int a = n<m = 1)

Wartość logiczna wyrażenia: 1574<6 lub 1574>6 wynosi 1 (_Bool b = n<m || n>m = 1)
l01 1 l02 1 l03 1
Niejawne i jawne konwersje typów:
n = 2, f = 0.333333, d = 1575111.000000, e = 3.333333e-01

Precyzja stałych i niejawne konwersje:
(1.0/3.0)*3.0: JednaTrzecia → d = 1.000000000000000;
JednaTrzeciaFloat → e = 1.000000029802322

Wynik operacji (1.0/3.0) / BEZ NAWIASÓW = 0.111111
Wynik operacji 3.0 * DODAWANIE = 5.000000

WŁASNE EKSPERYMENTY:
n = 2, m = 6
o = m + n = 8: n = 2, m = 6
o = m++ + n = 8: n = 2, m = 7
o = m + ++n = 10: n = 3, m = 7
o = m++ + ++n = 13: n = 4, m = 10
o = m++ + n++ = 15: n = 5, m = 12
o = ++m + ++n = 20: n = 6, m = 14

float: fx1 = 12300000.000000000000000, fx2 = 0.000000123000000,
(fx1+fx2)-fx1 ≠ fx2 (dla float) ( (fx1+fx2)-fx1 = 0.000000000000000 )

double: dx1 = 12300000.000000000000000, dx2 = 0.000000123000000,

```

4. Analiza kodu podanego przez prowadzącego i wykonanie następujących zada

4.1. sprawdziłem różne przypadki wypisywania typów zmiennych dla wyświetlania innych typów

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - zdefiniowanie po jednej zmiennej każdego z omawianych na wykładzie typów - w tym typów
// /** z określeniami: short, long, unsigned
// /** - podstawienie do każdej zmiennej dowolnie wybranej, poprawnej wartości
// /** - zdefiniowanie połączone z nadaniem wartości (zainicjowaniem) ponownie dla jednej zmiennej
// /** każdego z omawianych na wykładzie typów
// /** - wypisanie wartości zdefiniowanych zmiennych na ekranie w oknie terminala: dobranie odpowiedniego
// /** symbolu formatowania dla każdego typu - sprawdzenie poprawności działania (czyli wydrukowania
// /** nadanej wartości)
// /** - sprawdzenie efektu błędnego użycia symbolu formatowania: kiedy wydruk przestaje być poprawny
// /** (np. zmienna typu unsigned drukowana jako int, zmienna float drukowana jako int itd., itp.)
short int d01 = 42; // mniejszy int
unsigned long long d02 = 10000000000000000; // naturalne bardzo duże
float d03 = 5.32; // zmiennoprzecinkowa
double d04 = 5.231324; // zmiennoprzecinkowa ale dokładniejsza do większej miejsc po przecinku
char d05 = 'd'; // znak ascii
int d06 = 0; // całkowite
printf("%d %llu %f %lf %c %d\n", d01, d02, d03, d04, d05, d06);

printf("bledne uzynie\n");
printf("%c %llu %d %lf %d %f", d01, d02, d03, d04, d05, d06);
```

```
42 10000000000000000 5.320000 5.231324 d 0
bledne uzynie
* 10000000000000000 100 5.320000 0 5.231324
```

#### 4.2. sprawdziłem działanie inkrementacji na różne sposoby

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - napisanie kilku (2-3) wyrażeń arytmetycznych, w których wynik zależał będzie od kolejności
// /** zapisu i priorytetów wykonywanych operacji - testowanie poprawnego użycia nawiasów
int e01 = (4 + 5 * 2) / 8;
int e02 = ((41 - 12) * 12) / (2 + 10);
// /** operatory jednoargumentowe i ich priorytety
int p = n++;
n--; // powrót do początkowej wartości n
int q = ++n;
n--; // powrót do początkowej wartości n
printf("\nn = %d, wynik podstawienia: 'p = n++;' - %d, wynik podstawienia: 'q = ++n;' - %d\n", n, p, q);
```

```
n = 7, wynik podstawienia: 'p = n++;' - 7, wynik podstawienia: 'q = ++n;' - 8
```

#### 4.3. napisałem własne wyrażenia z operatorami

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - napisanie kilku (2-3) wyrażeń arytmetycznych z operatorami jednoargumentowymi, w których wynik
// /** zależał będzie od priorytetów wykonywanych operacji - testowanie poprawnego użycia nawiasów
int z01 = (p++) + ++q;
int z02 = (q++) / (p++);
```

#### 4.4. napisanie paru wyrażeń z modulo

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - napisanie kilku (2-3) wyrażeń arytmetycznych z funkcją modulo:
// /** - sprawdzanie podzielności jednych liczb przez inne
// /** - dzielenie z resztą i uzyskiwanie pierwotnej wartości (dla innych liczb niż w przykładzie
// /** powyżej)
int z03 = 4 % 3;
printf("4 % 3 = %d ", z03);
z03 = (4 / 3) + z03;
printf("z powrotem to %d", z03);
```

**4 % 3 = 1 z powrotem to 2**

#### 4.5. napisałem kilka symboli i je wyświetliłem

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - nadanie wartości kilku symbolom - wartości powinny być stałymi różnych typów liczbowych
// /** - podstawienie do zmiennych wartości za pomocą symboli określonych w #define i wydrukowanie
// /** wartości zmiennych (UWAGA: po podstawieniu za symbol napisu stanowi czego zapis liczby,
// /** funkcjonuje on w kodzie jako liczba określonego typu - jak w przykładzie powyżej - drukuj c
// /** go należy odpowiednio dobrać formatowanie)
printf("piec to %d, osiem to %d", PIEC, OSIEM);
```

```
#define SIEDEM 7 // wystąpienia symbolu SIEDEM są w kodzie zamieniane na 7
#define OSIEM 8
#define PIEC 5
```

**piec to 5, osiem to 8**

#### 4.6. sprawdziłem dokładność zmiennej float i double

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - zdefiniowanie kilku zmiennych typów float i double oraz nadanie im wartości
// /** za pomocą stałych typu float i double
// /** - wydrukowanie wartości zdefiniowanych i zainicjowanych zmiennych z wystarczającą liczbą cyfr,
// /** tak aby zobaczyć jaka jest dokładność (jaki jest błąd wartości zmiennej w stosunku do podstawian
// /** matematycznej wartości (może to dotyczyć także prostych ułamków dziesiętnych, np. 0.1, 0.2, 0.3)
float z05 = 4.2131123123f;
double z06 = 2.123123123;
printf("normalnie: %f a dokładniej %20.15f\n", z05, z05);
printf("normalnie: %lf a dokładniej %20.15lf\n", z06, z06);

// Obliczanie błęd wartości matematycznych
float expected_z05 = 4.2131123123f;
double expected_z06 = 2.123123123;
float error_z05 = expected_z05 - z05;
double error_z06 = expected_z06 - z06;

printf("Błąd dla z05: %f\n", error_z05);
printf("Błąd dla z06: %lf\n", error_z06);
```

```
normalnie: 4.213112 a dokładniej 4.213112354278564
normalnie: 2.123123 a dokładniej 2.123123123000000
Błąd dla z05: 0.000000
Błąd dla z06: 0.000000
```

#### 4.7. sprawdziłem czy dane wyrażenia są sobie równe

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - skonstruowanie kilku złożonych wyrażeń logicznych, które będą np. sprawdzać zasady rachunku z
// /** (p lub q) i r == (p i r) lub (q i r) - czy jest to równoważne
// /** (p i q) lub r == (p lub r) i (q lub r) - czy jest to równoważne
// /** - jaka jest kolejność operacji i wynik w przypadku usuwania kolejnych nawiasów
int l01 = ((p || q) && r) == ((p && r) || (q && r)); // prawo rozdzielności koniunkcji względem alternatywy
int l02 = ((p && q) || r) == ((p || r) && (q || r)); // prawo rozdzielności alternatywy względem koniunkcji
int l03 = (p && q || r) == (p || r && q || r);
```

```
l01 1 l02 1 l03 1
```

4.8. zamieniłem parę zmiennych na różne typy i sprawdziłem jak są konwersjonowane

```
// /** !!!!!!! ***** !!!!!!!
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)
// /** - zapisanie kilku wyrażeń z operacjami na zmiennych różnych typów i zapisie do zmiennej
// /** - sprawdzenie otrzymanego wyniku: jakich konwersji dokonał kompilator?
// /** - zapisanie kilku wyrażeń z jawną konwersją typów, obserwacja otrzymanych wyników (np.
// /** dla kilku wariantów podstawienia do f: n / 3 , (double) n / 3 , n / 3.0 , (double) (n / 3)
int m01 = 4 / 3;
int m02 = 4 / e;
int m03 = (float) (4 / 3);
int m04 = (float) (4.0 / 3.0);
int m05 = (int) ((char) 97 - 'a');
```

4.9. sprawdziłem jak symbole zachowują się z i bez nawiasów

```
// /** ZADANIA WŁASNE (po odkomentowaniu powyższego kodu i sprawdzeniu poprawności jego działania)

double wynik = (1.0/3.0) / BEZ_NAWIASOW;
printf("\nWynik operacji (1.0/3.0) / BEZ_NAWIASOW = %lf\n", wynik);

double wynik2 = 3.0 * DODAWANIE;
printf("Wynik operacji 3.0 * DODAWANIE = %lf\n", wynik2);
```

```
Wynik operacji (1.0/3.0) / BEZ_NAWIASOW = 0.111111
Wynik operacji 3.0 * DODAWANIE = 5.000000
```

4.10. sprawdziłem dokładniej działania na float i double, przy okazji sprawdzając różne tolerancje danych wyrażeń oraz ich precyzję



```

// Wypisanie wartości zmiennych
printf("\nProblemy z precyzją zmiennych float i double:\n");
printf("float: fx1 = %20.15f, fx2 = %20.15f\n", fx1, fx2);
printf("double: dx1 = %20.151f, dx2 = %20.151f\n", dx1, dx2);

// Sprawdzenie precyzji operacji dodawania i odejmowania
if ((fx1 + fx2) - fx1 == fx2) {
    printf("(fx1 + fx2) - fx1 == fx2 (dla float)\n");
} else {
    printf("(fx1 + fx2) - fx1 != fx2 (dla float) ( (fx1 + fx2) - fx1 = %.15f )\n", (fx1 + fx2) - fx1);
}

if ((dx1 + dx2) - dx1 == dx2) {
    printf("(dx1 + dx2) - dx1 == dx2 (dla double)\n");
} else {
    printf("(dx1 + dx2) - dx1 != dx2 (dla double) ( (dx1 + dx2) - dx1 = %.151f )\n", (dx1 + dx2) - dx1);
}

// Próba uniknięcia dzielenia przez zero lub utraty precyzji
if (fabs((fx1 + fx2) - fx1) < TOLERANCE) {
    printf("\nPróba dzielenia przez liczbę bliską zero! Przerwanie programu!\n");
    return -1;
}

float tx3 = 1.0f / ((fx1 + fx2) - fx1); // powinno być równe 1/fx2...
printf("\n1.0f / ((fx1 + fx2) - fx1) = %20.15f\n", tx3);

double zx3 = 1.0 / ((dx1 + dx2) - dx1); // powinno być równe 1/dx2...
printf("\n1.0 / ((dx1 + dx2) - dx1) = %.151f != 1.0 / dx2 = %.151f\n", zx3, 1.0 / dx2);
printf("Zostały tylko trzy cyfry znaczące na skutek skończonej precyzji...\n");
printf("\nProgram dotarł do kołca\n");
return(0);

```

```

~({})> ./zmienne

wartość zmiennej n = 7, wartość zmiennej znak_do_testowania = a
42 10000000000000000 5.320000 5.231324 d 0

bledne uzynie
* 10000000000000000 100 5.320000 0 5.231324
n = 7, (2 * n + 3) = 17 ≠ (2 * (n + 3)) = 20

n = 7, wynik podstawienia: 'p = n++;' - 7, wynik podstawienia: 'q = ++n;' - 8

n = 1574, n/7 = 224, reszta - n%7 = 6, n = (n/7)*7 + (n%7) = 1574
4 % 3 = 1 z powrotem to 2
r = 7, stała SIEDEM = 7
piec to 5, osiem to 8
liczby zmiennoprzecinkowe:
float - f = 1.0f/3.0f = 0.333333 (dokładnie: 0.333333343267441)
float - g = 1.0f/3.0 = 0.333333 (dokładnie: 0.333333343267441)
double - d = 1.0/3.0 = 0.333333 (dokładnie: 0.333333333333333)
double - e = 1.0f/3.0 = 0.333333 (dokładnie: 0.333333333333333)
(liczba całkowita s = 1/3 = 0 - dzielenie całkowite)
(liczba całkowita t = 1.0/3.0 = 0 - obcięcie przy konwersji)

float - f = 1.0f/3.0f = 3.333333e-01 (dokładnie: 3.3333333432674408e-01)
double - d = 1.0/3.0 = 3.333333e-01 (dokładnie: 3.333333333333333e-01)
normalnie: 4.213112 a dokładnie 4.213112354278564
normalnie: 2.123123 a dokładnie 2.123123123000000
Błąd dla z05: 0.000000
Błąd dla z06: 0.000000

Wartość logiczna wyrażenia: 1574<6 wynosi 0 (int a = n<m = 1)

Wartość logiczna wyrażenia: 1574<6 lub 1574>6 wynosi 1 (_Bool b = n<m || n>m = 1)
l01 1 l02 1 l03 1
Niejawne i jawne konwersje typów:
n = 2, f = 0.333333, d = 1575111.000000, e = 3.333333e-01

Precyzja stałych i niejawne konwersje:
(1.0/3.0)*3.0: JednaTrzecia → d = 1.000000000000000;
JednaTrzeciaFloat → e = 1.000000029802322

Wynik operacji (1.0/3.0) / BEZ NAMIASOW = 0.111111
Wynik operacji 3.0 * DODAWANIE = 5.000000

WŁASNE EKSPERYMENTY:
n = 2, m = 6
o = m + n = 8: n = 2, m = 6
o = m++ + n = 8: n = 2, m = 7
o = m + ++n = 10: n = 3, m = 7
o = m++ + ++n = 13: n = 4, m = 10
o = m++ + n++ = 15: n = 5, m = 12
o = ++m + ++n = 20: n = 6, m = 14

float: fx1 = 12300000.000000000000000, fx2 = 0.0000001230000007,
(fx1+fx2)-fx1 ≠ fx2 (dla float) ( (fx1+fx2)-fx1 = 0.000000000000000 )
double: dx1 = 12300000.000000000000000, dx2 = 0.0000001230000000,

```

## 5. Podsumowanie i Wnioski

Zadania przedstawione w sprawozdaniu obejmują szeroki zakres problemów programistycznych, od prostych operacji arytmetycznych, po bardziej złożone algorytmy związane z analizą danych i tworzeniem funkcji. Każde zadanie miało na celu rozwinięcie konkretnych umiejętności, takich jak efektywne zarządzanie danymi wejściowymi, obliczenia matematyczne, a także praca z pętlami i warunkami logicznymi.

Pierwsze zadanie, dotyczące liczenia nóg zwierząt, pozwala na ćwiczenie umiejętności pracy z funkcjami i arytmetyką w kontekście rzeczywistego problemu. Zadania związane z obliczaniem lat przestępnych oraz analizą przedziałów liczb uczą stosowania instrukcji warunkowych i pętli, co jest podstawą wielu algorytmów. Natomiast stworzenie gry „kamień, papier, nożyczki” wymagało użycia funkcji

losujących, co dodatkowo rozwija umiejętności związane z interakcją komputera z użytkownikiem.

Podsumowując, wszystkie zadania były wartościowe pod względem nauki kluczowych elementów programowania, takich jak logika, struktury danych oraz interakcja z użytkownikiem. Rozwiązywanie ich pozwoliło na wzmocnienie umiejętności praktycznych i teoretycznych, niezbędnych w pracy nad bardziej zaawansowanymi projektami programistycznymi.