Programowanie w języku C

- 20 godzin (10 × 2 godziny lekcyjne)
- 6 października – 8 grudnia
- Zaliczenie na ocenę:
  - Kartkówki + prace domowe (50% oceny)
  - Kolokwium albo program zaliczeniowy (50% oceny)
- Konsultacje:
  - Piątek, po wcześniejszym umówieniu
- rozanski@fizyka.umk.pl

Ćwiczenie 1

- Dane: $a, b \in \mathbf{N}$
- Szukane: min(a, b) czyli mniejsza z dwóch

- Do dyspozycji mamy:
  - Dowolną liczbę komórek pamięci, które możemy odczytywać i nadpisywać dowolnymi wartościami
  - Operacje matematyczne + − ×,
    ÷ (całkowitoliczbowe), reszta z dzielenia

Ćwiczenie 2

- Dane: $a, b, c, d \in \mathbf{N}$
- Szukane: min(a, b, c, d) czyli mniejsza z czterech
  - Sugestia: nie ścierajmy tablicy

- Do dyspozycji mamy:
  - Dowolną liczbę komórek pamięci, które możemy odczytywać i nadpisywać dowolnymi wartościami
  - Operacje matematyczne + − ×,
    ÷ (całkowitoliczbowe), reszta z dzielenia

Programowanie proceduralne

- Kod składa się z procedur (funkcji) o ustalonym wejściu i wyjściu
- Każda procedura wykonuje dobrze zdefiniowaną operację
  - Może korzystać z innych procedur
- Procedura ma określony poziom abstrakcji

… określony poziom abstrakcji

- ŹLE:

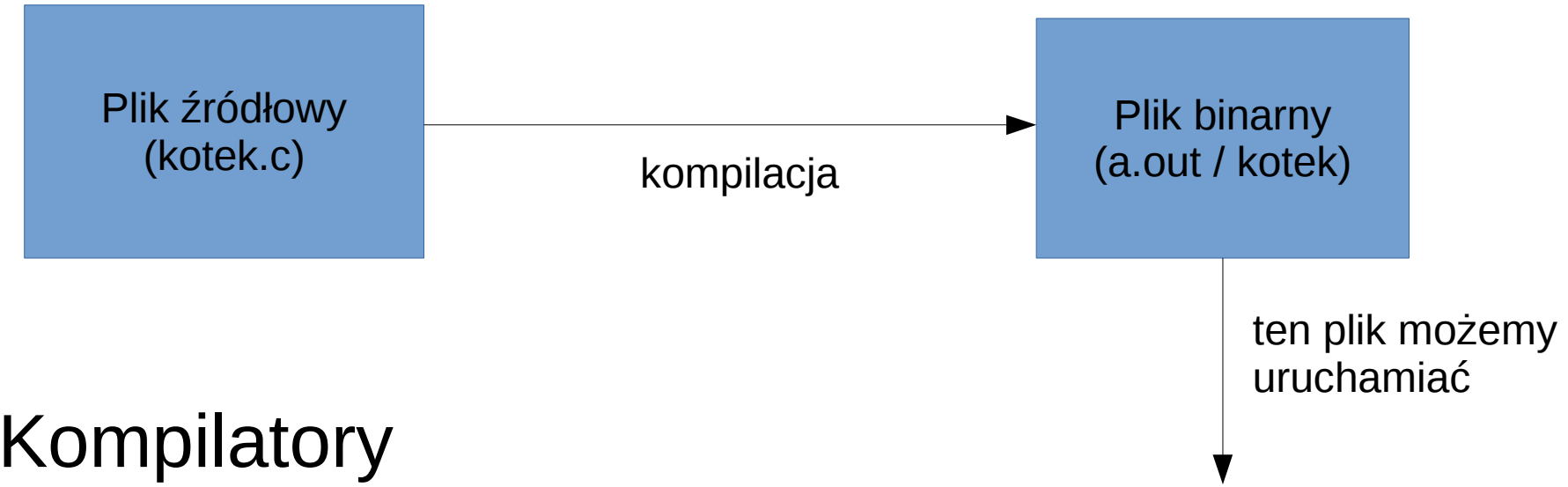
```
MójProgram:
Zapytaj użytkownika o liczby a, b
Jeśli a < b, to zamień a z b
Dopóki b > 0 {
    a := reszta z dzielenia a przez b
    Zamień a z b
}
Wypisz a na ekran
```

- DOBRZE:

```
MójProgram:
Zapytaj użytkownika o liczby x, y
nwd := NWD(x, y)
Wypisz nwd na ekran

NWD(a,b):
Jeśli a < b, to zamień a z b
Dopóki b > 0 {
    a := reszta z dzielenia a przez b
    Zamień a z b
}
wynik := wartość a
```

Język C

- Standard C99
- Język kompilowany



- Kompilatory
  - Linux: gcc, clang, icc
  - Windows: gcc (MinGW), Microsoft C, Borland C

Prosty plik źródłowy C

```
int dodaj(int x, int y)
{
  int wynik = x + y;
  return wynik;
}

int main(void)
{
  int a, b, c;
  a = 2;
  b = 2;
  c = dodaj(a, b);
}
```

Prosty plik źródłowy C

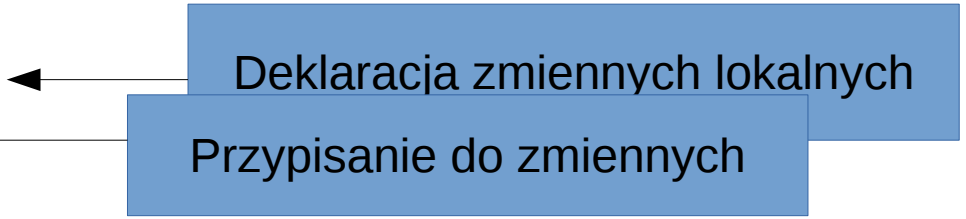
```
int dodaj(int x, int y)
{
  int wynik = x + y;
  return wynik;
}

int main(void)
{
  int a, b, c;
  a = 2;
  b = 2;
  c = dodaj(a, b);
}
```

Prosty plik źródłowy C

```
int dodaj(int x, int y)
{
  int wynik = x + y;
  return wynik;
}

int main(void)
{
  int a, b, c;
  a = 2;
  b = 2;
  c = dodaj(a, b);
}
```

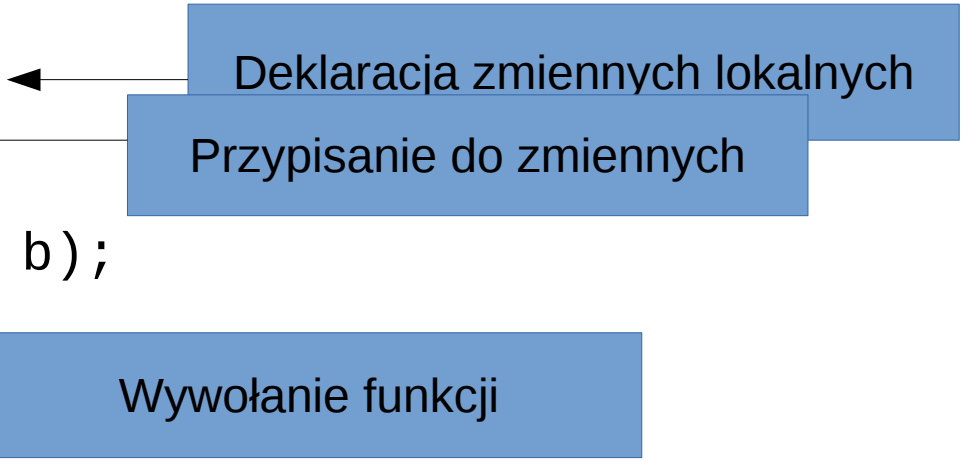
Prosty plik źródłowy C

```
int dodaj(int x, int y)
{
  int wynik = x + y;
  return wynik;
}

int main(void)
{
  int a, b, c;
  a = 2;
  b = 2;
  c = dodaj(a, b);
}
```



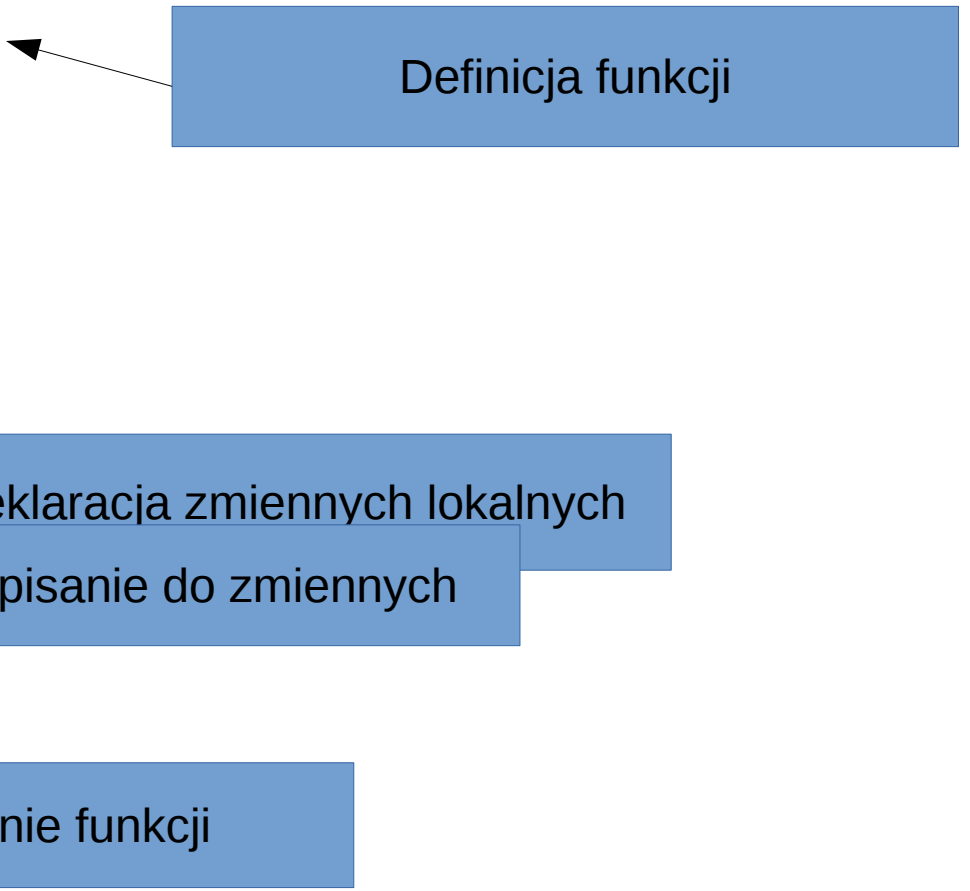
Prosty plik źródłowy C

```
int dodaj(int x, int y)
{
  int wynik = x + y;
  return wynik;
}

int main(void)
{
  int a, b, c;
  a = 2;
  b = 2;
  c = dodaj(a, b);
}
```



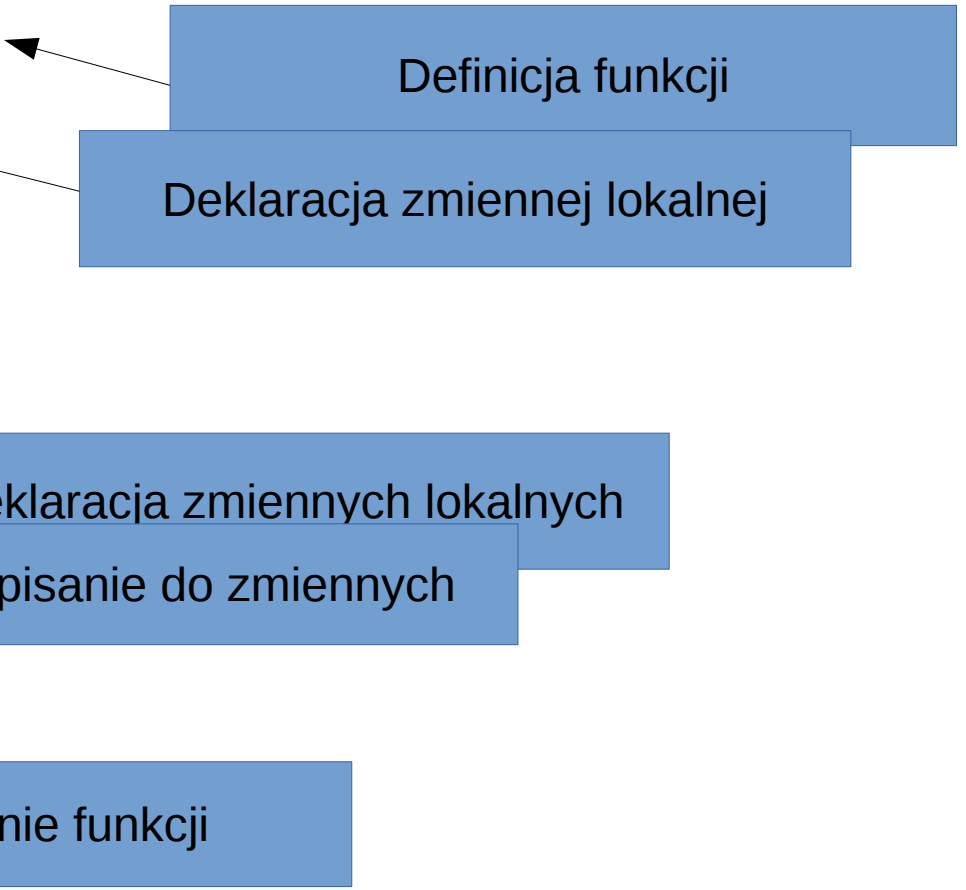
Prosty plik źródłowy C

```
int dodaj(int x, int y)
{
  int wynik = x + y;
  return wynik;
}

int main(void)
{
  int a, b, c;
  a = 2;
  b = 2;
  c = dodaj(a, b);
}
```

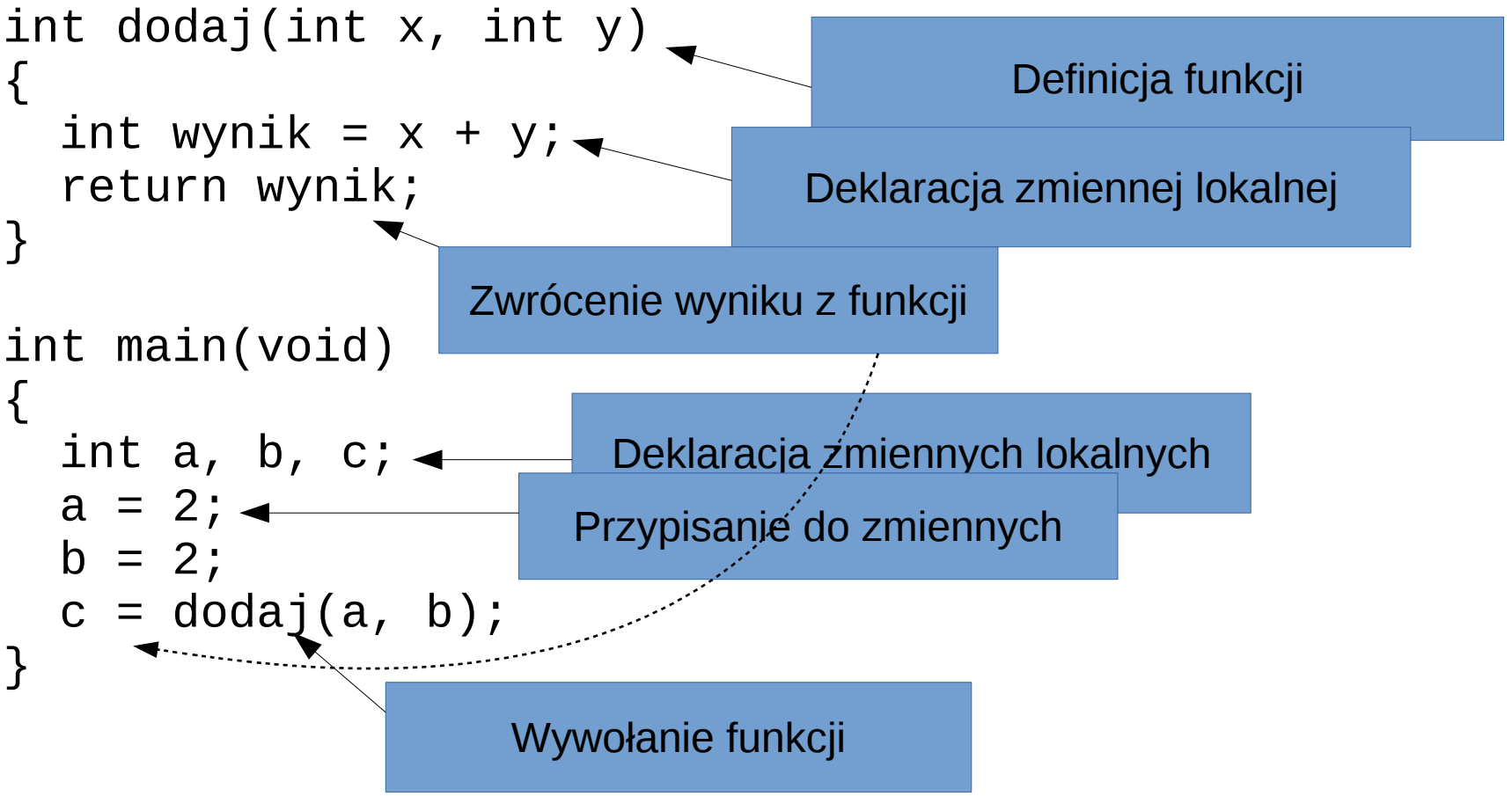


Prosty plik źródłowy C

```
int dodaj(int x, int y)
{
  int wynik = x + y;
  return wynik;
}

int main(void)
{
  int a, b, c;
  a = 2;
  b = 2;
  c = dodaj(a, b);
}
```



Prosty plik źródłowy C



Proste typy danych w C

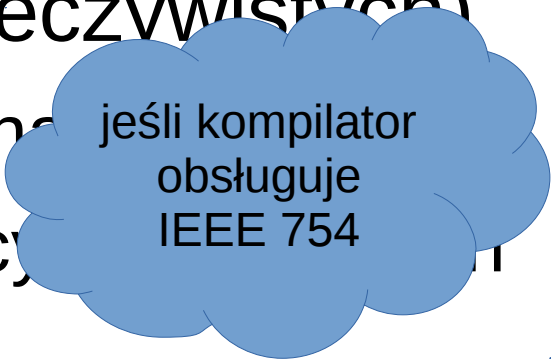
- Całkowitoliczbowe (dla liczb całkowitych):
  - char: 8 bitów = 1 bajt
    - signed char: −128…+127, unsigned char: 0…255
  - short: ≥ 16 bitów = 2 bajty
    - short int: −32768…32767, unsigned short int: 0…65535
  - int: ≥ 16 bitów = 2 bajty, ale zazwyczaj 4
    - int unsigned int
  - long ≥ 32 bity = 4 bajty, ale zazwyczaj 8
    - long int unsigned long int
  - long long ≥ 64 bity = 8 bajtów

Proste typy danych w C

- Zmiennoprzecinkowe (dla liczb rzeczywistych)
  - float: 32 bity, dokładność 7–8 cyfr znaczących
  - double: 64 bity, dokładność 15–16 cyfr znaczących

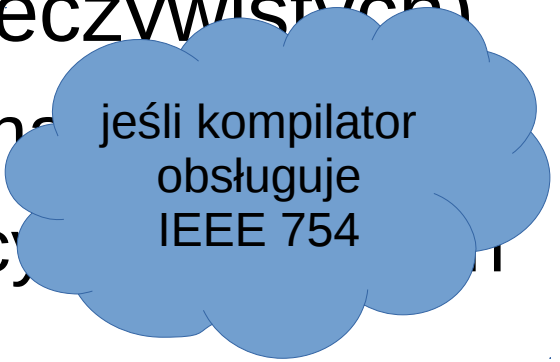
Proste typy danych w C

- Zmiennoprzecinkowe (dla liczb rzeczywistych)
  - float: 32 bity, dokładność 7–8 cyfr zn
  - double: 64 bity, dokładność 15–16 cy



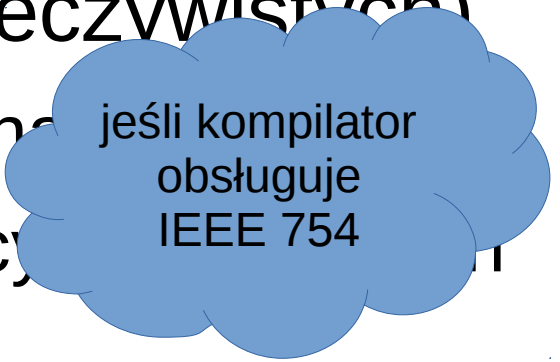
Proste typy danych w C

- Zmiennoprzecinkowe (dla liczb rzeczywistych)
  - float: 32 bity, dokładność 7–8 cyfr zn
  - double: 64 bity, dokładność 15–16 cy
  - long double: 80 lub 128 bitów



Proste typy danych w C

- Zmiennoprzecinkowe (dla liczb rzeczywistych)
  - float: 32 bity, dokładność 7–8 cyfr zn
  - double: 64 bity, dokładność 15–16 cy
  - long double: 80 lub 128 bitów



- Wartość logiczna
  - bool: false lub true → przechowywane jako int !

    (wymagane `#include <stdbool.h>` na początku pliku,
    w przeciwnym razie typ dostępny jest tylko jako _Bool)

Komentarze

- Krótki komentarz (jednolinijkowy)

```
int a = 3; // możemy połączyć deklarację z inicjalizacją
```

- Dłuższy komentarz (wielolinijkowy)

```
int a, b, c = 3; /* takie połączenie może casem mylić,
                    bo w powyższej notacji
                    tylko c otrzyma wartość 3 */
```

Podstawowe operatory w C

- Działania arytmetyczne + – * / %
- Porównanie == != < > <= >=
  - operatory porównania zwracają 1 (true) lub 0 (false)
- Operatory rzutowania

```
int a = 15;
double x = 100 / a;  // x ma wartość 6
double x = 100 / (double) a;  // x ma wartość 6.(6)
```

Podstawowe operatory w C

- Przypisanie = (ale także += −= *= /= %=)

```
int a = 2;
a += 8;  // a jest równe 10
```

- Inkrementacja i dekrementacja ++ −−

```
int a = 2;
int b = a++;  // b jest równe 2, a jest równe 3
int c = ++a;  // c jest równe 4, a jest równe 4
```

Kompilacja (Linux)

- Wersja bezpośrednia
  - gcc mójplik.c
    - Wygeneruje plik wykonywalny „a.out” z pliku „mójplik.c”
  - gcc -o mójplik mójplik.c
    - Wygeneruje plik wykonywalny „mójplik” z pliku „mójplik.c”
- Wersja z „make”
  - make mójplik
    - Wygeneruje plik wykonywalny „mójplik” z pliku „mójplik.c”
- Uruchomienie programu (w terminalu):
  - ./mójplik

Kompilacja (Linux)

- Zalecane flagi: -Wall -Wextra -O2
- Wersja bezpośrednia
  - gcc -Wall -Wextra -O2 -o mójplik mójplik.c
    - Wygeneruje plik wykonywalny „mójplik” z pliku „mójplik.c”
- Wersja z „make”
  - Tworzymy plik „Makefile” w którym piszemy

    ```
    CFLAGS=-Wall -Wextra -O2
    ```

  - make mójplik
    - Wygeneruje plik wykonywalny „mójplik” z pliku „mójplik.c”

Ćwiczenie 3

- Dane: $n \in \mathbf{N}$
- Szukane: wartość silni $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$

- Do dyspozycji mamy:
  - Dowolną liczbę komórek pamięci, które możemy odczytywać i nadpisywać dowolnymi wartościami
  - Operacje matematyczne + − ×,
    ÷ (całkowitoliczbowe), reszta z dzielenia

Instrukcja warunkowa

```
if (warunek) {
  // ten kod wykona się tylko jeśli warunek ≠ 0
} else {
  // ten kod wykona się w przeciwnym wypadku
}
```

Instrukcja warunkowa

```
if (warunek) {
  // ten kod wykona się tylko jeśli warunek ≠ 0
} else {
  // ten kod wykona się w przeciwnym wypadku
}
```

- Przykład:

```
int a = 5;
if (a <= 0) {
  puts(„To nie jest liczba dodatnia.”);
} else {
  if (a % 2 == 0) {
    puts(„To jest dodatnia liczba parzysta.”);
  } else {
    puts(„To jest dodatnia liczba nieparzysta.”);
  }
}
```

Instrukcja warunkowa

```
if (warunek) {
  // ten kod wykona się tylko jeśli warunek ≠ 0
} else {
  // ten kod wykona się w przeciwnym wypadku
}
```

- Przykład:

```
int a = 5;
if (a <= 0) {
  puts(„To nie jest liczba dodatnia.”);
} else if (a % 2 == 0) {
  puts(„To jest dodatnia liczba parzysta.”);
} else {
  puts(„To jest dodatnia liczba nieparzysta.”);
}
```

Instrukcja warunkowa

```
if (warunek) {
  // ten kod wykona się tylko jeśli warunek ≠ 0
} else {
  // ten kod wykona się w przeciwnym wypadku
}
```

- Przykład:

```
int a = 5;
if (a <= 0)
  puts(„To nie jest liczba dodatnia.”);
else if (a % 2 == 0)
  puts(„To jest dodatnia liczba parzysta.”);
else
  puts(„To jest dodatnia liczba nieparzysta.”);
```

Instrukcja warunkowa

```
if (warunek) {
  // ten kod wykona się tylko jeśli warunek ≠ 0
}
```

- Operatory logiczne:
  - `a && b` daje 1 ⇔ a ≠ 0 i b ≠ 0
  - `a || b` daje 1 ⇔ a ≠ 0 lub b ≠ 0
  - `!a` daje 1 gdy a = 0, zaś 0 gdy a ≠ 0

Instrukcja warunkowa

```
if (warunek) {
  // ten kod wykona się tylko jeśli warunek ≠ 0
}
```

- Operatory logiczne:
  - a && b daje 1 ⇔ a ≠ 0 i b ≠ 0
  - a || b daje 1 ⇔ a ≠ 0 lub b ≠ 0
  - !a daje 1 gdy a = 0, zaś 0 gdy a ≠ 0
- Przykład:

```
int a = 5;
if (a > 0 && a % 2 == 0) {
  puts(„Tak, to jest dodatnia liczba parzysta.”);
}
```

Ćwiczenie 4

- Dane: $a, b \in \mathbf{N} \quad (a \geq b)$
- Szukane: największy wspólny dzielnik *a* i *b*
  - Sugestia: algorytm Euklidesa

- Do dyspozycji mamy:
  - Dowolną liczbę komórek pamięci, które możemy odczytywać i nadpisywać dowolnymi wartościami
  - Operacje matematyczne + − ×,
    ÷ (całkowitoliczbowe), reszta z dzielenia

Ćwiczenie 5

- Dane: $a, b \in \mathbf{N} \quad (a \geq b)$
- Szukane: największa wspólna wielokrotność *a* i *b*
  - Sugestia: nie ścierajmy tablicy

- Do dyspozycji mamy:
  - Dowolną liczbę komórek pamięci, które możemy odczytywać i nadpisywać dowolnymi wartościami
  - Operacje matematyczne + − ×,
    ÷ (całkowitoliczbowe), reszta z dzielenia

Pętle w języku C

- while
- do…while
- for

Pętla while

```
while (wyrażenie) {
  // zawartość pętli
}
```

1. Sprawdź czy wyrażenie = 0.
   Jeśli tak, zakończ (wyjdź z pętli).
2. Wykonaj zawartość pętli.
3. Wróć do kroku 1.

Pętla do…while

```
do {
  // zawartość pętli
} while (wyrażenie)
```

1. Wykonaj zawartość pętli.
2. Sprawdź czy wyrażenie = 0.
   Jeśli tak, zakończ (wyjdź z pętli).
3. Wróć do kroku 1.

Pętla for

```
for (instrukcjaA; wyrażenieB; instrukcjaC) {
  // zawartość pętli
}
```

1. Wykonaj instrukcjęA.
2. Sprawdź czy wyrażenieB = 0.
   Jeśli tak, zakończ (wyjdź z pętli).
3. Wykonaj zawartość pętli.
4. Wykonaj instrukcjęC.
5. Wróć do kroku 2.

Pętla for

```
for (int i=0; i<10; ++i) {
  // zawartość pętli
}
```

1. Wykonaj instrukcję `int i = 0`.
2. Sprawdź czy `i ≥ 10`.
   Jeśli tak, zakończ (wyjdź z pętli).
3. Wykonaj zawartość pętli.
4. Wykonaj instrukcję `++i`.
5. Wróć do kroku 2.