



Programowanie I

Rafał NOWAK

Plan zajęć :

- Jak działa komputer? Wstęp do algorytmów
- Zapis liczb, typy zmiennych
- Arytmetyka binarna i działania na bitach
- Konstrukcje iteracyjne
- Złożone typy danych
- Wskaźniki
- Modularyzacja programu – funkcje, podział na pliki
- Łańcuchy tekstowe
- Operacje na plikach
- Dynamiczna alokacja pamięci
- Zaawansowana reprezentacja danych
- Algorytmy, złożoność obliczeniowa
- Elementy języka C++
- Wybrane algorytmy

Czym jest programowanie?

Programowanie komputerów – proces projektowania, tworzenia, testowania i utrzymywania kodu źródłowego programów komputerowych lub urządzeń mikroprocesorowych (mikrokontrolery).

Programowanie wymaga wiedzy i doświadczenia w wielu różnych dziedzinach, takich jak:

- projektowanie aplikacji,
- algorytmika,
- struktury danych,
- języki programowania i narzędzia programistyczne,
- kompilatory,
- sposób działania podzespołów komputera.

Jak działa komputer?

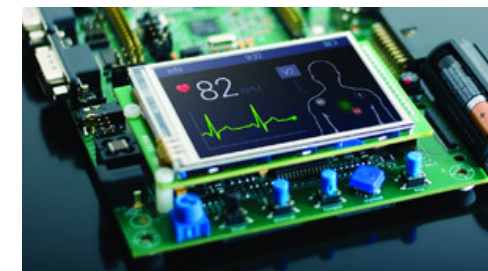
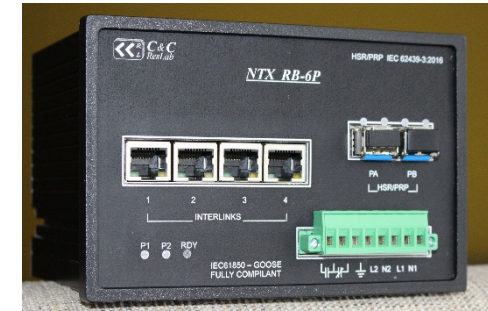
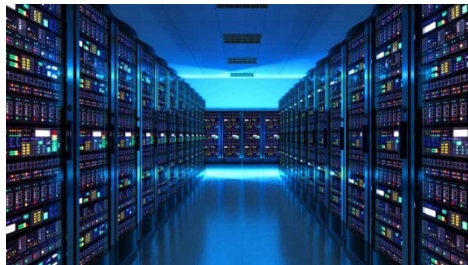
Wstęp do algorytmów

Jak działa komputer? Wstęp do algorytmów

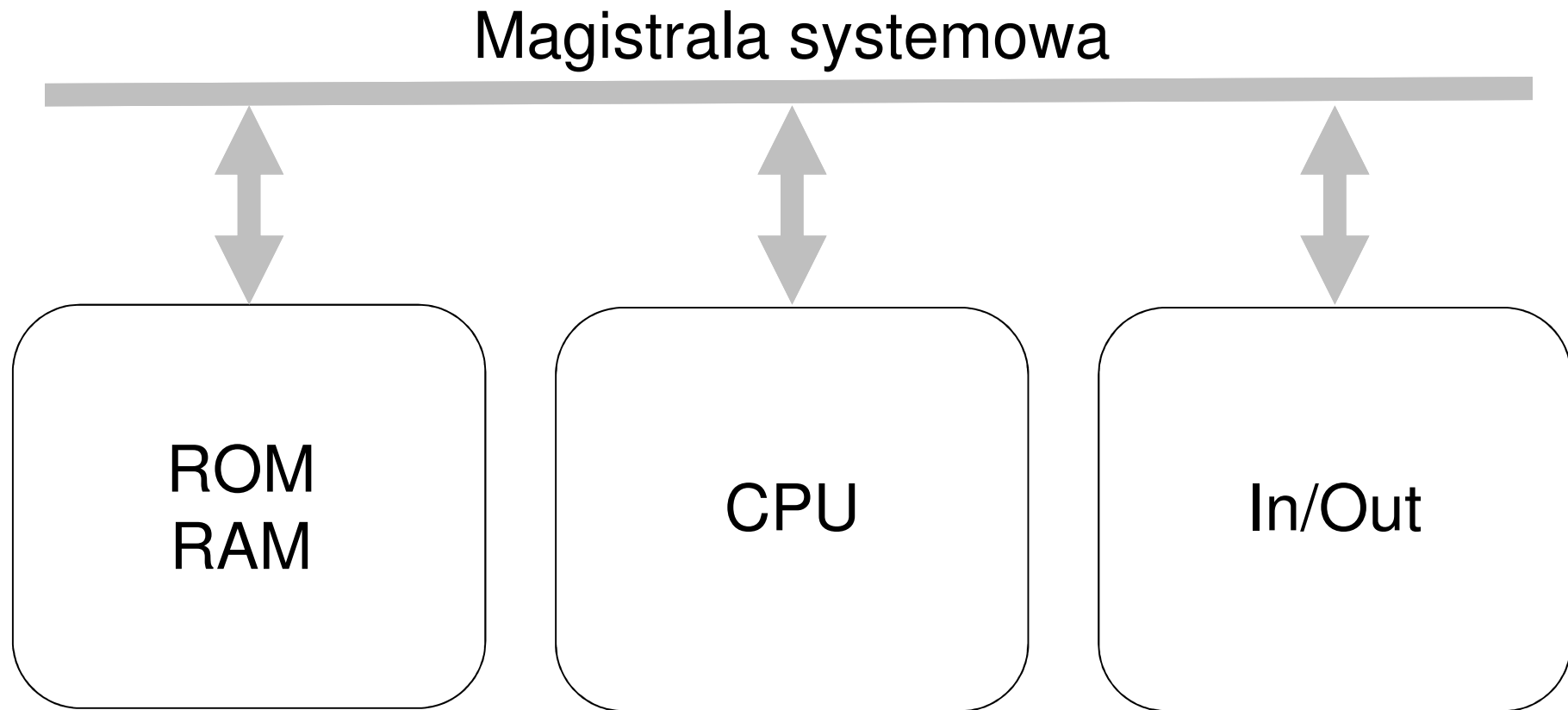
Plan zajęć :

- Podstawowa architektura komputerów
- Czym jest algorytm?
- Rodzaje algorytmów
- Języki programowania
- Proces tworzenia programu
- Prosty przykład programu w języku C

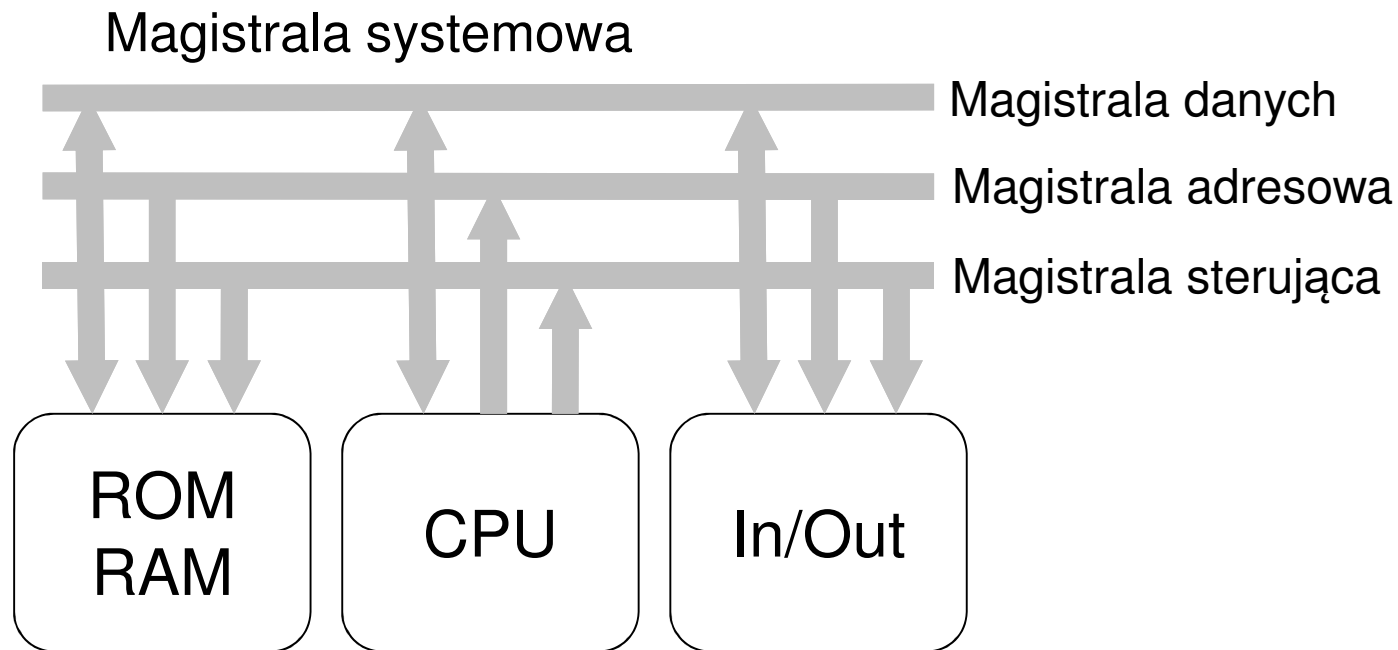
Czym jest komputer?



Podstawowa architektura komputera



Podstawowa architektura komputera



Pamięć **ROM** zawiera rozkazy (programy) określające jak przetwarzać dane

Pamięć **RAM** zawiera dane

CPU przetwarza dane

In/Out – urządzenia wejścia wyjścia służą do „komunikacji” ze „światem zewnętrznym”

CPU

Procesor, **CPU** (ang. central processing unit) – sekwencyjne urządzenie cyfrowe, które pobiera dane z pamięci operacyjnej, interpretuje je i wykonuje jako rozkazy. Procesory wykonują ciągi prostych operacji matematyczno-logicznych ze zbioru operacji podstawowych, określonych zazwyczaj przez producenta procesora jako lista rozkazów procesora.

Do typowych rozkazów wykonywanych przez procesor należą:

- kopiowanie danych
- działania arytmetyczne
- działania na bitach
- skoki

Algorytm

Algorytm to przepis na rozwiązanie określonego problemu za pomocą prostych czynności wykonywanych w ściśle określonej kolejności.

Czynności:

- muszą być znacznie prostsze od realizowanego algorytmu
- muszą być wykonywane dla danego „sprzętu”
(prostota czynności jest sprawą względną)

Kolejność:

- określenie kolejności wykonywania czynności jest krytyczne dla osiągnięcia celu algorytmu
- musi istnieć mechanizm rozgałęziania algorytmu, tj. decydowania o kolejności w trakcie wykonywania algorytmu na podstawie zaistniałych warunków

Typy i właściwości algorytmów

Można wyróżnić algorytmy:

- liniowe
- rozgałęzione (warunkowe)
- iteracyjne
- rekurencyjne

Algorytm musi:

- być opisany w sposób jednoznaczny, dający możliwość jednoznacznego rozwiązania zadania (dla tych samych danych wejściowych i stanu układu musi dawać taki sam wynik)
- dawać rozwiązanie w skończonym i akceptowalnym czasie
- mieć wyróżniony punkt startowy i końcowy

Sposoby opisu algorytmu

- język naturalny (np. język angielski)
- opis graficzny (schemat blokowy)
- pseudokod lub język programowania

Schemat blokowy

Za pomocą schematu blokowego możliwe jest zapisanie każdego poprawnego algorytmu.

Każdy algorytm można zapisać za pomocą wielu różnych schematów blokowych.

Zapis algorytmu za pomocą schematu blokowego cechuje **zwięzłość, czytelność i wysoki poziom abstrakcji**.

- elementarne czynności oznaczone są blokami (węzły sieci), a kolejność wyznaczona jest poprzez gałęzie sieci łączące węzły
- kształt bloków odpowiada rodzajowi operacji, a strzałki gałęzi identyfikują jednoznacznie ich kolejność
- struktura algorytmu jest niezależna od architektury konkretnej maszyny i rodzaju kodowania liczb

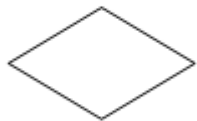
Podstawowe symbole graficzne opisu algorytmu



Proces, operacja przetwarzania, blok obliczeniowy



Proces predefiniowany, procedura, funkcja



Proces decyzyjny, rozgałęzienie



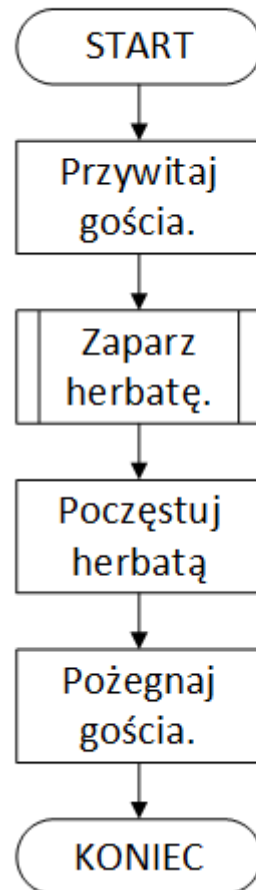
Blok danych wejścia-wyjścia



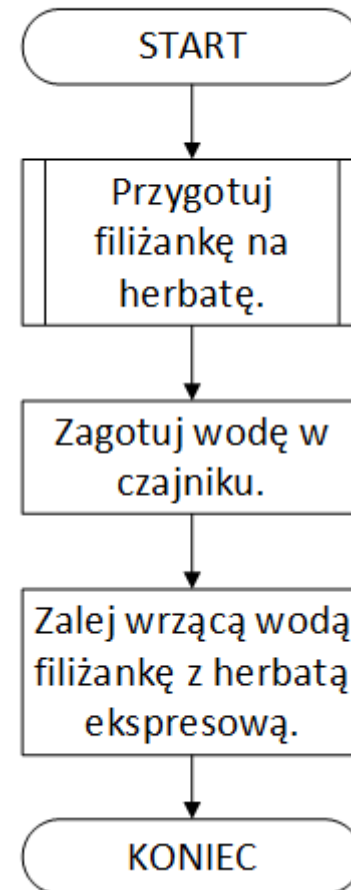
Terminator, punkt graniczny programu

Przykład algorytmu liniowego

Przyjęcie gościa

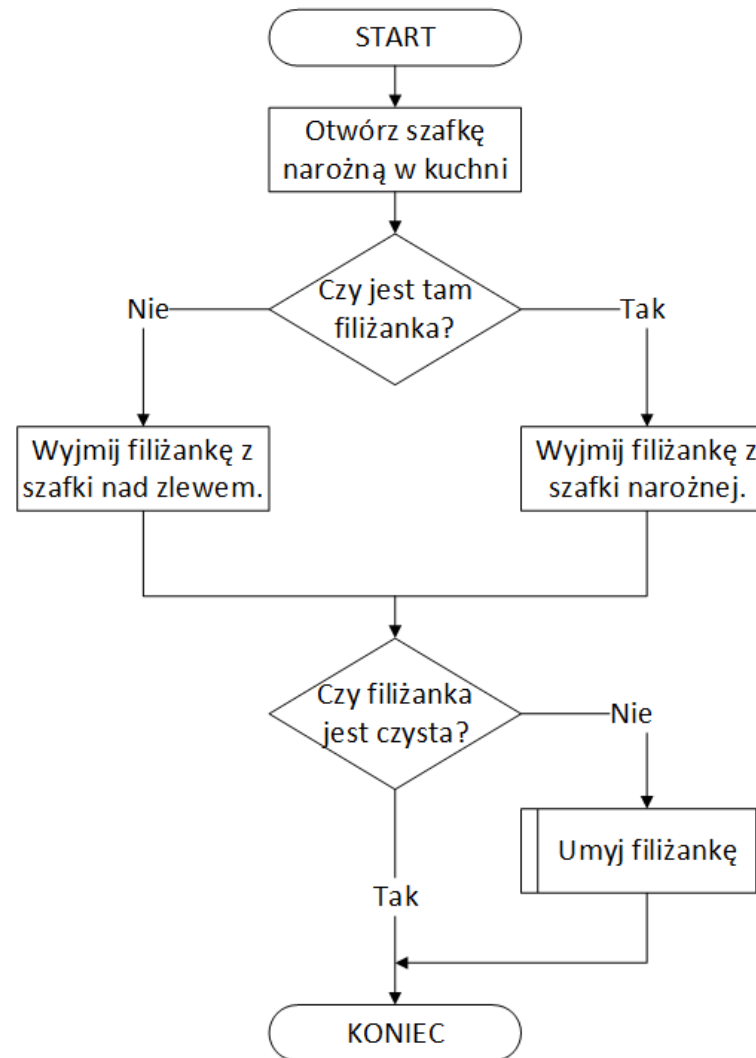


Zaparczenie herbaty

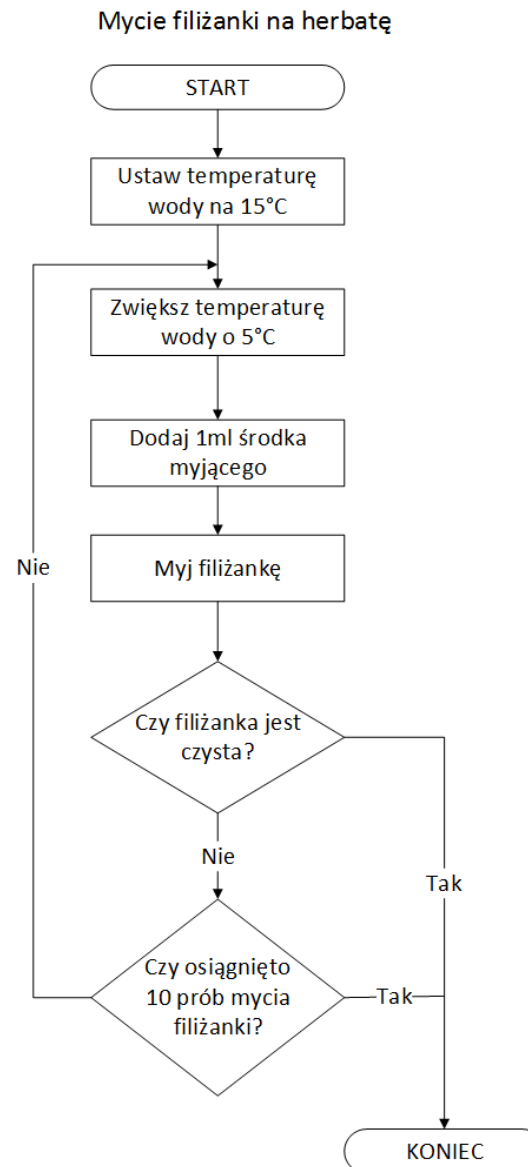


Przykład algorytmu rozgałęzionego

Przygotowanie filiżanki na herbatę

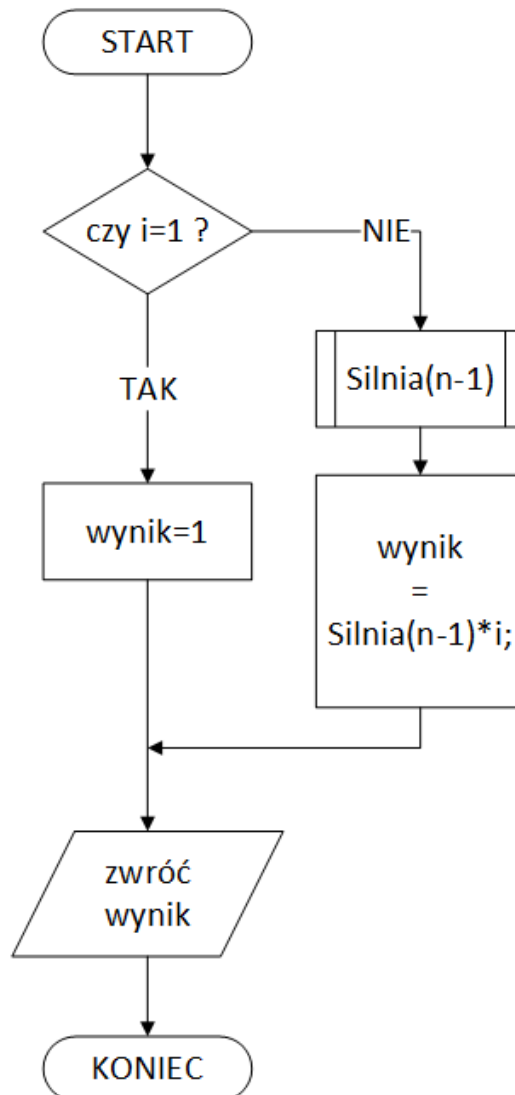


Przykład algorytmu iteracyjnego

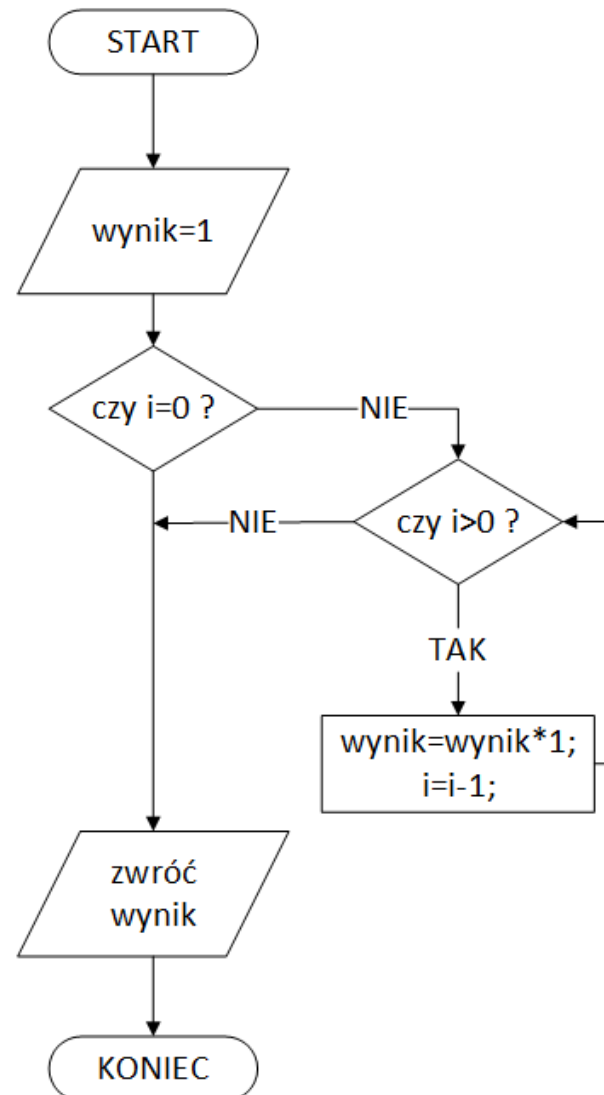


Przykład algorytmu rekurencyjnego

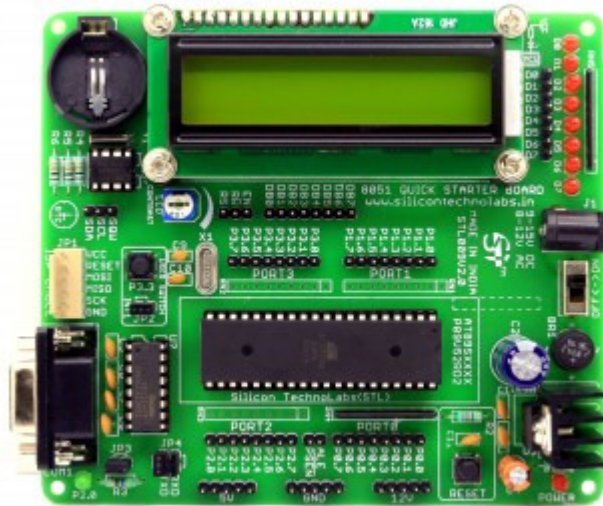
Algorytm rekurencyjny



Algorytm iteracyjny

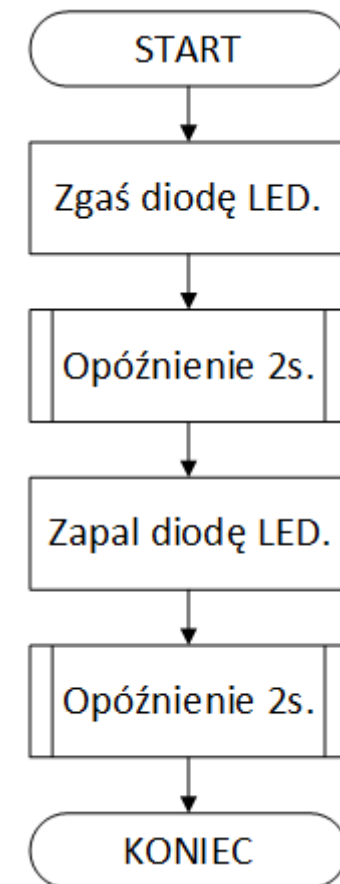


Język który rozumie komputer – kod maszynowy



```

00000000: 02 00 73 8F 82 8E 83 C3|E5 82 94 01 E5 83 94 00
00000010: 40 36 E4 F5 0A E4 F5 08|F5 09 AE 08 AF 09 E4 FC
00000020: FD 7B 20 7A CB F9 F8 D3|12 00 49 40 0A 05 09 E5
00000030: 09 70 E7 05 08 80 E3 05|0A E5 0A B4 03 D7 E5 82
00000040: 15 82 70 C3 15 83 80 BF|22 EB 9F F5 F0 EA 9E 42
00000050: F0 E9 9D 42 F0 EC 64 80|C8 64 80 98 45 F0 22 D2
00000060: 80 7F 02 7E 00 12 00 03|C2 80 7F 02 7E 00 12 00
00000070: 03 80 EC 78 7F E4 F6 D8|FD 75 81 0A 02 00 5F
  
```



Asembler

```

1          ; FUNCTION _Delay (BEGIN)
2                                     ; SOURCE LINE # 11
3      ;---- Variable 'time' assigned to Register 'DPTR' ----
4      0000 8F82          MOV      DPL,R7
5      0002 8E83          MOV      DPH,R6
6                                     ; SOURCE LINE # 12
7      0004      ?C0001:
8                                     ; SOURCE LINE # 16
9      0004 C3           CLR      C
10     0005 E582          MOV      A,DPL
11     0007 9401          SUBB     A,#01H
12     0009 E583          MOV      A,DPH
13     000B 9400          SUBB     A,#00H
14     000D 4036          JC       ?C0009
15                                     ; SOURCE LINE # 17
16                                     ; SOURCE LINE # 18
17     000F E4           CLR      A
18     0010 F500          R        MOV      i,A
19     0012      ?C0003:
20                                     ; SOURCE LINE # 19
21     0012 E4           CLR      A
22     0013 F500          R        MOV      j,A
23     0015 F500          R        MOV      j+01H,A
24     0017      ?C0006:
25     0017 AE00          R        MOV      R6,j
26     0019 AF00          R        MOV      R7,j+01H
27     001B E4           CLR      A
28     001C FC           MOV      R4,A
29     001D FD
30     001E 7B20
31     0020 7ACB

```

Język C

```
1  #include <reg51.h>          //dołączenie definicji rejestrów
2                               //mikrokontrolera
3  sbit PortLED = P0^0;        //definicja portu diody LED
4
5
6  //opóźnienie około 1 sekundy dla kwarcu 3,6864 MHz
7  void Delay(unsigned int time)
8  {
9      unsigned int j;
10     unsigned char i;
11
12     while (time >= 1)        //wykonanie pętli FOR zajmuje około 1 sek.
13     {                        //pętla jest powtarzana TIME razy
14         for (i=0; i<3; i++)
15             for (j=0; j<52000; j++);
16         time--;
17     }
18 }
19
20 //początek programu głównego
21 void main(void)
22 {
23     while (1)                //pętla nieskończona
24     {
25         PortLED = 1;          //zgaszenie diody LED
26         Delay(2);             //opóźnienie około 2 sek.
27         PortLED = 0;          //zaświecenie LED
28         Delay(2);             //opóźnienie około 2 sek.
29     }
30 }
31
```

Języki programowania

Niskiego poziomu

Wysokiego poziomu

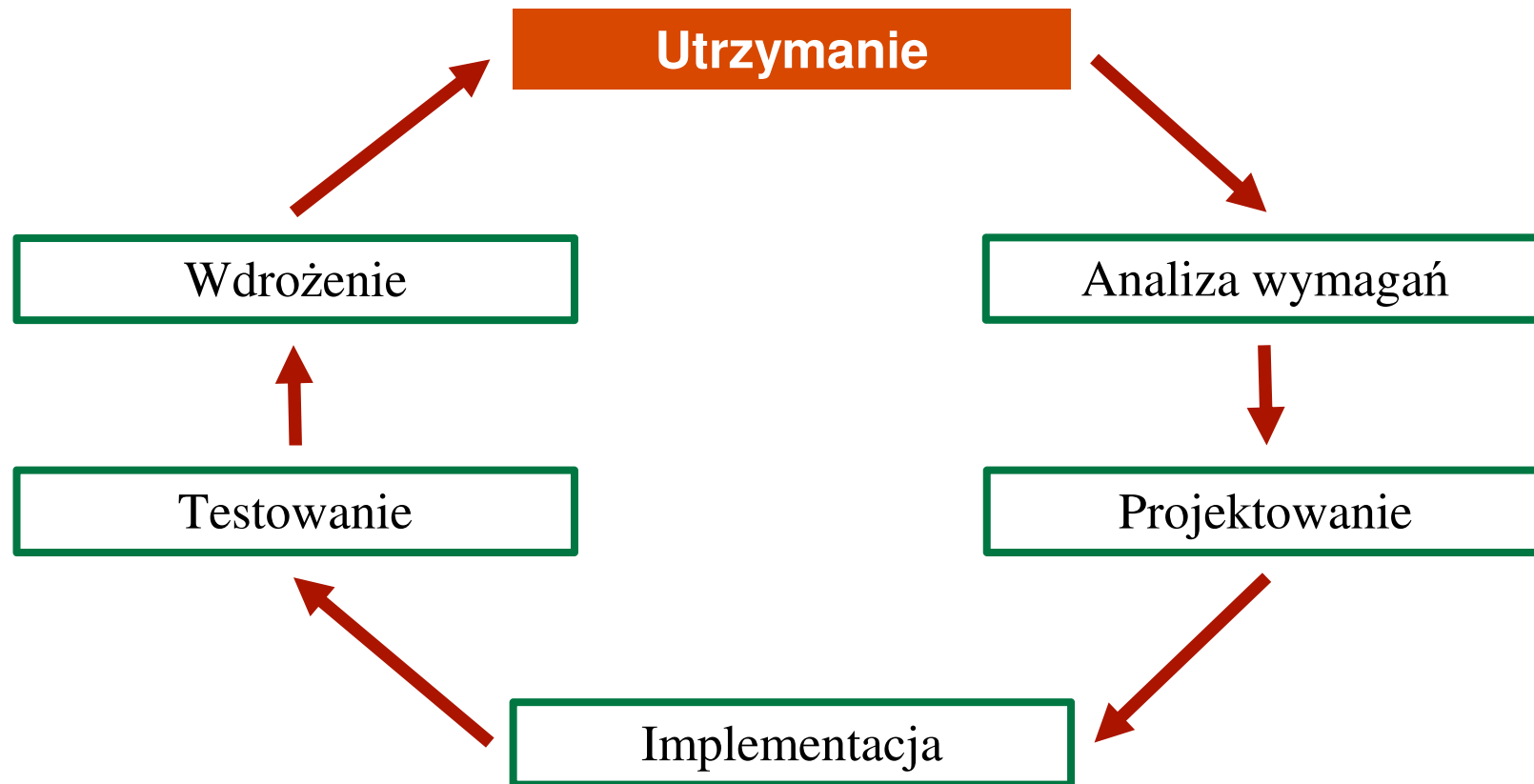
Kompilowane do kodu natywnego (np. C, C++)

Kompilowane do pseudokodu (np. Java)

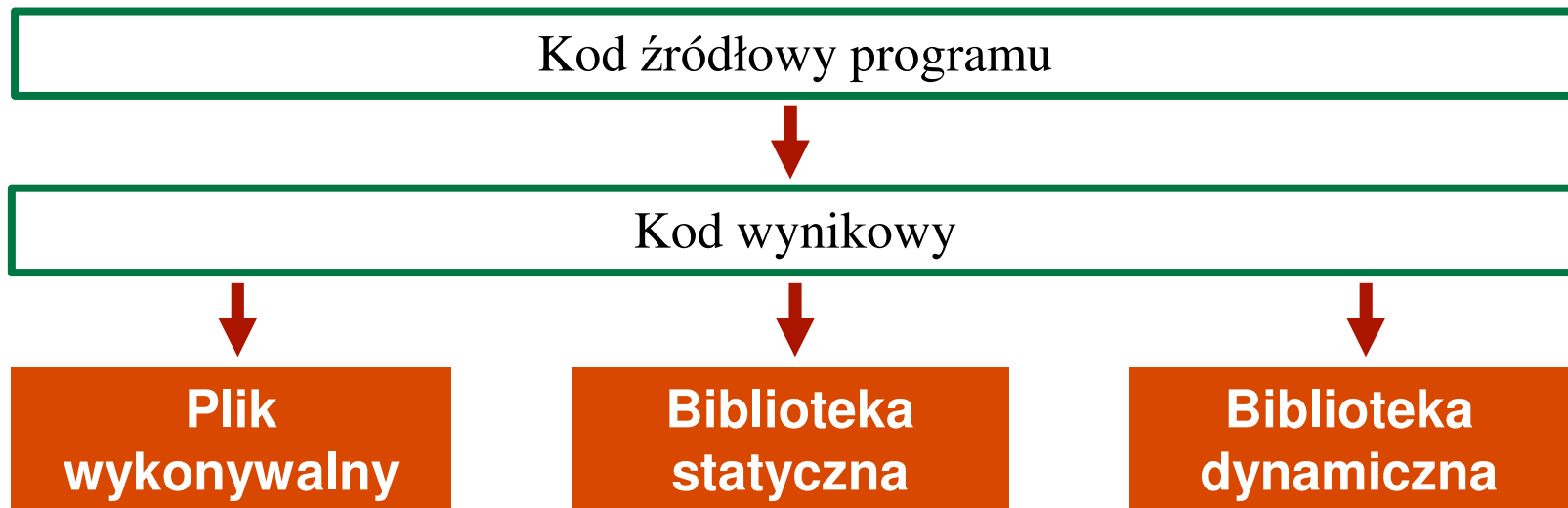
Interpretowane (np. Python, Bash)

SDLC (ang. systems development life cycle)

Cykl życia systemu informatycznego



Proces tworzenia programu

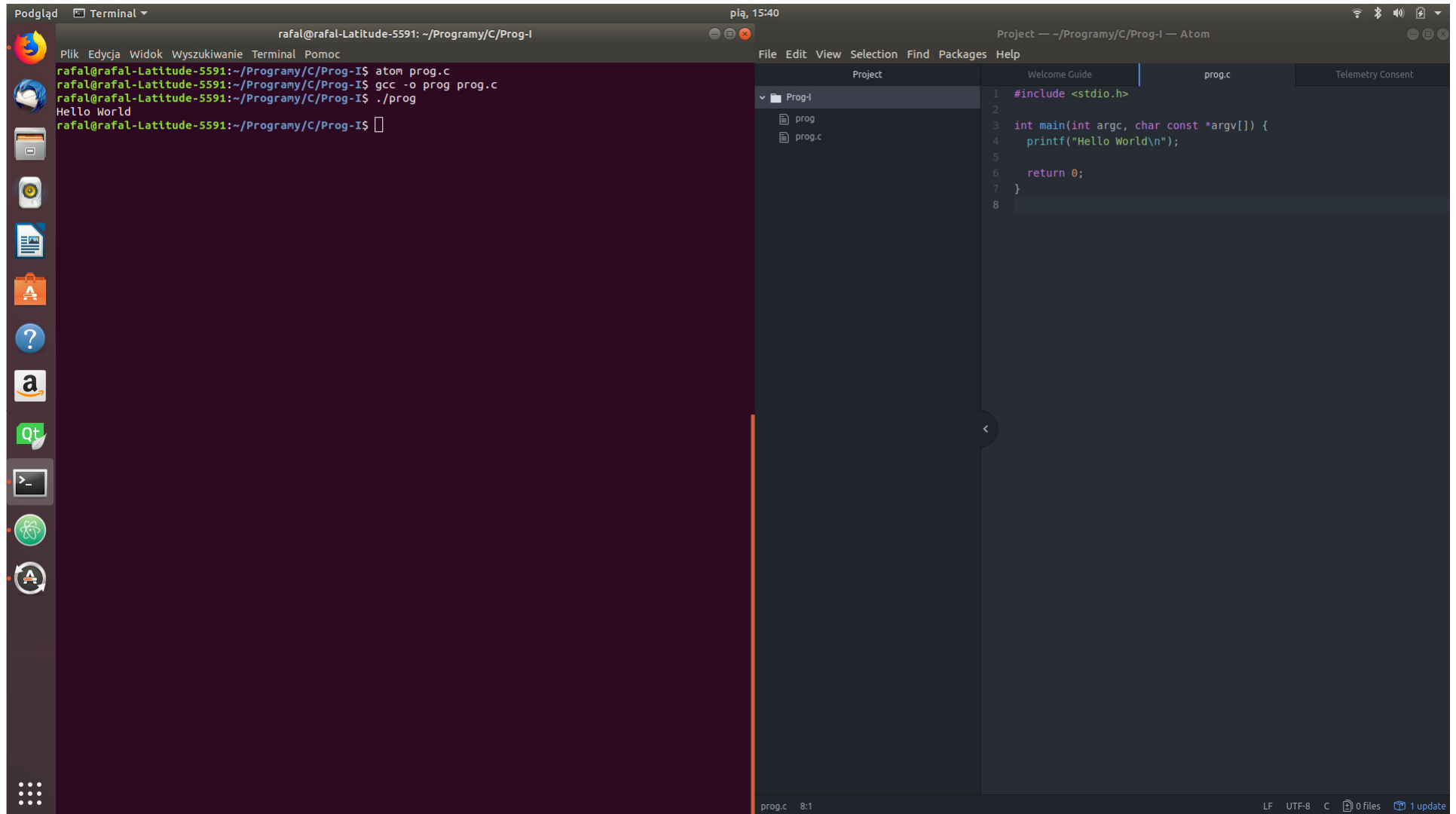


Pierwszy program

```
1  #include <stdio.h>
2
3  int main(int argc, char const *argv[]) {
4      printf("Hello World\n");
5
6      return 0;
7  }
```



Pierwszy program



The screenshot displays a Linux desktop environment. On the left, a vertical dock contains various application icons. The main workspace is divided into two windows. The left window is a terminal titled 'rafal@rafal-Latitude-5591: ~/Programy/C/Prog-I'. It shows the following commands and output:

```
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ atom prog.c
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ gcc -o prog prog.c
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ ./prog
Hello World
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$
```

The right window is the Atom code editor, titled 'Project -- ~/Programy/C/Prog-I -- Atom'. It shows the source code for 'prog.c':

```
1 #include <stdio.h>
2
3 int main(int argc, char const *argv[]) {
4     printf("Hello World\n");
5
6     return 0;
7 }
8
```

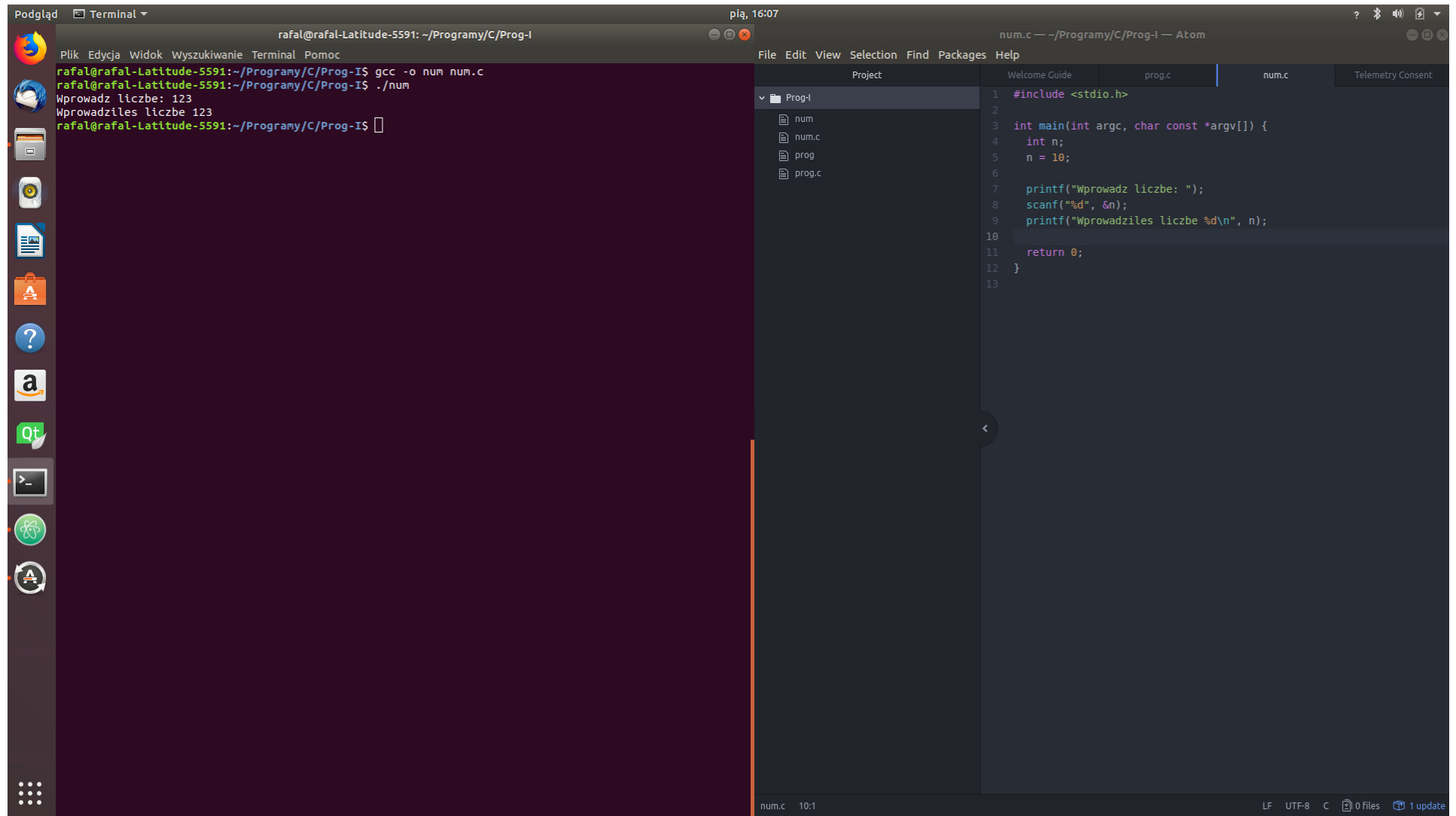
The status bar at the bottom of the Atom window indicates 'prog.c 8:1', 'LF', 'UTF-8', 'C', '0 files', and '1 update'.

Pobieranie liczby

```
1  #include <stdio.h>
2
3  □ int main(int argc, char const *argv[]) {
4      int n;
5      n = 10;
6
7      printf("Wprowadz liczbe: ");
8      scanf("%d", &n);
9      printf("Wprowadziles liczbe %d\n", n);
10
11     return 0;
12 }
13
14
```



Pobieranie liczby



The screenshot displays a Linux desktop environment. On the left, a vertical dock contains icons for various applications including Firefox, a file manager, and a terminal. The terminal window, titled 'rafal@rafal-Latitude-5591: ~/Programy/C/Prog-I', shows the following commands and output:

```
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ gcc -o num num.c
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ ./num
Wprowadz liczbe: 123
Wprowadziles liczbe 123
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$
```

On the right, the Atom code editor is open, showing the source code for 'num.c'. The code is as follows:

```
1 #include <stdio.h>
2
3 int main(int argc, char const *argv[]) {
4     int n;
5     n = 10;
6
7     printf("Wprowadz liczbe: ");
8     scanf("%d", &n);
9     printf("Wprowadziles liczbe %d\n", n);
10
11     return 0;
12 }
13
```

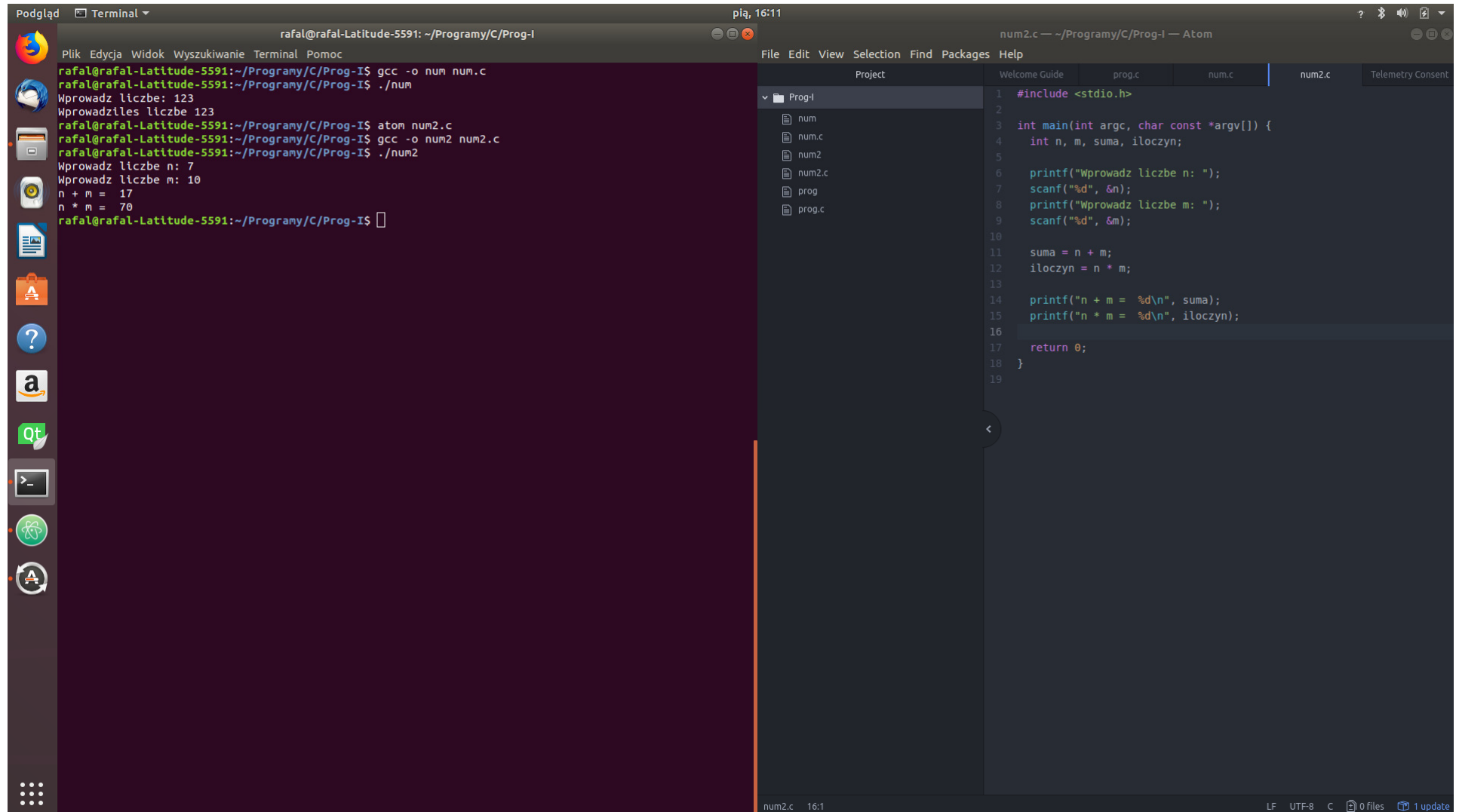
The Atom editor's interface includes a menu bar (File, Edit, View, Selection, Find, Packages, Help), a project sidebar showing the file structure of 'Prog-I' (num, num.c, prog, prog.c), and a status bar at the bottom indicating the current file is 'num.c' at line 10, column 1, with encoding 'UTF-8' and 1 update available.

Proste obliczenia

```
1  #include <stdio.h>
2
3  □ int main(int argc, char const *argv[]) {
4      int n, m, suma, iloczyn;
5
6      printf("Wprowadz liczbe n: ");
7      scanf("%d", &n);
8      printf("Wprowadz liczbe m: ");
9      scanf("%d", &m);
10
11     suma = n + m;
12     iloczyn = n * m;
13
14     printf("n + m = %d\n", suma);
15     printf("n * m = %d\n", iloczyn);
16
17     return 0;
18 }
19
20
```



Proste obliczenia



The screenshot displays a Linux desktop environment with a terminal window and an Atom code editor. The terminal window, titled "rafal@rafal-Latitude-5591: ~/Programy/C/Prog-I", shows the following commands and output:

```
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ gcc -o num num.c
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ ./num
Wprowadz liczbe: 123
Wprowadziles liczbe 123
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ atom num2.c
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ gcc -o num2 num2.c
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$ ./num2
Wprowadz liczbe n: 7
Wprowadz liczbe m: 10
n + m = 17
n * m = 70
rafal@rafal-Latitude-5591:~/Programy/C/Prog-I$
```

The Atom code editor, titled "num2.c — ~/Programy/C/Prog-I — Atom", shows the source code of the program:

```
1 #include <stdio.h>
2
3 int main(int argc, char const *argv[]) {
4     int n, m, suma, iloczyn;
5
6     printf("Wprowadz liczbe n: ");
7     scanf("%d", &n);
8     printf("Wprowadz liczbe m: ");
9     scanf("%d", &m);
10
11     suma = n + m;
12     iloczyn = n * m;
13
14     printf("n + m = %d\n", suma);
15     printf("n * m = %d\n", iloczyn);
16
17     return 0;
18 }
19
```

Jak działa komputer? Wstęp do algorytmów

Zadania :

- Napisz program obliczający pole powierzchni i obwód koła
- Napisz program obliczający pole powierzchni i obwód prostokąta

Jak działa komputer? Wstęp do algorytmów

Podsumowanie :

- Podstawowa architektura komputerów
- Czym jest algorytm?
- Rodzaje algorytmów
- Języki programowania
- Proces tworzenia programu
- Prosty przykład programu w języku C

Zapis liczb, typy zmiennych

Zapis liczb, typy zmiennych

Plan zajęć :

- Czym jest liczba
- Systemy liczbowe
- Zapis liczb całkowitych
- Liczby ze znakiem i bez znaku
- Kodowanie BCD
- Zapis stałopozycyjny
- Zapis zmiennopozycyjny IEEE
- Typy danych
- Deklaracja zmiennych
- Operator przypisania
- Operatory arytmetyczne
- Promocja i rzutowanie typów
- Formatowanie strumieni wejścia i wyjścia

Czym jest liczba?

Liczba jest pojęciem abstrakcyjnym, pewną ideą.

Jest to narzędzie, **za pomocą którego nasz umysł określa ilość.**

100	zapis dziesiętny
C	zapis rzymski
64₁₆	zapis szesnastkowy
1100100₂	zapis binarny
144₈	zapis oktalny
sto	zapis w języku polskim

Systemy liczbowe

Zapis liczby **25** w formie skreślanych kresek:



- w takim systemie do zanotowania liczby n stosuje się ciąg n znaków
- żeby ułatwić czytanie takich wartości, zwykle kreski układa się w grupy po 5
- zaletą takich systemów zapisu jest prostota ich użycia w przypadku zliczania obiektów
- wadą jest ilość miejsca potrzebnego na takie zapisanie liczb oraz trudność wykonywania obliczeń

Systemy liczbowe

Liczby rzymskie:

- w systemie rzymskim liczby zapisuje się za pomocą ciągów cyfr rzymskich
- jest to system pozycyjny, oznacza to że kolejność zapisu poszczególnych cyfr ma znaczenie
- cechuje go mniejsza objętość w stosunku do zapisu w formie skreślanych kresek
- jego wadą jest trudność dokonywania obliczeń

Systemy liczbowe

Dziesiętny pozycyjny system liczbowy:



- zaletą tego systemu jest mała objętość oraz łatwość wykonywania obliczeń

Podstawa systemu

- Podstawa systemu liczbowego to wartość, która jest w przypadku każdej cyfry podnoszona do potęgi
- Aby stworzyć system liczbowy o podstawie n , potrzeba n różnych cyfr
- Najmniejszą możliwą podstawą jest dwa
- W przypadku podstaw od 2 do 10 przyjęło się używać cyfr arabskich od 0 do $n-1$
- W przypadku podstaw większych od dziesięciu do wartości powyżej dziesięciu zwyczajowo używa się liter alfabetu, a..z lub A..Z (wielkość liter nie ma znaczenia); w ten sposób można zapisać systemy liczbowe o podstawie nieprzekraczającej 36 (10 cyfr, 26 liter)
- Nie ma powszechnie przyjętej konwencji zapisywania wartości w systemach liczbowych, gdzie 10 cyfr i 26 liter nie wystarcza

Binarny system liczbowy

System binarny zachowuje się analogicznie do systemu dziesiętnego, istnieją jednak dwie różnice:

- Używane są tylko dwie cyfry 0 i 1 (zamiast 0..9)
- Zamiast potęg dziesięciu używane są potęgi dwójki

Konwersja między zapisem binarnym i dziesiętnym

Aby zamienić wartość binarną na dziesiętną tworzymy sumę poszczególnych cyfr pomnożonych przez 2^i , gdzie i to numer cyfry licząc od zera

$$\begin{aligned} & 1100\ 1010_2 = \\ & = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = \\ & = 128 + 64 + 8 + 2 = 202_{10} \end{aligned}$$

Konwersja między zapisem dziesiętnym i binarnym

Zamiany zapisu dziesiętnego na binarny można dokonać za pomocą następującego algorytmu:

1. Jeżeli liczba jest parzysta wstawiamy 0, jeśli nieparzysta 1.
2. Dzielimy liczbę przez dwa odrzucając resztę z dzielenia.
3. Jeśli iloraz jest zerem działanie algorytmu zostaje zakończone.
4. Jeżeli iloraz nie jest zerem, a liczba jest nieparzysta, wstawiamy 1 przed dotychczasowy wynik. Jeżeli nie jest zerem, a liczba jest parzysta, przed dotychczasowy wynik wstawiamy 0.
5. Wracamy do kroku 2 i powtarzamy algorytm.

Konwersja między zapisem dziesiętnym i binarnym

sposób alternatywny

Wypisujemy kolejne potęgi 2 od jedynki aż do wartości większej bądź równej przekształcanej liczbie.

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1
0	1	1	0	0	1	0	1	0
202		74				10	2	
-128		-64				-8	-2	
<hr/>		<hr/>				<hr/>	<hr/>	
74		10				2	0	

Teraz analizujemy otrzymany ciąg rozpoczynając od lewej strony (największej wartości), jeżeli napotykamy na liczbę mniejszą bądź równą przekształcanej to wstawiamy 1 i od przekształcanej liczby odejmujemy stosowną wartość (którą poddamy dalszej analizie), w przeciwnym wypadku wstawiamy 0.

Szesnastkowy (heksadecymalny) system liczbowy

- Dwójkowe liczby są bardzo rozwlekłe (czytanie i zapisywanie takich wartości jest niewygodne)
- Programiści często zamiast zapisu dwójkowego stosują zapis szesnastkowy
- Zapis szesnastkowy ma podstawę 16

Konwersja między zapisem szesnastkowym i dziesiętnym

$$\begin{aligned} 1234_{16} &= \\ &= 1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0 = \\ &= 4096 + 512 + 48 + 4 = 4660_{10} \end{aligned}$$

Konwersja między zapisem szesnastkowym a dwójkowym

- Zapis szesnastkowy jest powszechnie stosowany z uwagi na swoją zwartość oraz łatwość zamiany liczb binarnych na heksadecymalne i odwrotnie

bin	hex	bin	hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Konwersja między zapisem szesnastkowym i dwójkowym

- aby zamienić liczbę szesnastkową na dwójkową, zamieniamy każdą cyfrę szesnastkową na odpowiadające jej cztery bity

$$\begin{array}{cccc}
 & A & B & C & D \\
 & 1010 & 1011 & 1100 & 1101 \\
 = & \mathbf{1011} & \mathbf{1011} & \mathbf{1100} & \mathbf{1101}_2
 \end{array}$$

Konwersja między zapisem dwójkowym i szesnastkowym

- aby zamienić liczbę dwójkową na szesnastkową należy uzupełnić liczbę binarną zerami, aby jej długość była krotnością czwórki, następnie dzielimy liczbę na grupy czterobitowe i według tabeli zamieniamy poszczególne czwórki na cyfry szesnastkowe

Zapis liczb całkowitych

- Większość współczesnych komputerów działa binarnie (wewnętrznie zarówno wartości, jak i inne obiekty zapisywane są dwójkowo)
- Większość systemów nie może używać całkiem dowolnych wartości binarnych, konieczne jest używanie wielkości o określonym rozmiarze

Bity

Najmniejszą jednostką danych w komputerze binarnym jest bit

Bit może przyjmować tylko dwie różne wartości:

- zero (0) i jeden (1)
- fałsz (0) i prawda (1)
- wyłączony (0) i włączony (1)
- mężczyzna (0) i kobieta (1)
- źle (0) i dobrze (1)

Łańcuchy bitowe

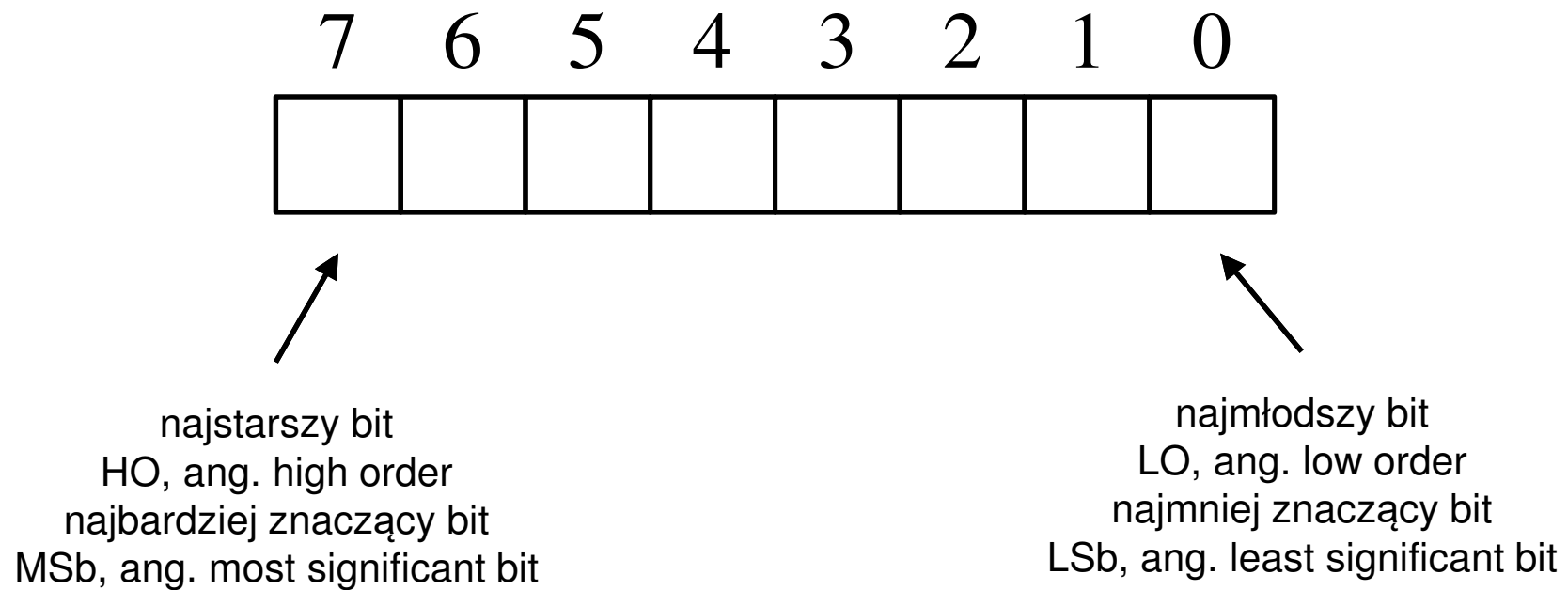
Łącząc bity w łańcuchy, można stworzyć zapis binarny równoważny każdemu innemu zapisowi liczb (np. ósemkowemu czy szesnastkowemu).

W większości komputerów nie można łączyć ze sobą całkiem dowolnej liczby bitów, trzeba posługiwać się ciągami o ustalonych długościach.

Cztery bity to pół bajta. Zwykle komputery nie zapewniają wydajnych metod dostępu do takich obiektów. Półbajty są ciekawe dlatego, że pojedynczą cyfrę szesnastkową zapisuje się właśnie przy użyciu pół bajta.

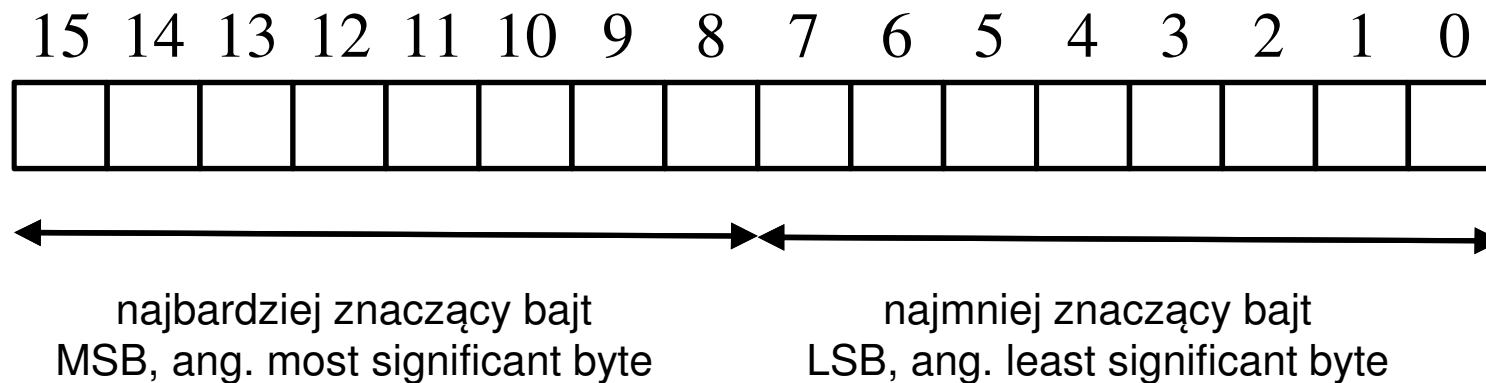
Bajt to osiem bitów. W przypadku wielu CPU jest to najmniejszy obiekt, jakiemu można nadać adres. Chodzi o to, że procesory potrafią sprawnie pobierać dane rozmieszczone w pamięci co osiem bitów. Z tego powodu w wielu językach programowania nawet bardzo małe obiekty zajmują jeden bajt niezależnie od tego, ile bitów jest w nich potrzebnych.

Bajt

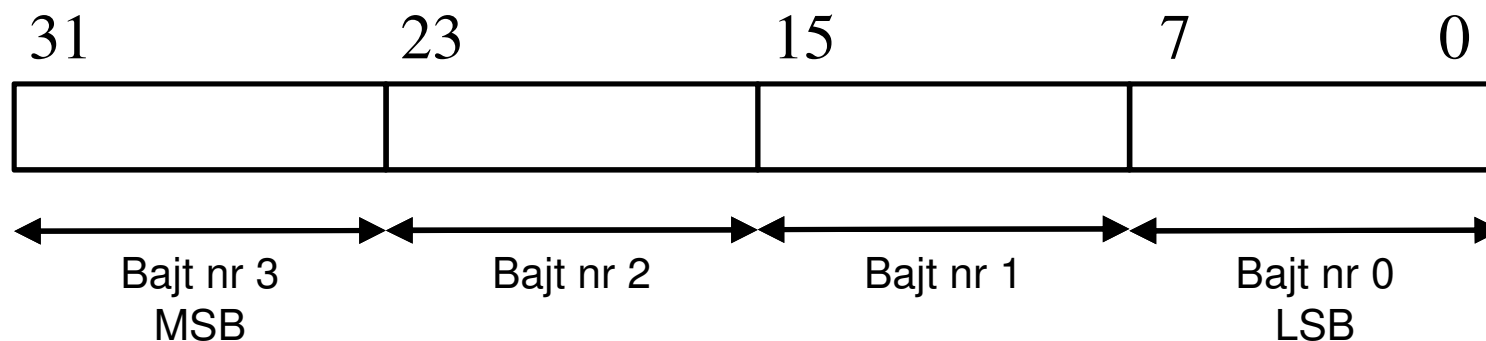


Słowo

- Pojęcie słowo zmienia swoje znaczenie w zależności od tego, o jakiej maszynie mowa
- Czasami słowo to obiekt 16-bitowy (w przypadku maszyn klasy 80x86)



Podwójne słowo



Kolejność bajtów

W sytuacjach, kiedy liczby całkowite lub jakiegokolwiek inne dane zapisywane są przy użyciu wielu (przynajmniej dwóch) bajtów, nie istnieje jeden unikatowy sposób uporządkowania tych bajtów w pamięci lub w czasie transmisji przez dowolne medium i musi być użyta jedna z wielu konwencji ustalająca kolejność bajtów.

Big endian

Procesor zapisujący 32-bitowe wartości w pamięci, przykładowo 0x4A3B2C1D pod adresem 100, umieszcza dane, zajmując adresy od 100 do 103 w nast. kolejności

	100	101	102	103	
...	4A	3B	2C	1D	...

Little endian

Procesor zapisujący 32-bitowe wartości w pamięci, przykładowo 0x4A3B2C1D pod adresem 100, umieszcza dane, zajmując adresy od 100 do 103 w nast. kolejności

	100	101	102	103	
...	1D	2C	3B	4A	...

Liczby ze znakiem i bez znaku

Korzystając z n bitów można zapisać 2^n różnych wartości

Np. na jednym bajcie można zapisać wartości:

- od 0 do 2^8-1 , czyli od **0** do **255**

albo

- od -2^7 do 2^7-1 , czyli od **-128** do **127**

System uzupełnienia do dwójki

W systemie uzupełnienia do dwójki wykorzystuje się bit HO jako bit znaku

- Jeżeli bit ten jest zerem, liczba jest nieujemna
- Jeżeli bit ten jest jedynką, liczba jest ujemna

Aby zamienić liczbę na ujemną zapisaną jako uzupełniona do dwójki, korzystamy z następującego algorytmu:

1. Negujemy wszystkie bity liczby, czyli zamieniamy zera na jedynki a jedynki na zera.
2. Dodajemy do uzyskanego wyniku 1.

Aby zamienić liczbę ujemną na dodatnią wykorzystujemy ten sam algorytm.

Przykład

Obliczenie 8-bitowego odpowiednika wartości dziesiętnej **-5**

0000 0101	5 (binarnie)
1111 1010	negacja wszystkich bitów
1111 1011	dodanie jedynki do wyniku

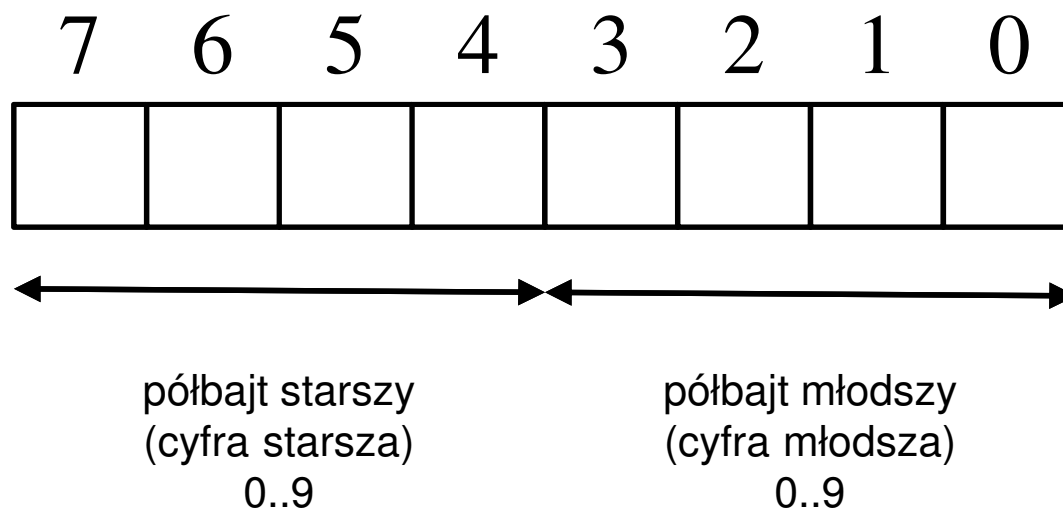
UWAGA!!!

W systemie uzupełnienia do dwójki nie można negować najmniejszej wartości ujemnej.

BCD (ang. binary-coded decimal)

Zapis dziesiętny kodowany binarnie

- zapis ten pozwala zapisać wartości dziesiętne w systemie dwójkowym
- wartości BCD składają się z ciągów półbajtów, z których każdy służy do zapisu wartości z zakresu 0..9
- każdy półbajt odpowiada w kodzie BCD jednej cyfrze dziesiętnej



Zapis stałopozycyjny

- W komputerach stosowane są dwie metody zapisu ułamków: zapis stałopozycyjny i zapis zmiennopozycyjny
- Jeżeli arytmetyka zmiennopozycyjna nie jest obsługiwana przez urządzenie, w oprogramowaniu które wymaga szybkiego przetwarzania ułamków stosuje się arytmetykę stałopozycyjną (systemy wbudowane); narzut z programowym przetwarzaniem ułamków jest mniejszy niż w przypadku obliczeń zmiennopozycyjnych.
- Wartości ułamkowe mieszczą się między zerem a jedynką, w systemach pozycyjnych zapisujemy je za pomocą cyfr umieszczonych po przecinku (dziesiętnym, binarnym, ...)
- W systemie dwójkowym każdy bit na prawo od kropki oznacza zero lub jedynkę mnożone przez kolejne ujemne potęgi dwójki. Jeżeli zatem zapisujemy dwójkowo wartość z częścią ułamkową, tę część ułamkową zapisujemy jako sumę ułamków binarnych. Aby np. zapisać dwójkowo 5,25 użylibyśmy postaci:

$$101,02_2 = 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-2} = 4 + 1 + 0,25 = 5,25_{10}$$

Zapis stałopozycyjny

problem dokładnego zapisu dowolnej liczby

- Liczby stałopozycyjne to niewielki podzbiór liczb rzeczywistych
- Między dwiema liczbami całkowitymi istnieje nieskończenie wiele wartości, liczby stałopozycyjne nie pozwalają dokładnie zapisać dowolnej liczby mieszczącej się między nimi (potrzebnych byłoby do tego nieskończenie wiele bitów)
- W przypadku zapisu stałopozycyjnego liczby rzeczywiste trzeba przybliżać

Przykład: 8-bitowy format stałopozycyjny

6 bitów
część całkowita

0..63
-32..+31

2 bity
część ułamkowa

0,0 0,25 0,5 0,75

↑
tylko 4 różne wartości

- Nie można zapisać w ten sposób liczby **1,3**
- Zamiast niej musi być wartość **1,25**

Możliwe rozwiązanie

system stałopozycyjny z zapisem BCD

16-bitowy format

8 bitów
część całkowita

8 bitów
część ułamkowa

- Można zapisać liczby od 0,0 do 99,99
- Dokładność na prawo od przecinka wynosi dwa znaki
- 1,3 można zapisać jako: 0 1 3 0
- Póki używamy w naszych liczbach części ułamkowych jedynie z zakresu 0,00..0,99, taki zapis BCD jest dokładniejszy od odpowiadającego mu zapisu binarnego (z 8-bitową częścią ułamkową)
- W ogólnym przypadku zapis binarny jest dokładniejszy, w przypadku 8-bitów format binarny pozwala zapisać 256 różnych wartości, zaś format BCD tylko 100

Skalowane formaty liczbowe

- Jedną z zalet skalowalnego formatu liczbowego jest możliwość wybrania podstawy – niekoniecznie dziesiętnej. Jeżeli np. mamy do czynienia z ułamkami trójkowymi, możemy pomnożyć wartości wejściowe przez trzy lub potęgi trzech i dokładnie zapisać wartości takie jak $1/3$, $2/3$, $4/9$, $7/27$ itd. Wartości tych nie można dokładnie zapisać w systemie binarnym ani dziesiętnym.
- Aby zapisać ułamki mnoży się wartość wejściową przez pewną wartość tak, aby część ułamkową zmienić w całkowitą
- Np. jeżeli zależy nam na dokładności dwóch miejsc po przecinku, mnożymy wartości wejściowe przez 100.

$$1,3 \cdot 100 = 130$$

$$1,5 \cdot 100 = 150$$

$$130 + 150 = 280$$

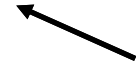
- Dokładność obliczeń jest większa, koszt to ograniczenie zakresu liczb
- W przypadku dodawania lub odejmowania liczb skalowanych oba argumenty muszą być skalowane tak samo

Skalowane formaty liczbowe

- W przypadku mnożenia i dzielenia argumenty nie muszą być przed wykonaniem działania tak samo skalowane, ale już po wykonaniu takiego działania konieczne może być skalowanie wyniku

$$\begin{aligned} & ((0,25 \cdot (100)) \cdot (0,01 \cdot (100))) = \\ & = 0,25 \cdot 0,01 \cdot (100 \cdot 100) = \\ & = 0,0025 \cdot (10\ 000) = 25 \end{aligned}$$

wynik jest skalowany
czynnikiem 10 000



$$\begin{aligned} & (5 \cdot 100) \div (2,5 \cdot 100) = \\ & = 500 \div 250 = 2 \end{aligned}$$



Tak naprawdę otrzymaliśmy 0,02 (jeżeli weźmiemy pod uwagę skalowanie)
W rzeczywistości powinniśmy uzyskać 200 (czyli 2,0)

Zapis wymierny

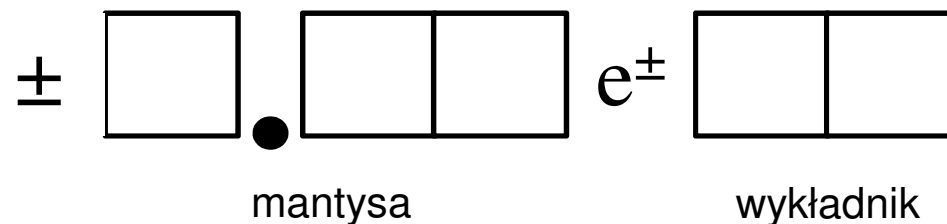
- W zapisie wymiernym do zapisu ułamków używa się par liczb, z których jedna (**n**) to licznik (ang. numerator), druga (**d**) to mianownik (ang. denominator)
- Póki n i d są względnie pierwsze (czyli nie dzielą się bez reszty przez tę samą liczbę), otrzymujemy wygodny zapis wartości ułamkowych
- Arytmetyka jest prosta: korzysta się tu ze szkolnych algorytmów na dodawanie, odejmowanie, mnożenie i dzielenie
- Problemem jest tutaj możliwość powstawania dużych liczników i mianowników – na tyle dużych, że mogą wystąpić przepełnienia; poza tym jednak zapis ten jest wygodny, gdyż pozwala zapisać bardzo dużo ułamków

$$x = \frac{n}{d}$$

Zapis zmiennopozycyjny

- Liczby zmiennopozycyjne to jedynie przybliżenie liczb rzeczywistych

Prosty zapis zmiennopozycyjny



- Zaletą zapisu zmiennopozycyjnego (z mantysą i wykładnikiem) jest możliwość zapisywania liczb z bardzo szerokiego zakresu
- nie można jednak dokładnie zapisać tylu wartości ile da się zapisać w analogicznym formacie całkowitoliczbowym (chodzi o to że zapis zmiennopozycyjny dostarcza wielu form tej samej wartości, np. 1,00e+1 i 0,10e+2 to ta sama liczba)
- Zapis ten komplikuje proces obliczania

Zapis zmiennopozycyjny

dodawanie i odejmowanie

- Kiedy dodajemy lub odejmujemy dwie liczby zapisane naukowo, musimy wyrównać ich wykładniki

$$\begin{aligned} 1,23e1 + 4,56e0 &= \\ = 1,23e1 + 0,456e1 &= 1,686e1 \end{aligned}$$

wynik nie mieści się
na założonych
trzech cyfrach
znaczących

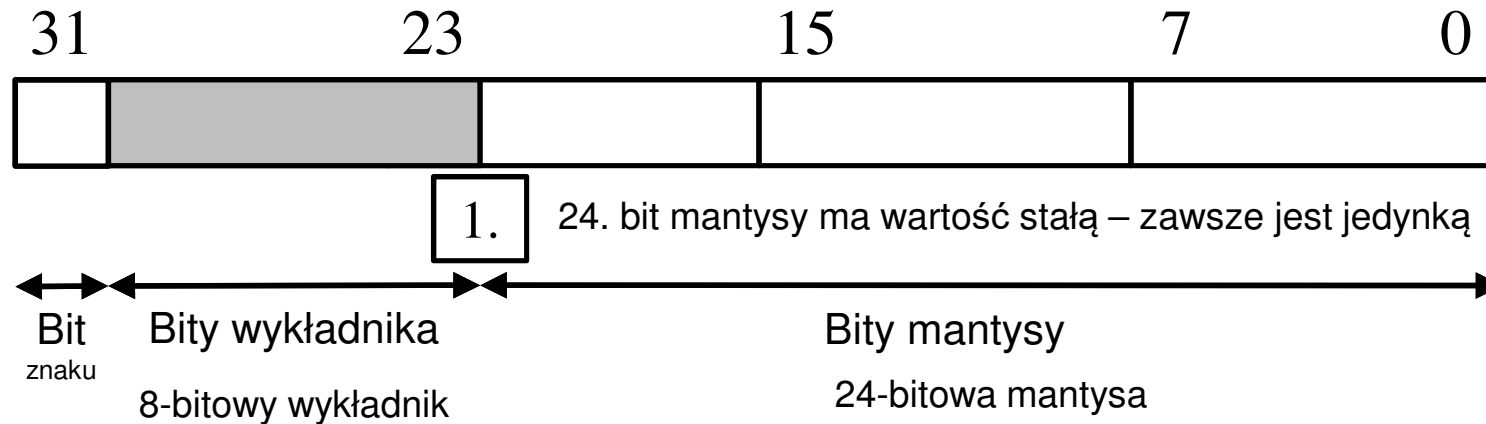
- Jeżeli wynik nie mieści się na założonej ilości cyfr znaczących to trzeba go zaokrąglić lub obciąć (zaokrąglanie daje dokładniejsze wyniki)
- Brak precyzji (czyli zbyt mało cyfr lub bitów w trakcie obliczeń) wpływa na dokładność (czyli poprawność tych obliczeń)
- Kolejność obliczeń może wpływać na dokładność wyniku

Zapis zmiennopozycyjny

mnożenie i dzielenie

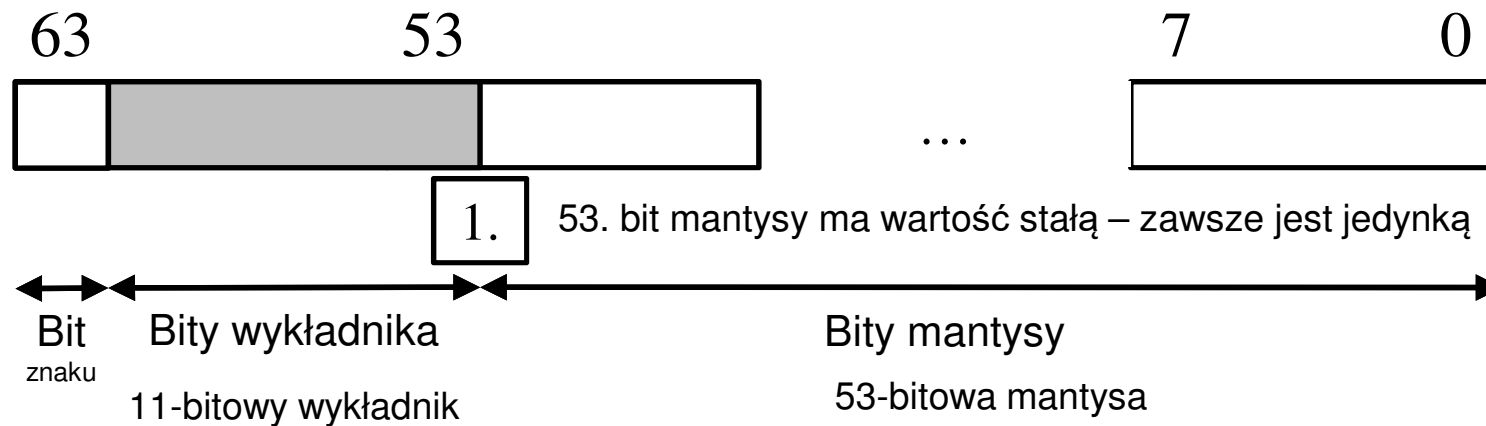
- W przypadku mnożenia i dzielenia nie trzeba przed wykonaniem działań wyrównywać wykładników
- Dodaje się wykładniki i mnoży mantysy (ewentualnie odejmuje się wykładniki i dzieli mantysy)
- Mnożenie i dzielenie same z siebie dają poprawne wyniki, jednak powodują one eskalację błędów, które mogły pojawić się we wcześniejszych obliczeniach
- Kiedy wykonywany jest ciąg obliczeń obejmujący dodawanie, odejmowanie, mnożenie i dzielenie należy dążyć w pierwszej kolejności do wykonania mnożenia i dzielenia.
- Kiedy sprawdza się równość dwóch liczb zmiennopozycyjnych, zawsze należy ustalić, czy różnica między nimi jest mniejsza od najmniejszego dopuszczalnego błędu.

Format zmiennopozycyjny IEEE pojedynczej precyzji



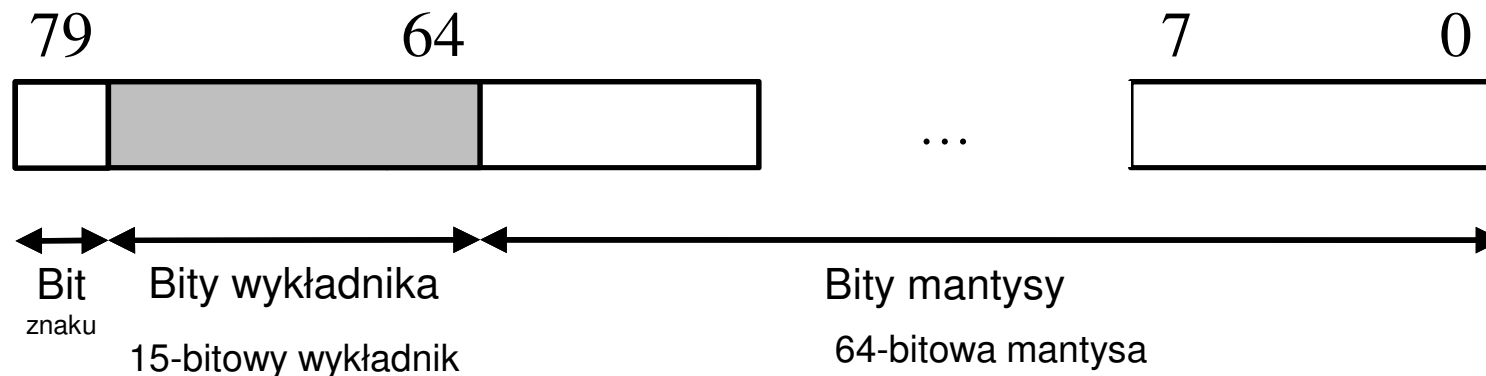
- Obecność narzuconego bitu – jedynki w mantysie powoduje, że wartość ta zawsze pozostaje w zakresie od 1,0 do 2,0
- W mantysie zamiast uzupełnienia do dwójki używa się formatu uzupełnienia do jedynki, oznacza to, że 24-bitowe wartości mantysy to liczby binarne bez znaku, a bit znaku na miejscu 31. określa czy wartość jest dodatnia czy ujemna
- Wykładnik ma 8 bitów i wykorzystuje format nadmiaru -127
- Format ten daje 6 ½ cyfry dziesiętnej dokładności
- Dynamiczny zakres to około $2^{\pm 128}$, czyli około $10^{\pm 38}$

Format zmiennopozycyjny IEEE podwójnej precyzji



- W mantysie zamiast uzupełnienia do dwójki używa się formatu uzupełnienia do jedynki, oznacza to, że 53-bitowe wartości mantysy to liczby binarne bez znaku, a bit znaku na miejscu 63. określa czy wartość jest dodatnia czy ujemna
- Wykładnik ma 11 bitów i wykorzystuje format nadmiaru -1023
- Format ten daje $14 \frac{1}{2}$ cyfry dziesiętnej dokładności
- Dynamiczny zakres to około $10^{\pm 308}$

Format zmiennopozycyjny IEEE zwiększonej precyzji



- W przeciwieństwie do wartości pojedynczej i podwójnej precyzji, tutaj mantysa nie ma ustalonego najstarszego bitu o wartości jeden
- Wykładnik ma 15 bitów i wykorzystuje format nadmiaru -16383
- W koprocessorach 80x86 wszystkie obliczenia robione są właśnie w formacie o zwiększonej precyzji

Format zmiennopozycyjny IEEE - normalizacja

- Aby w trakcie obliczeń zmiennopozycyjnych zachować maksymalną precyzję, zwykle korzysta się z wartości znormalizowanych.
- Znormalizowana wartość zmiennopozycyjna to taka, w której najstarszy bit mantysy zawiera jedynkę.
- Trzymanie normalizowanych liczb zmiennopozycyjnych jest korzystne, gdyż pozwala zachować maksymalną liczbę bitów dokładności.
- Jeśli kilka najstarszych bitów mantysy to zera, mantysa tak naprawdę ma mniej bitów precyzji, niż można by oczekiwać.
- Zatem obliczenia zmiennopozycyjne są dokładniejsze, jeśli operuje się tylko na wartościach znormalizowanych.

Format zmiennopozycyjny IEEE - zaokrąglanie

- Podczas wykonywania obliczeń funkcje zmiennopozycyjne mogą podawać wyniki o większej precyzji niż przewiduje to format zmiennopozycyjny (ten nadmiar zapisywany jest na bitach kontrolnych)
- Po zakończeniu obliczeń wynik trzeba zapisać ponownie w zmiennej zmiennopozycyjnej, więc z dodatkowymi bitami precyzji trzeba coś zrobić
- Sposób obsługi owych dodatkowych bitów kontrolnych przez system to właśnie proces zaokrąglania, który może mieć wpływ na dokładność obliczeń

Tradycyjnie używane są cztery rodzaje zaokrągleń:

- Obcinanie
- Zaokrąglanie w górę
- Zaokrąglanie w dół
- Zaokrąglanie do najbliższej wartości

Specjalne wartości zmiennopozycyjne

- Format zmiennopozycyjny IEEE zawiera specjalne metody kodowania kilku wartości wyróżnionych
- Normalnie wykładnik nie zawiera samych zer ani samych jedynek, takie wykładniki wskazują właśnie na wartość specjalną

NaN	Zapis zmiennopozycyjny	Wartość
SNaN	32-bitowy	%s_11111111_0xxxx...xx
SNaN	64-bitowy	%s_1111111111_0xxxxx...x
SNaN	80-bitowy	%s_1111111111_0xxxxx...x
QNaN	32-bitowy	%s_11111111_1xxxx...xx
QNaN	64-bitowy	%s_1111111111_1xxxxx...x
QNaN	80-bitowy	%s_1111111111_1xxxxx...x

Specjalne wartości zmiennopozycyjne

- Dwie inne wartości specjalne zapisuje się za pomocą wykładnika zawierającego same jedynki i mantysy zawierającej same zera. Wtedy bit znaku decyduje, czy wynik to $+\infty$, czy $-\infty$. Kiedy w obliczeniach jako jeden z argumentów pojawia się ∞ , działania arytmetyczne dadzą jedną z dobrze określonych wartości.

działanie	wynik	działanie	wynik
$n \div \pm\infty$	0	$n - \infty$	$-\infty$
$\pm\infty \cdot \pm\infty$	$\pm\infty$	$\pm 0 \div \pm 0$	NaN
$\pm \text{nie-zero} \div 0$	$\pm\infty$	$\infty - \infty$	NaN
$\infty + \infty$	∞	$\pm\infty \div \pm\infty$	NaN
$n + \infty$	∞	$\pm\infty \cdot 0$	NaN

- W końcu jeśli wszystkie bity wykładnika są zerami, bit znaku wskazuje, z jaką wartością mamy do czynienia -0, czy +0. Z punktu widzenia porównań i wszelkich działań +0 jest równe -0.

Wyjątki obliczeń zmiennopozycyjnych

Standard liczb zmiennopozycyjnych IEEE zawiera definicje pewnych warunków brzegowych, kiedy procesor zmiennopozycyjny lub oprogramowanie obsługujące liczby zmiennopozycyjne powinny poinformować aplikację o zaistniałej sytuacji.

Wyjątki te to:

- Nieprawidłowa operacja
- Dzielenie przez zero
- Nieznormalizowany operand
- Przepelnienie liczby
- Niedopelnienie liczby
- Niedokładny wynik

Ogólna składnia języka C

// /* */ komentarze

#dyrektywa_preprocesora

deklaracja;

instrukcja;

{ blok_instrukcji }

```
1  #include <stdio.h>
2
3  int main(int argc, char const *argv[]) {
4      int n;
5      n = 10;
6
7      printf("Wprowadz liczbe: ");
8      scanf("%d", &n);
9      printf("Wprowadziles liczbe %d\n", n);
10
11     return 0;
12 }
13
14
```



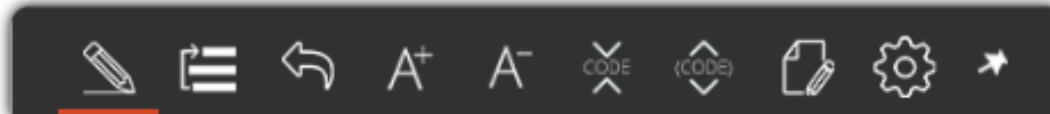
Słowa kluczowe języka C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Deklaracje zmiennych

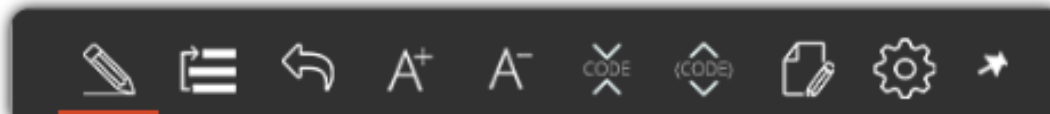
[specyfikator] [modyfikator] typ identyfikator[, identyfikator];

```
1 int a;  
2 static char a, b;  
3 unsigned long zm1;  
4
```



... identyfikator [= wartosc_inicjalizacyjna] ...

```
1 const int a = 0;  
2 float b = 1, c = 2;  
3
```



deklaracja zmiennej jest jednocześnie jej definicją

deklaracja zmiennej zawiera następujące informacje dla kompilatora:

- nazwę zmiennej
- wielkość zajmowanej pamięci
- zakres zmiennej

Typy całkowite

Zakres liczb całkowitych zależy od używanego kompilatora.

Typ	Min. wartość	Max. wartość	Ilość bajtów
char	-128	128	1
unsigned char	0	255	1
short int	-32 768	32 767	2
unsigned short int	0	65 535	2
int	-2 147 483 648	2 147 483 647	4
unsigned int	0	4 294 967 295	4
long int	-2 147 483 648	2 147 483 647	4
unsigned long int	0	4 294 967 295	4

$\text{bool} \leq \text{char} \leq \text{short int} \leq \text{int} \leq \text{long int}$

Typy zmiennoprzecinkowe

Zakres liczb zmiennoprzecinkowych zależy od używanego kompilatora.

Typ	Zakres wartości	Znaczące cyfry	Ilość bajtów
float	$\approx 3.4 \cdot 10^{\pm 38}$	7	4
double	$\approx 1.7 \cdot 10^{\pm 308}$	15	8
long double	$\approx 1.2 \cdot 10^{\pm 4932}$	19	10

float ≤ double ≤ long double

Podstawowe operatory

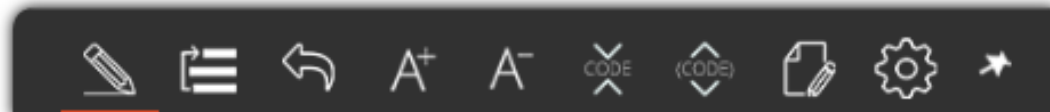
Operator przypisania

=

lewa_strona = prawa_strona

- Obie strony operatora powinny mieć zgodny typ danych
- W przypadku różnych typów wystąpi automatyczne rzutowanie do typu lewej strony
- Operator przypisania jest prawostronnie łączny

```
1 int a, b, c;  
2 ...  
3 a = b;  
4 a = b = c;  
5
```



Podstawowe operatory

Operatory arytmetyczne

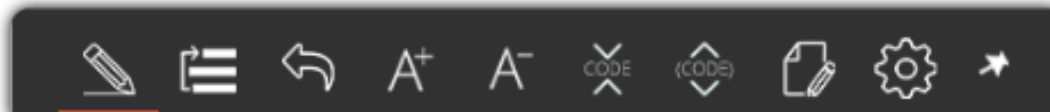
+ - * / %

lewa_strona <op.> prawa_strona

- Operatory arytmetyczne *, / i % mają wyższy priorytet od operatorów + i -
- Operatory arytmetyczne są lewostronnie łączne

wyrażenie <op.> wyrażenie <op.> wyrażenie <op.>

```
1  int a, b, c;  
2  ...  
3  a = b + b;  
4  a = a + 1;  
5  a = a - a - b;  
6  a = a + c * d;  
7
```



Podstawowe operatory

Operatory przypisania

`+=` `-=` `*=` `/=` `%=`

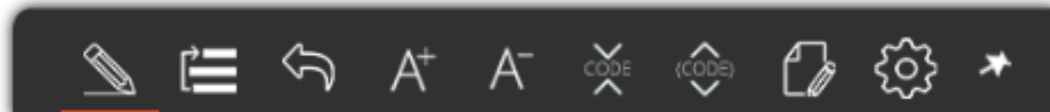
`lewa_strona <.>= prawa_strona`



`lewa_strona = lewa_strona <.> prawa_strona`

- Prawostronnie łączne
- Operatory przypisania mają jeden z najniższych priorytetów

```
1 int a, b, c;  
2 ...  
3 a += 1;  
4 a *= a;  
5 a *= a + 1;  
6
```



Ekran konsoli

```
int printf (const char* format, ...)
```

%[flags][width][.precision][length]specifier

specifier	Output	example
c	Character	a
d, i	Signed decimal integer	392
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
f	Decimal floating point, lowercase	392.65
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
o	Unsigned octal	610
s	String of characters	tekst
u	Unsigned decimal integer	392
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
p	Pointer address	B800:0000
%	A % followed by another % character will write a single % to the stream.	%

Klawiatura

int scanf (const char* format, ...)

%[*][width][length]specifier

specifier	description
c	Character
d	Decimal integer
e, E, f, g, G	Floating point number
o	Octal integer
s	String of characters
u	Decimal integer
x, X	Hexadecimal integer

Przykład

```
1  #include <stdio.h>
2
3  □ int main(int argc, char const *argv[]) {
4      int n;
5      n = 10;
6
7      printf("Wprowadz liczbe: ");
8      scanf("%d", &n);
9      printf("Wprowadziles liczbe %d\n", n);
10
11     return 0;
12 }
13
14
```



Typ całkowity – sytuacje specyficzne

Typ całkowity występuje w dwóch odmianach (char, int)

- signed – ze znakiem (domyślnie)
- unsigned (bez znaku)

Ta sama zawartość pamięci (rejestru) ma dwie różne interpretacje

signed \neq unsigned

Specjalne znaczenie wartości „-1” dla typu ze znakiem oznacza, że wszystkie bity przyjmują wartość 1

Dla 16 bitów $-1 = 65535 = 0xFFFF$

Mieszanie typów signed i unsigned wymaga większej uwagi i staranności

Porównanie liczby ze znakiem i bez znaku daje wynik ***nieokreślony!***

Typ całkowity – sytuacje specyficzne

Przekroczenie zakresu typu w wyniku dodawania

- dla zmiennej 16 bitowej bez znaku:

$$32767 + 1 = 32768$$

$$65535 + 1 = 0$$

- dla zmiennej 16 bitowej ze znakiem:

$$32767 + 1 = -32768$$

$$65535 + 1 = 0 \quad (-1 + 1 = 0)$$

Powyższe obliczenia są prawdziwe tylko dla arytmetyki U2!

Typ całkowity – sytuacje specyficzne

Przekroczenie zakresu typu w wyniku mnożenia

- dla zmiennej 16 bitowej bez znaku:

$$256 * 256 = 0$$

- dla zmiennej 16 bitowej ze znakiem:

$$128 * 256 = -32768$$

- w ogólnym przypadku mnożenie dwóch liczb n bitowych daje w wyniku liczbę $2n$ bitową.
- przy małych liczbach problem ten nie jest dostrzegany
- przy skomplikowanych wyrażeniach taki błąd jest trudny do wykrycia

Zapis liczb, typy zmiennych

Zadania :

- Napisz program , który wczytuje ze standardowego wejścia liczbę wymierną i wypisuje ją na standardowym wyjściu – iloraz 2 liczb całkowitych
- Napisz program , który wczytuje ze standardowego wejścia trzy liczby całkowite, a następnie wypisuje je w oddzielnych liniach na standardowym wyjściu
- Napisz program , który wczytuje ze standardowego wejścia liczbę całkowitą i wypisuje na standardowym wyjściu liczbę o jeden większą
- Napisz program , który wczytuje ze standardowego wejścia trzy liczby całkowite i wypisuje na standardowym wyjściu ich średnią arytmetyczną
- Napisz program , który wczytuje ze standardowego wejścia liczbę wymierną i wypisuje ją na standardowym wyjściu z dokładnością do 2 miejsc po przecinku.
- Napisz program , który wczytuje ze standardowego wejścia liczbę wymierną i wypisuje ją na standardowym wyjściu w notacji wykładniczej (czyli takiej, w której 0.2 to 2.0e-1).

Zapis liczb, typy zmiennych

Podsumowanie :

- Czym jest liczba
- Systemy liczbowe
- Zapis liczb całkowitych
- Liczby ze znakiem i bez znaku
- Kodowanie BCD
- Zapis stałopozycyjny
- Zapis zmiennopozycyjny IEEE
- Typy danych
- Deklaracja zmiennych
- Operator przypisania
- Operatory arytmetyczne
- Promocja i rzutowanie typów
- Formatowanie strumieni wejścia i wyjścia

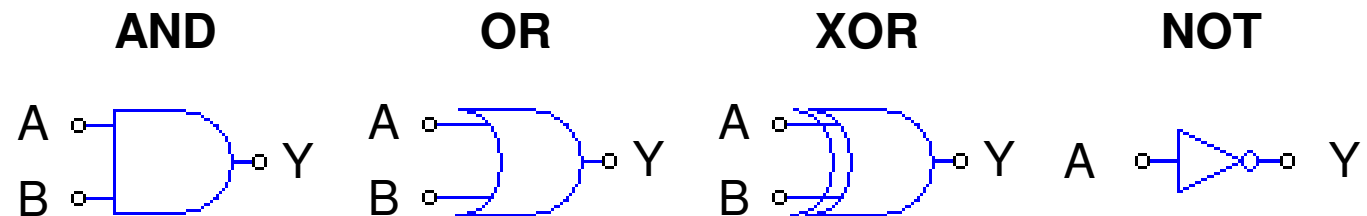
Arytmetyka binarna i działania na bitach

Arytmetyka binarna i działania na bitach

Plan zajęć :

- Operacje logiczne na bitach
- Operacje logiczne na liczbach binarnych i ciągach bitów
- Przesunięcia i rotacje
- Pakowanie i rozpakowywanie danych
- Operatory logiczne
- Operatory bitowe (binarne)
- Konstrukcja warunkowa if
- Operator warunkowy ?
- Konstrukcja warunkowa switch ... case

Podstawowe operacje na bitach



A	B		Y	Y	Y	Y
0	0		0	0	0	1
0	1		0	1	1	1
1	0		0	1	1	0
1	1		1	1	0	0

Podstawowe operatory

Operatory bitowe

- dwuargumentowe

& | ^ >> <<

lewa_strona <-> prawa_strona

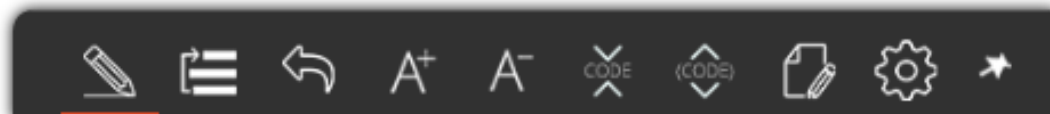
- jednoargumentowe

~

<-> prawa_strona

```

1 unsigned char a = 240, b = 15;
2 // a = %1111_0000 b = %0000_1111
3 a = ~a;          // a = %0000_1111   a = 15
4 a = a | b;       // a = %1111_1111   a = 255
5 a &= b;          // a = %0000_0000   a = 0
6 a = a ^ b;       // a = %1111_1111   a = 255
7
```

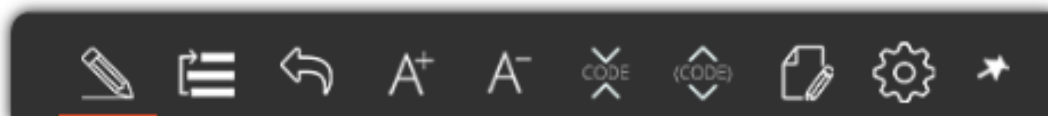


Ustawianie / kasowanie wybranych bitów

Maskowanie

- Ustawianie wybranych bitów na 1
te bity, które mają być ustawione muszą w masce mieć wartość 1
wynik = dana | maska;
- Ustawianie wybranych bitów na 0
te bity, które mają być ustawione muszą w masce mieć wartość 0
wynik = dana & maska;

```
1 unsigned char dana = 0x5a, maska = 0x0f, wynik = 0;  
2 // dana = %0101_1010 maska = %0000_1111  
3 wynik = dana | maska;      // wynik = %0101_1111   wynik = 0x5f  
4 wynik = dana & maska;      // wynik = %0000_1010   wynik = 0x0a  
5
```



Testowanie wybranych bitów

Maskowanie

- Sprawdzanie, czy wybrane bity są 1
te bity, które mają być sprawdzone muszą w masce mieć wartość 1
wynik = dana & maska;
następnie sprawdzamy czy (wynik == maska)
- Sprawdzanie, czy wybrane bity są 0
te bity, które mają być sprawdzone muszą w masce mieć wartość 1
wynik = (dana & maska) ^ maska;
następnie sprawdzamy czy (wynik == maska)

Tworzenie liczników modulo za pomocą operacji AND

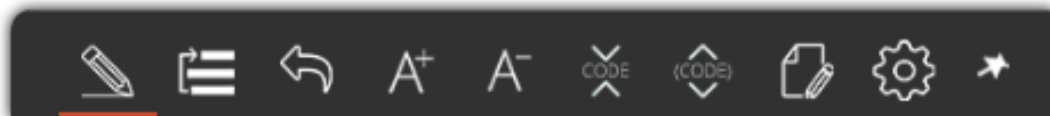
- licznik **modulo** **n** można w języku c zrealizować za pomocą instrukcji:

licznik = (licznik + 1) % n;

- Można również wykorzystać do tego celu operator bitowy AND

licznik = (licznik + 1) & (n - 1);

```
1 unsigned char dana = 0x5a, maska = 0x0f, wynik = 0;
2 // dana = %0101_1010 maska = %0000_1111
3 wynik = dana | maska;      // wynik = %0101_1111   wynik = 0x5f
4 wynik = dana & maska;      // wynik = %0000_1010   wynik = 0x0a
5
```

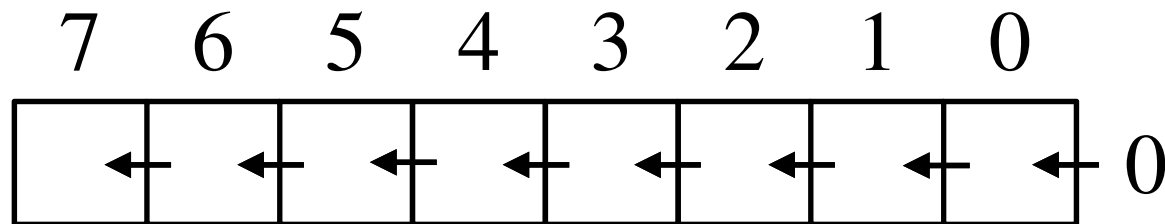


Przesunięcia bitowe

przesunięcie bitowe w lewo

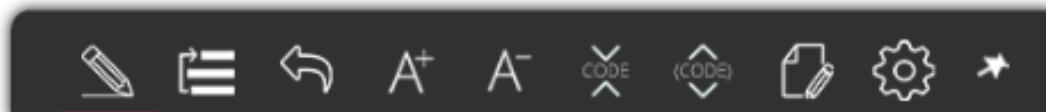
- Przesunięcie w lewo oznacza przesunięcie każdego bita łańcucha o jedno miejsce w lewo (do bita zerowego wstawiamy 0)

dana = dana << 1;



```

1 unsigned char dana = 0x5a, przesuniecie = 2, wynik = 0;
2 // dana = %0101_1010
3 wynik = dana << przesuniecie;      // wynik = %0110_1000   wynik = 0x68
4
5
```

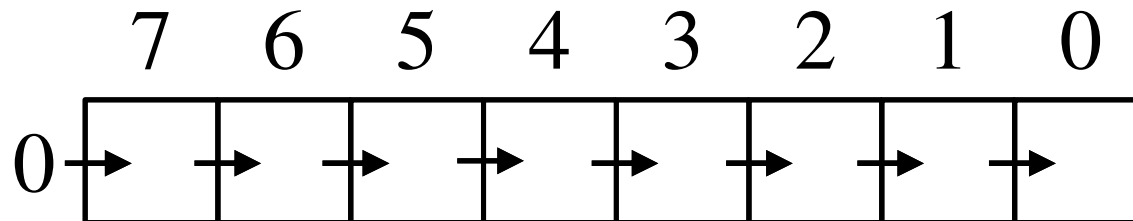


Przesunięcia bitowe

przesunięcie bitowe w prawo

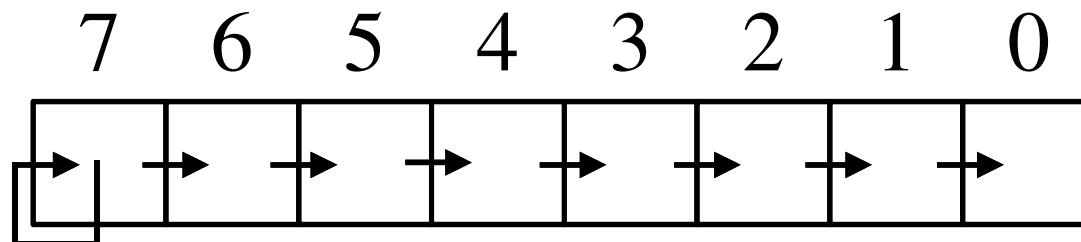
- Przesunięcie w prawo oznacza przesunięcie każdego bita łańcucha o jedno miejsce w prawo (do bita najstarszego wstawiamy 0)

dana = dana >> 1; //dla typów bez znaku



- Przesunięcie arytmetyczne w prawo oznacza przesunięcie każdego bita łańcucha o jedno miejsce w prawo (najstarszy bit pozostaje niezmienny)

dana = dana >> 1; //dla typów ze znakiem



Operacje bitowe

Pakowanie / rozpakowywanie danych

- Operacje bitowe można wykorzystać do pakowania i rozpakowywania danych

```
1 unsigned char dzien = 1, miesiac = 9;
2 unsigned short int rok = 2018;
3 unsigned int spakowane = 0;
4
5 //pakowanie
6 spakowane = ( rok << 16 ) | ( miesiac << 8 ) | dzien;
7
8 //rozpakowywanie
9 rok = ( spakowane >> 16 ) & 0xffff;
10 miesiac = (spakowane >> 8) & 0xff;
11 dzien = spakowane & 0xff;
12
13
```



Podstawowe operatory

Operatory logiczne

- dwuargumentowe

&& ||

lewa_strona <-> prawa_strona

- jednoargumentowe

!

<-> prawa_strona

A	B		A && B	A B	!A
false	false		false	false	true
false	true		false	true	true
true	false		false	true	false
true	true		true	true	false

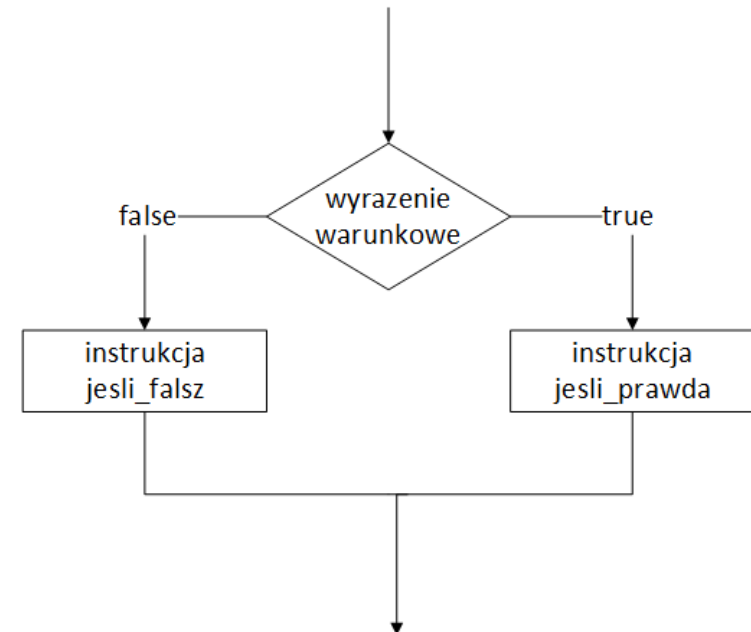
Instrukcja warunkowa if

if (*wyrażenie_warunkowe*)

instrukcja_jeśli_prawda;

else

instrukcja_jeśli_falsz;



```
1  if ( a != 0 )
2      a--;
3  else
4      a = 100;
5  //-----
6  if ( a )
7      a--;
8  else
9      a = 100;
10 //-----
11 if ( !a-- )
12     a = 100;
13
```



Instrukcja warunkowa if

```
1  if ( delta < 0 ) {  
2      //równanie nie posiada pierwiastków rzeczywistych  
3      ...  
4  } else if ( delta == 0 ) {  
5      //równanie posiada jeden pierwiastek rzeczywisty  
6      ...  
7  }  
8  else {  
9      //równanie posiada jeden pierwiastek rzeczywisty  
10     ...  
11 }  
12  
13 //-----  
14  
15 if ( delta < 0 ) {  
16     //równanie nie posiada pierwiastków rzeczywistych  
17     ...  
18 }  
19  
20 if ( delta == 0 ) {  
21     //równanie posiada jeden pierwiastek rzeczywisty  
22     ...  
23 }  
24  
25 if ( delta < 0 ) {  
26     //równanie posiada jeden pierwiastek rzeczywisty  
27     ...  
28 }  
29  
30
```

Operator warunkowy ?

wyrażenie_logiczne ? wyrażenie_jesli_prawda : wyrażenie_jesli_falsz;

- konstrukcja warunkowa z wykorzystaniem operatora warunkowego tworzy wyrażenie

b = b < 0 ? 0 : b;

b = b > 0 ? b : -b;

Konstrukcja warunkowa switch .. case

```
switch ( wyrażenie )  
{  
    case wartosc_1:  
        instrukcje_1;  
        break;  
    case wartosc_2:  
        instrukcje_2;  
        break;  
    ...  
    case wartosc_n:  
        instrukcje_n;  
        break;  
    default:  
        instrukcje;  
        break;  
}
```

Konstrukcja warunkowa switch .. case

```

1  #include <stdio.h>
2
3  int main(int argc, char const *argv[]) {
4      float a = 0, b = 1; char opcja = 0;
5
6      printf("Kalkulator v0.001\n");
7      printf(" A - dodawanie a + b\n");
8      printf(" B - dzielenie a / b\n");
9      printf(" Twój wybór? "); scanf("%c", &opcja);
10     printf(" Wprowadz a: "); scanf("%f", &a);
11     printf(" Wprowadz b: "); scanf("%f", &b);
12
13     switch ( opcja ) {
14         case 'A':
15         case 'a':
16             printf(" Wynik operacji a + b = %f\n", a + b);
17             break;
18
19         case 'B':
20         case 'b':
21             if ( b == 0 ) {
22                 printf(" Wynik operacji a / b jest nieokreslony\n");
23             } else {
24                 printf(" Wynik operacji a / b = %f\n", a / b);
25             }
26             break;
27
28         default:
29             printf(" ");
30             break;

```

Arytmetyka binarna i działania na bitach

Zadania :

- Napisz program , który wczytuje ze standardowego wejścia dwie liczby całkowite i wypisuje na standardowym wyjściu większą z nich (w przypadku gdy podane liczby są równe, program powinien wypisać którąkolwiek z nich).
- Napisz program , który wczytuje ze standardowego wejścia dwie liczby całkowite i wypisuje tą o większej wartości bezwzględnej.
- Napisz program , który wczytuje ze standardowego wejścia współczynniki równania kwadratowego z jedną niewiadomą i wypisuje na standardowym wyjściu wszystkie rozwiązania rzeczywiste tego równania lub odpowiednią informację w przypadku braku rozwiązań.

Arytmetyka binarna i działania na bitach

Podsumowanie :

- Operacje logiczne na bitach
- Operacje logiczne na liczbach binarnych i ciągach bitów
- Przesunięcia i rotacje
- Pakowanie i rozpakowywanie danych
- Operatory logiczne
- Operatory bitowe (binarne)
- Konstrukcja warunkowa if
- Operator warunkowy ?
- Konstrukcja warunkowa switch ... case

Konstrukcje iteracyjne

Konstrukcje iteracyjne

Plan zajęć :

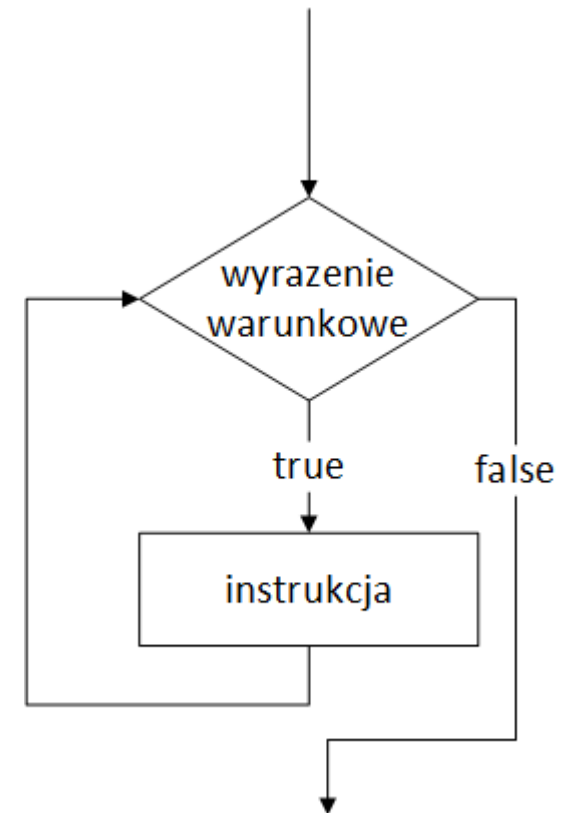
- While
- Do ... while
- For
- Słowa kluczowe break i continue
- Słowo kluczowe goto

Konstrukcja iteracyjna while

while (wyrażenie_warunkowe)

instrukcja;

```
1  int a = 10;  
2  while ( a-- )  
3      printf ("%d \n", a);  
4  // 9 8 7 6 5 4 3 2 1 0  
5  //-----  
6  while ( 1 ) {  
7      ...  
8  }  
9  
10
```



Konstrukcja iteracyjna do ... while

do

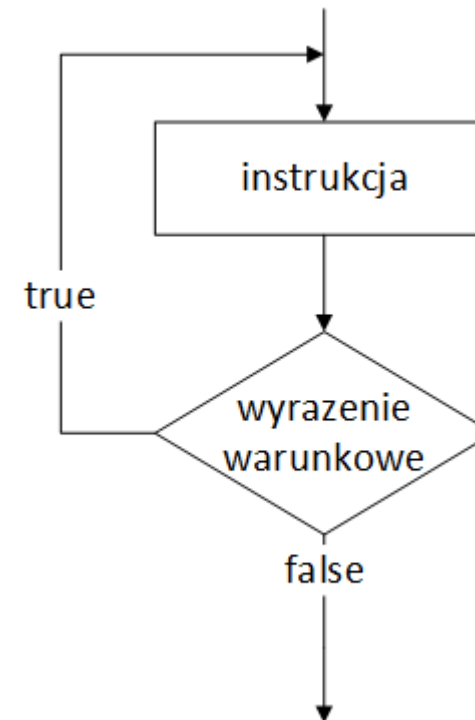
instrukcja;

while (wyrażenie_warunkowe);

```

1  int a = 10;
2  do
3      printf (" %d ", a);
4  while ( a-- );
5  // 10 9 8 7 6 5 4 3 2 1
6  //-----
7  char a = 0;
8  do{
9      printf ("Opcja?\n");
10     scanf ("%c", &a);
11 } while( a != 'x' );
12

```



Konstrukcja iteracyjna for

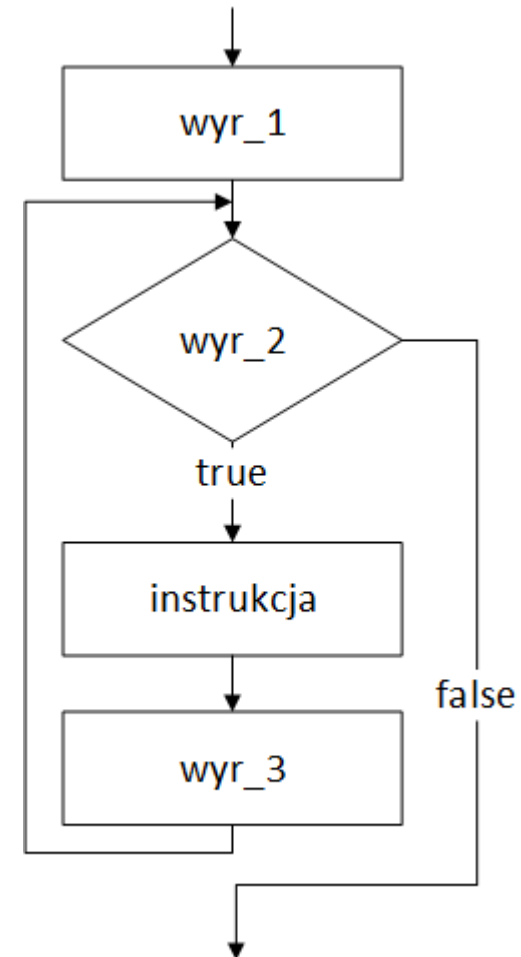
for (wyr_1; wyr2; wyr3)

instrukcja;

```

1  int a = 0;
2  for(a = 0; a <= 10; a++)
3      printf ("%d ", a);
4
5  // 0 1 2 3 4 5 6 7 8 9 10
6  //-----
7  int a = 0, b = 0;
8  for(a = 0, b = 5; a <= 10 && b >=0; a++, b--)
9      printf ("%d,%d ", a, b);
10 // 0,5 1,4 2,3 3,2 4,1 5,0
11
12
13

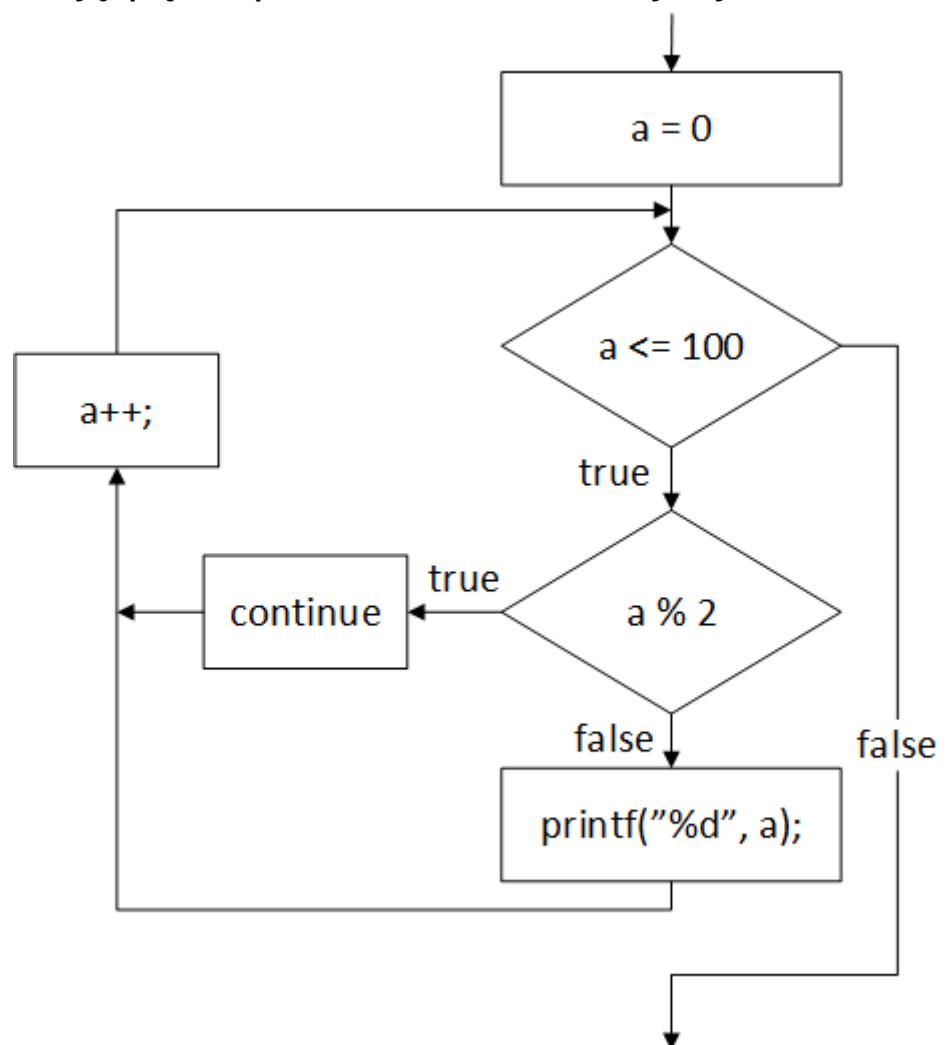
```



Słowo kluczowe continue

- zatrzymuje aktualnie wykonywaną iterację pętli i przechodzi do kolejnej iteracji

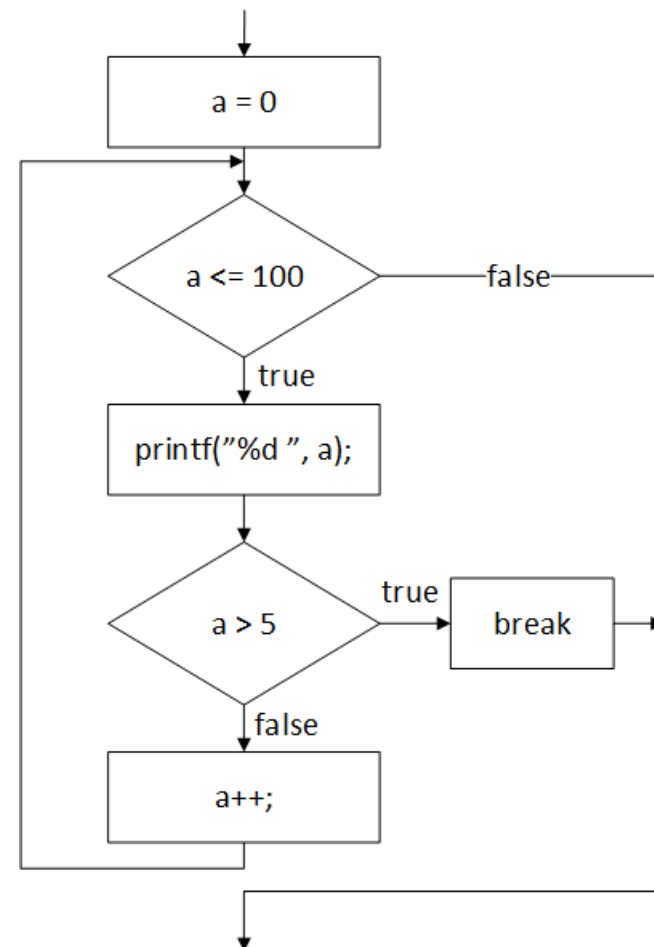
```
1  int a = 0;  
2  for(a = 0; a <= 10; a++){  
3      if (a % 2 ) continue;  
4      printf ("%d ", a);  
5  }  
6  
7  // 0 2 4 6 8 10  
8  
9
```



Słowo kluczowe break

- zatrzymuje aktualnie wykonywaną iterację pętli i opuszcza pętlę
- dotyczy konstrukcji: for, while, do ... while, dwitch ... case

```
1  int a = 0;
2  for(a = 0; a <= 100; a++){
3      printf ("%d ", a);
4      if (a > 5 ) break;
5  }
6
7  // 0 1 2 3 4 5 6
8
```



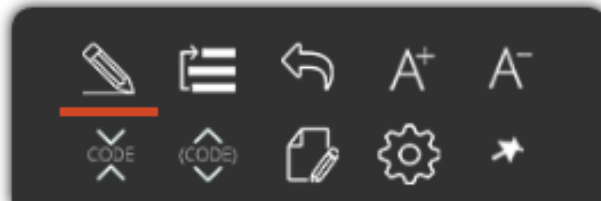
Słowo kluczowe goto

- Odpowiada instrukcji assemblerowej skoku bezwarunkowego w dowolne miejsce (wymaga jedynie określenia punktu skoku – etykiety)
- ***Nie należy używać goto, gdy dostępna jest inna równoważna, standardowa konstrukcja***

```

1  if ( a != 0 ) goto dalej;
2  blad = 1;
3  goto koniec;
4  dalej:
5  c /= a;
6  koniec:
7
8

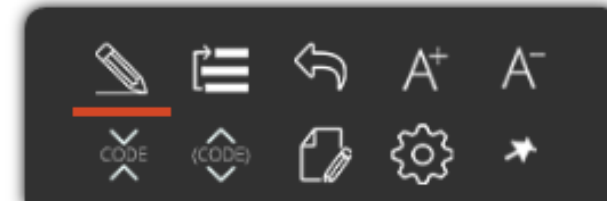
```



```

1  if ( a != 0 )
2      c /= a;
3  else
4      blad = 1;
5
6

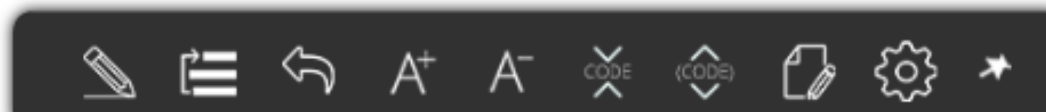
```



```

1  int a = 0, b = 0;
2  for (a = 0; a < 100; a++)
3  {
4      for (b = 0; b < 100; b++) {
5          ...
6          if ( b == 10 && a == 99 ) goto wyjscie;
7      }
8  wyjscie:

```



Konstrukcje iteracyjne

Zadania :

- Napisz program wczytujący ze standardowego wejścia dwie dodatnie liczby całkowite n i m , i wypisujący w kolejnych wierszach na standardowym wyjściu wszystkie dodatnie wielokrotności n mniejsze od m .
- Napisz program, który wczytuje ze standardowego wejścia nieujemną liczbę całkowitą n i wypisuje na standardowym wyjściu liczbę $n!$.
- Napisz program, który wczytuje ze standardowego wejścia nieujemną liczbę całkowitą n i wypisuje na standardowym wyjściu wszystkie liczby pierwsze mniejsze lub równe n .

Konstrukcje iteracyjne

Podsumowanie :

- While
- Do ... while
- For
- Słowa kluczowe break i continue
- Słowo kluczowe goto

Złożone typy danych

Złożone typy danych

Plan zajęć :

- Unie
- Struktury
- Tablice jednowymiarowe i wielowymiarowe
- Typ wyliczeniowy enum
- Operator sizeof()

Tablice

- W języku C można w prosty sposób gromadzić zmienne tego samego typu w postaci tablicy zmiennych
- Każda zmienna w tablicy ma swój indeks, począwszy od indeksu zerowego dla zmiennej znajdującej się na początku tablicy

Deklaracja tablicy:

`typ_zmiennej nazwa_tablicy[liczba_elementów];`

`int zmienne[10];`

- Możliwa jest inicjalizacja elementów tablicy w trakcie jej deklaracji

`const float wspolczynniki[3] = {1.23, 1.34, 1.55};`

Operator indeksowania tablicy []

- Dostęp do poszczególnych elementów tablicy uzyskujemy z wykorzystaniem operatora []

nazwa_tablicy[indeks_tablicy]

- Argumentem operatora indeksowania tablicy może być wyłącznie typ stałoprzecinkowy
- W języku C nie ma kontroli zakresu indeksu tablicy

```
1 int values[10]; //deklaracja tablicy złożonej z 10 elementów typu int
2 double tst[30]; //deklaracja tablicy złożonej z 30 elementów typu double
3 char text[6]; //deklaracja tablicy złożonej z 6 elementów typu char
4
5 values[3] = 21; //przypisanie czwartemu elementowi tablicy wartości 21
6 values[2] = values[1] + 2;
7 printf("%f", values[7]);
8
9
```



Przykład – wyszukiwanie największego elementu w tablicy

```
1  #include <stdio.h>
2  #define SIZE 10
3
4  int main(int argc, char const *argv[]) {
5      double tab[SIZE] = {1.1 , 4.3 , 6.7 , -3.4, 2.4 , 0.1 , 4.2 , -1.2, 0.01 , 1.3};
6      double val;
7      int n , pos;
8
9      val = tab[0] ;
10     pos=0;
11     for ( n = 1; n < SIZE; n++){
12         if( val < tab[n] ) {
13             val = tab[n];
14             pos = n;
15         }
16     }
17
18     printf("Najwieksza liczba: %f, na pozycji: %d\n" , val , pos ) ;
19
20     return 0;
21 }
22
23
24
```



Tablice wielowymiarowe

- Język C umożliwia definiowanie i korzystanie z tablic wielowymiarowych, deklaracja ma następującą postać:

`typ_zmiennej nazwa_tablicy[n1][n2]...[nk];`

Przykład:

```
int macierz[6][10];
```

Pytanie:

Czy jest to tablica zawierająca 6 wierszy i 10 kolumn, czy też jest to tablica zawierająca 6 kolumn i 10 wierszy?

Tablice wielowymiarowe – kolejność zapisu

- Rozważmy następujące deklaracje:

1. `int a;`
2. `int b[10];`
3. `int c[6][10];`
4. `int d[3][6][10];`

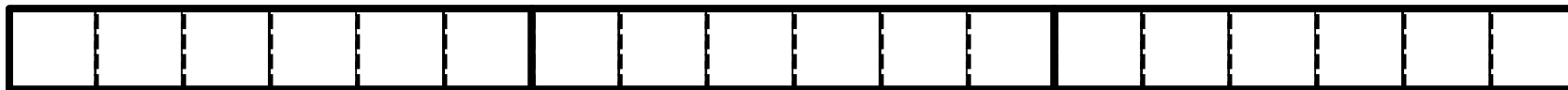
1. `a` jest pojedynczą liczbą
2. `b` jest wektorem zawierającym 10 elementów
3. `c` jest wektorem zawierającym 6 elementów, przy czym każdy z tych elementów zawiera 10 elementowy wektor
4. `d` jest tablicą trzejelementową, w której każdy element jest tablicą sześcieelementową, składającą się z tablic 10 elementowych.

Tablice wielowymiarowe – kolejność zapisu

```
int tablica[3];
```



```
int tablica[3][6];
```



```
int tablica[3][6] = { {1, 2, 3, 4, 5, 6},  
                     {4, 5, 6, 7, 8, 9},  
                     {10, 11, 12, 13, 14, 15} };
```

Przykład – tablica wielowymiarowa

```
1  #include <stdio.h>
2  #define SIZE 4
3
4  int main(int argc, char const *argv[]) {
5      double tab [SIZE][SIZE];
6      int n , k ;
7      for ( n = 0; n < SIZE ; n++)
8          for ( k = 0; k < SIZE ; k++)
9              tab[n][k] = k == n;
10
11     for ( n = 0; n < SIZE ; n++) {
12         for ( k = 0; k < SIZE ; k++)
13             printf ("%2.1f\t" , tab[n][k]);
14         printf ("\n" ) ;
15     }
16
17     return 0;
18 }
19
20
21
```



Struktury

- Struktura jest typem złożonym z ciągu zmiennych różnych typów.
- Zmienne te noszą nazwę składowych lub pól struktury, są one przechowywane w jednym spójnym bloku pamięci.
- Struktury są deklarowane poprzez wymienienie listy składników w nich zawartych, lista ta zawiera nazwę i typ każdego ze składników.

```

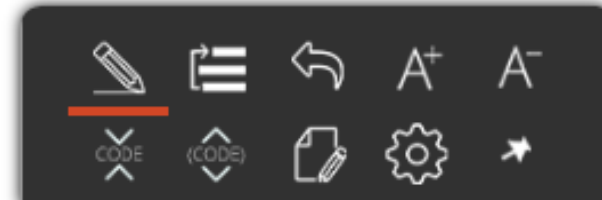
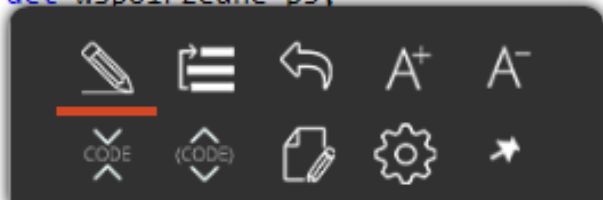
1  struct
2  {
3      double x;
4      double y;
5  } p1;
6  //-----
7  struct wspolrzedne
8  {
9      double x;
10     double y;
11 };
12
13 struct wspolrzedne p2;
14 struct wspolrzedne p3;
15

```

```

1  typedef struct wspolrzedne
2  {
3      double x;
4      double y;
5  } Punkt;
6
7  Punkt p4;
8  Punkt punkty[20];
9

```



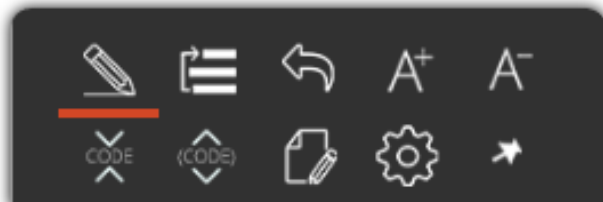
Struktury

- Składowymi struktur mogą być zmienne skalarne, tablice, wskaźniki oraz nawet inne struktury.
- Dostęp do składowych struktury odbywa się za pomocą operatora

```

1 struct wspolrzedne
2 {
3     double x;
4     double y;
5 };
6 struct prostokat
7 {
8     struct wspolrzedne LGwierzcholek;
9     struct wspolrzedne PDwierzcholek;
10 };
11 struct kolo
12 {
13     struct wsp {double x, y; };
14     int srednica;
15 } k1;
16

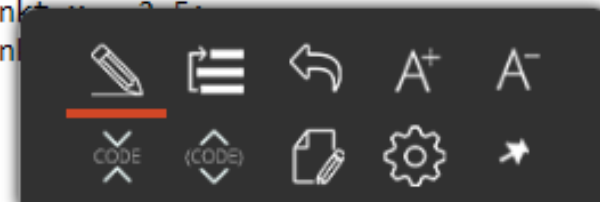
```



```

1 struct wspolrzedne
2 {
3     double x;
4     double y;
5 } punkt;
6 struct prostokat
7 {
8     struct wspolrzedne LGwierzcholek;
9     struct wspolrzedne PDwierzcholek;
10 } pr1;
11
12 struct prostokat pr2 = { {1.1, 2.2}, {3.3, 4.4} };
13 pr1 = pr2;
14
15 punkt = { 2.5, 3.5 };
16
17

```



Przykład – struktury

```
1  #include <stdio.h>
2  #include <math.h>
3
4  struct Point
5  {
6      float x, y;
7  };
8
9  int main()
10 {
11     printf("Program ten oblicza odleglsc pomiedzy dwoma punktami w ukkladzie wspolrzednych\n");
12     struct Point p1 = {.y = 0, .x = 2};
13     struct Point p2 = {.x = 20};
14     float distance = 0.0;
15     printf("Podaj wspolrzedne pierwszego punktu (x, y): ");
16     scanf("%f %f", &(p1.x), &(p1.y));
17     printf("Podaj wspolrzedne pierwszego punktu (x, y): ");
18     scanf("%f %f", &(p2.x), &(p2.y));
19     distance = sqrt( ( p2.x - p1.x ) * ( p2.x - p1.x ) + ( p2.y - p1.y ) * ( p2.y - p1.y ) );
20     printf ("P1 = (%f, %f)\n", p1.x, p1.y);
21     printf ("P2 = (%f, %f)\n", p2.x, p2.y);
22     printf("odleglosc = %f\n", distance);
23
24     return 0;
25 }
26
```



Unie

- Unia umożliwia zdefiniowanie zmiennej, która będzie reprezentowana przez różne typy danych
- Dostęp do zmiennej zdefiniowanej jako unia możliwy jest poprzez różne typy danych
- Zdefiniowane elementy unii współdzielą ten sam obszar pamięci
- Rozmiar unii jest największym rozmiarem elementu unii

```
1 union dana {  
2     short int slowo;  
3     struct  
4     {  
5         char mlodszy_bajt;  
6         char starszy_bajt;  
7     } dwa_bajty;  
8 };  
9
```



Przykład – unie

```
1  #include <stdio.h>
2
3  union dana {
4      short int slowo;
5      struct
6      {
7          char mlodsz_bajt;
8          char starszy_bajt;
9      } dwa_bajty;
10 };
11
12 int main()
13 {
14     union dana zmienna, zmienna2;
15
16     //zmienna.dwa_bajty.mlodsz_bajt = 0x34;
17     //zmienna.dwa_bajty.starszy_bajt = 0x12;
18     zmienna.slowo = 0x1234;
19
20     //0x3412
21     zmienna2.dwa_bajty.starszy_bajt = zmienna.dwa_bajty.mlodsz_bajt;
22     zmienna2.dwa_bajty.mlodsz_bajt = zmienna.dwa_bajty.starszy_bajt;
23
24
25     printf("zmienna = 0x%04x\n", zmienna.slowo);
26     printf("zmienna2 = 0x%04x\n\n", zmienna2.slowo);
27
28     return 0;
29 }
30
```

enum – typ wyliczeniowy

- Typ wyliczeniowy stosowany jest kiedy istnieje potrzeba zdefiniowania listy stałych reprezentujących dane w sposób opisowy, np. zbiór kolorów, kierunków, czynności, itp.
- Zmienna danego typu wyliczeniowego może przyjmować tylko jedną z podanych w definicji wartości
- Typ wyliczeniowy funkcjonalnie odpowiada typowi **int**, a poszczególne jej wartości są równoważne kolejnym stałym całkowitym, co można zadeklarować w sposób jawny

```
1 enum Colors { RED, GREEN, BLUE }; //definicja typu wyliczeniowego Colors
2 enum Directions { Left, Right, Up = 3, Down }; //stałe 0, 1, 3, 4
3
4 enum Colors color;
5 color = RED;
6 color = GREEN;
7
8
```



sizeof()

- Operator zwracający rozmiar wskazanego typu danych lub zmiennej
- Podany rozmiar zwracany jest w bajtach (w przypadku większości kompilatorów)
- W przypadku podania jako argumentu operatora nazwy tablicy, zwrócony zostanie jej rozmiar

```
1 unsigned int rozmiar = 0;  
2 rozmiar = sizeof(long);  
3 rozmiar = sizeof(moja_zmienna);  
4 rozmiar = sizeof(moja_struktura);  
5  
6
```



Złożone typy danych

Zadania :

- Napisz program wczytujący wartości do tablicy liczb całkowitych o wymiarach 4x4, wyświetlający tę tablicę oraz umożliwiający zamianę wartości minimalnej z maksymalną.
- Napisz program wyznaczający parametry funkcji liniowej na podstawie współrzędnych dwóch dowolnych punktów przez które ona przechodzi

Wskaźniki

Wskaźniki

Plan zajęć :

- Zmienna wskaźnikowa
- Operator referencji &
- Operator dereferencji *
- Operator strzałki ->
- Arytmetyka wskaźników

Zmienna wskaźnikowa

- Zmienna wskaźnikowa zawiera informacje o miejscu przechowywania innej zmiennej w postaci jej adresu
- Zmienna wskaźnikowa umożliwia pośredni dostęp do innej zmiennej, której adres przechowuje
- Niebezpieczeństwo wykorzystania zmiennych wskaźnikowych związane jest z możliwością pośredniego dostępu do nieistniejących w pamięci zmiennych lub próby dostępu do obszarów pamięci, które są niewłaściwe

```
1 //deklaracja zmiennej wskaźnikowej
2 int *a;
3 const int *a;
4 int *const a;
5 const int *const a;
6
7 //deklaracja tablicy jest deklaracją zmiennej wskaźnikowej
8 //z przypisanym jej adresem pierwszego elementu tablicy (którego indeks wynosi 0)
9 int tablica[20];
10 int* tab;
11 tab = tablica;
12
13
```

Operator referencji &

- Operator referencji & umożliwia uzyskanie adresu zmiennej podanej jako argument operatora

```
1  int a = 10;  
2  int *b = &a;  
3  
4  int tablica[10];  
5  int *element;  
6  element = &tablica[3];  
7  element = tablica + 3;  
8  
9
```



Operator dereferencji *

- Operator dereferencji * umożliwia dostęp do zmiennej, której adres przechowywany jest w zmiennej wskaźnikowej
- Indeksy tablicy są w rzeczywistości ukrytymi wyrażeniami dereferencji

```
1  int a = 10;  
2  int *b = &a;  
3  *b = 0;  
4  
5  int tablica[10];  
6  int *element;  
7  element = &tablica[3];  
8  *element = 0;  
9  
10 tablica[7] = 10;  
11 *(tablica + 7) = 10;  
12  
13
```



Operator strzałki ->

- Aby odwołać się do składników struktury, dysponując wskaźnikiem do niej, w pierwszym kroku należy zastosować dereferencję na wskaźniku (dzięki czemu otrzymuje się w jej wyniku strukturę) a następnie skorzystać z operatora kropki w celu wybrania składnika
- W języku C dostępny jest dodatkowy operator realizujący to zadanie – operator strzałki ->

```
1  struct Point
2  {
3      float x, y;
4  };
5
6  struct Point p1 = {.y = 0, .x = 2};
7  struct Point *ptr = &p1;
8
9  ptr->x = 2.5;
10 ptr->y = 5.2;
11
12 (*ptr).x = 2.5;
13 (*ptr).y = 5.2;
14
15
```

wskaźnik **NULL**

- Informacja, że wskaźnik na nic nie wskazuje
- Formalnie jest to wartość zero
- Fizycznie pod adresem zerowym może znajdować się inna wartość, przekształceniem pomiędzy wewnętrzną wartością i zerem zajmuje się kompilator

```
1  int *b = NULL;
2  int *element = NULL;
3
4
5  if ( a ){
6      ...
7  }
8
9  if ( element != NULL ) {
10     ...
11 }
12
13
```



Niezainicjowane i nieprawidłowe wskaźniki

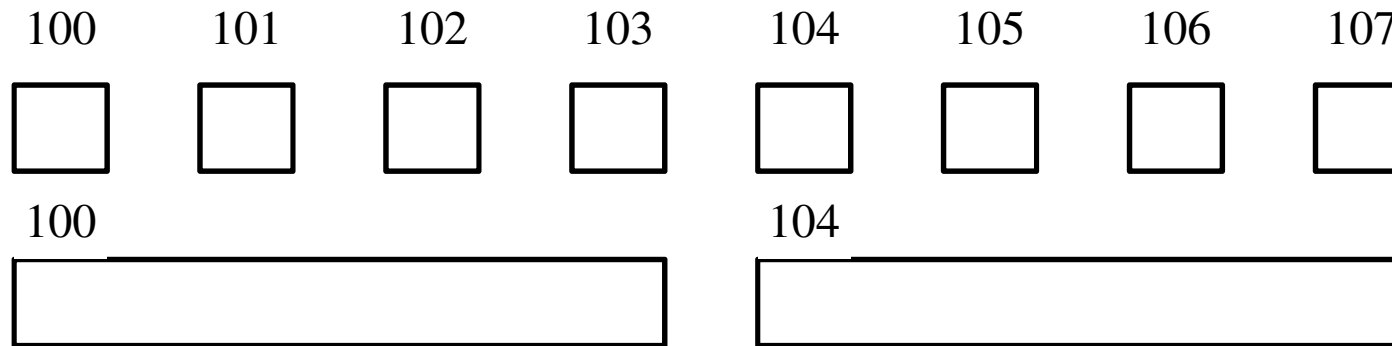
- W przypadku, gdy wartością początkową będzie niedozwolony adres to przypisanie doprowadzi do błędu powodującego zakończenie programu
- NAJGORSZY PRZYPADEK WYSTĄPI GDY WSKAŹNIK BĘDZIE ZAWIERAŁ DOZWOLONY ADRES

```
1  int *a;  
2  .  
3  .  
4  .  
5  
6  *a = 12;  
7
```



Przykład

- Rozważmy przykładową mapę pamięci

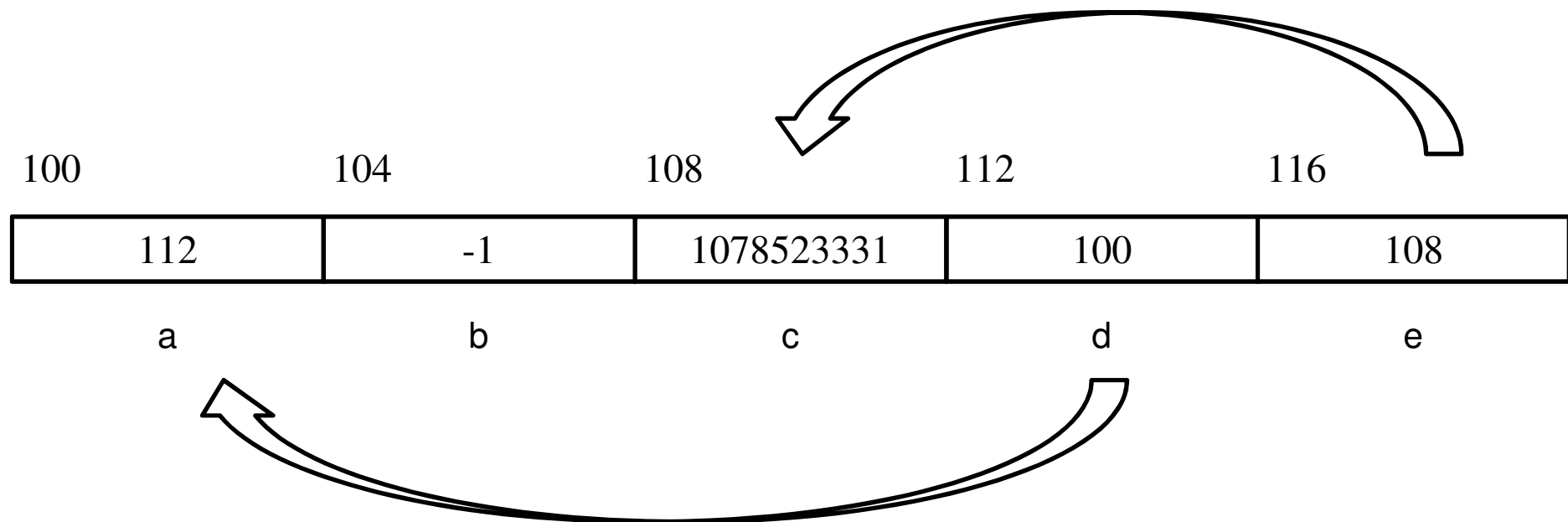
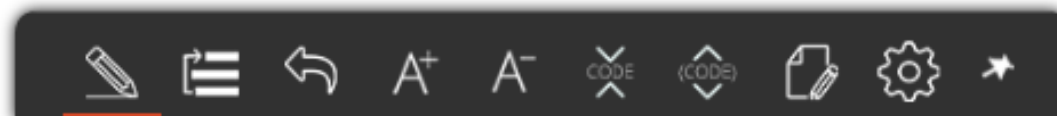


- Na rysunku pokazano zawartość poszczególnych komórek pamięci (wartości zinterpretowano jako liczbę całkowitą ze znakiem)

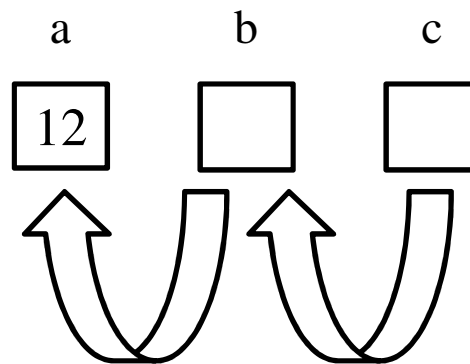
100	104	108	112	116
112	-1	1078523331	100	108
a	b	c	d	e

Przykład

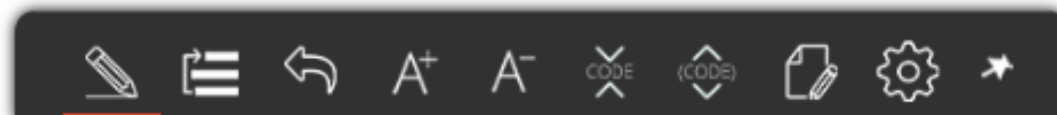
```
1 int a= 112, b = -1;  
2 float c = 3.14;  
3 int *d = &a;  
4 float *e &c;  
5  
6
```



Wskaźnik na wskaźnik



```
1  int a = 12;
2  int *b = &a;
3  int **c = &b;
4
5  a = 10;
6  *b = 10;
7  **c = 10;
8
```



Arytmetyka wskaźników

Arytmetyka wskaźników w C jest ograniczona do dokładnie dwóch postaci:

wskaźnik \pm wartość całkowita

- Postać ta jest zdefiniowana w standardzie języka tylko dla wskaźników wskazujących na element tablicy, daje ona wynik tego samego typu co wskaźnik

wskaźnik - wskaźnik

- Odjęcie jednego wskaźnika od drugiego jest dozwolone tylko wtedy, gdy oba wskaźniki wskazują na elementy tej samej tablicy
- Wynikiem odejmowania dwóch wskaźników jest typ ptrdiff_t, który jest wartością całkowitą ze znakiem, jego wartością jest odległość (mierzona w elementach tablicy) pomiędzy dwoma wskaźnikami

Wskaźniki

Podsumowanie :

- Zmienna wskaźnikowa
- Operator referencji &
- Operator dereferencji *
- Operator strzałki ->
- Arytmetyka wskaźników

Modularyzacja programu – funkcje, podział na pliki

Modularyzacja programu – funkcje, podział na pliki

Plan zajęć :

- Deklaracja funkcji
- Definicja funkcji
- Argumenty funkcji
- Przeciążanie funkcji
- Wskaźnik do funkcji
- Funkcja inline
- Przesłanki do utworzenia funkcji
- Projektowanie funkcji

Modularyzacja programu

- Modularyzacja programu jest procesem jego podziału na mniejsze fragmenty, z których każdy będzie spójny pod względem przyjętych kryteriów
- Kod źródłowy programu można podzielić na podprogramy, które będą wykorzystywane wielokrotnie w ramach całego programu
- Modularyzacja programu zmniejsza długość jego kodu wynikowego, poprawia czytelność kodu źródłowego i ułatwia weryfikację implementacji algorytmu
- Modularyzacja programu umożliwia tworzenie dużych, rozproszonych projektów programistycznych

Funkcje

- Funkcja w języku C stanowi formę podprogramu, może ona być wywoływana wielokrotnie z różnych miejsc w programie
- W przypadku odwoływania się funkcji do samej siebie mamy do czynienia z rekurencją
- Funkcja może realizować zaimplementowany w niej algorytm w oparciu o parametry przekazywane w postaci argumentów wejściowych oraz zwracać wyniki działania podprogramu
- W programowaniu strukturalnym funkcja traktowana jest jak „czarna skrzynka”
- Funkcja zwraca jedną wartość jako wynik albo nie zwraca żadnej wartości

Funkcje

Funkcja w języku C składa się z dwóch części

- Deklaracji funkcji:

```
typ_zwracany nazwa_funkcji(typ_arg1 nazwa_zm1, typ_arg1 nazwa_zm1, ...);
```

```
int suma (int a, int b );
```

- Definicji funkcji:

```
typ_zwracany nazwa_funkcji(typ_arg1 nazwa_zm1, typ_arg1 nazwa_zm1, ...) {}
```

```
int suma (int a, int b ) {
```

```
    return a + b;
```

```
}
```

Deklarację funkcji można pominąć, jeżeli jej definicja jest widoczna w miejscu wywołania.

Funkcje – podział na pliki

- Deklaracje funkcji umieszcza się w pliku nagłówkowym (*.h)
- Definicje funkcji umieszcza się w pliku źródłowym (*.c)
- Deklaracja funkcji może być tylko jedna
- Przy stosowaniu plików nagłówkowych stosuje się konstrukcję preprocesora

```
#ifndef NAGLOWEK_H
```

```
#define NAGLOWEK_H
```

```
    ...deklaracje funkcji – plik naglowek.h
```

```
#endif
```

- W przypadku wielokrotnego dołączenia tego samego pliku nagłówkowego, deklaracja funkcji będzie tylko jedna

Funkcje – instrukcja return

- Instrukcja return przerywa wykonywanie funkcji i zwraca do wywołującej funkcji wartość obliczonego wyrażenia (wartość tą umieszcza się po słowie return)
- W sytuacji gdy typ obliczonej wartości wyrażenia nie jest zgodny z zadeklarowanym typem zwracany przez funkcję przeprowadzana jest automatyczna konwersja
- W przypadku funkcji która nie zwraca żadnej wartości, instrukcja return umożliwia wcześniejsze zakończenie jej wykonywania (wcześniejsze opuszczenie funkcji)

```
1  int power (int base, int n ) {  
2      if ( !base ) return 0;  
3  
4      int p = 1;  
5      for ( int i = 1; i <= n, i++)  
6          p *= base;  
7  
8      return p;  
9  }  
10  
11
```



Funkcje – parametry

- W języku C parametry przekazywane są do funkcji przez wartość, oznacza to że parametry występujące w deklaracji funkcji są de facto jej zmiennymi lokalnymi, które przyjmują wartości początkowe podane w trakcie wywołania funkcji.
- Modyfikacja parametrów wewnątrz funkcji nie wpływa na wartość zmiennych, będących parametrami wywołania funkcji.
- Aby mieć możliwość modyfikacji zmiennej wewnątrz funkcji, należy jako parametr przekazać wskaźnik do tej zmiennej

```
1 void swap (int &a, int &b ) {  
2     int temp = *a;  
3     *a = *b;  
4     *b = temp;  
5 }  
6  
7
```



Funkcje – typ void

- Identyfikuje funkcje, które nie pobierają argumentów, bądź nie zwracają wartości instrukcją return

```
1  int generator_liczby ( void ){  
2      return rand();  
3  }  
4  
5  void wyswietl_liczbe (int liczba ) {  
6      printf ( "%d\n", liczba );  
7  }  
8  
9
```



Funkcje – argumenty domyślne

- Możliwe jest przypisanie argumentom funkcji wartości domyślnych
- typ_zwracany nazwa_funkcji(typ_arg nazwa = wart_domyslne ...) { ... }
- Domyślne argumenty mogą zostać pominięte przy wywołaniu funkcji
 - Domyślne argumenty muszą się znajdować na końcu listy parametrów danej funkcji

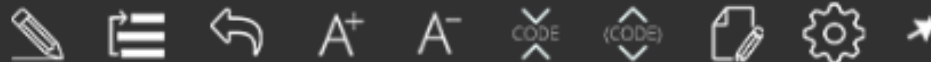
```
1  const char witaj[] = "Czesc, mam na imie ";
2
3  void wypisz_powitanie (const char *imie, const char powitanie = witaj ) {
4      printf("%s%s", powitanie, imie)
5  }
6
7  //-----
8  wypisz_powitanie ( "Andrzej" );
9  //Czesc, mam na imie Andrzej
10
11 //-----
12 wypisz_powitanie ( "Andrzej", "Czesc, jestem" );
13 //Czesc, jestem Andrzej
14
```



Przeciążanie funkcji

- Funkcja może mieć różne implementacje, zależne od typów argumentów oraz ich liczby
- Należy zachować szczególną ostrożność przy przeciążaniu funkcji z argumentami domyślnymi

```
1 float pole_powierzchni ( float promien ){  
2     return 3.14 * promien * promien;  
3 }  
4  
5 float pole_powierzchni ( float a, float b ){  
6     return a * b;  
7 }  
8  
9
```



Funkcje inline

- Deklaracja funkcji może być poprzedzona kwalifikatorem inline
- Taka funkcja będzie wstawiana w miejsce jej użycia, zamiast jej wywołania
- Większe możliwości optymalizacyjne
- Kompilator wymaga dostępu do treści jej definicji, w przeciwnym wypadku ignoruje kwalifikator inline
- W przypadku bibliotek taka funkcja musi być zdefiniowana w pliku nagłówkowym

Wskaźnik do funkcji

- Nazwa funkcji reprezentuje zmienną wskaźnikową z przypisanym adresem funkcji (jej położeniem w pamięci)
- Możliwe jest zadeklarowanie zmiennej wskaźnikowej odwołującej się do określonego typu funkcji i przypisanie jej adresu istniejącej w pamięci funkcji

```
1  int suma ( int a, int b ) { return a + b; }
2  int roznica ( int a, int b ) { return a - b; }
3
4  int ( *a )( int a, int b );
5  a = suma;
6  printf ( "%d", a( 1, 2) );
7  void *b;
8  b = ( void* ) roznica;
9  a = ( int (*) (int, int) )b;
10 printf ( "%d", a( 1, 2) );
11 printf ( "%d", ( (( int (*) (int, int) )b)(2,1) ) );
12
13 //3
14 //-1
15 //1
16
17
18
```



Przesłanki utworzenia funkcji

- Redukowanie złożoności
- Wprowadzanie jasnych abstrakcji pośrednich
- Zapobieganie powtórzeniom kodu
- Ukrywanie sekwencji
- Ukrywanie operacji wskaźnikowych
- Zwiększanie przenośności kodu
- Upraszczanie złożonych testów logicznych
- Zwiększanie wydajności

Przesłanki utworzenia funkcji

- Izolowanie złożoności
- Ukrywanie szczegółów implementacji
- Ograniczanie zasięgu skutków zmian
- Ukrywanie danych globalnych
- Tworzenie scentralizowanych punktów kontroli
- Ułatwianie ponownego użycia kodu
- Wykonywanie określonej refaktoryzacji

Funkcje – zagadnienia kluczowe

- Funkcje powinny być małe
 - Funkcje powinny być mniejsze niż są
 - Funkcje powinny wykonywać jedną operację, powinny robić to dobrze, powinny robić tylko to
 - W funkcji powinien występować tylko jeden poziom abstrakcji
 - Nazwa powinna dokładnie opisywać przeznaczenie funkcji (nie należy obawiać się konstruowania długich nazw)
-
- Funkcje nie powinny zwracać wskaźników do zmiennych lokalnych

Projektowanie funkcji

- opis w pseudokodzie w formie komentarza
- implementacja w oparciu o pseudokod
- w efekcie otrzymujemy poprawną funkcję zawierającą sensowne komentarze stanowiące jej dokumentację

Modularyzacja programu – funkcje, podział na pliki

Zadania :

- Napisz program, który wczytuje ze standardowego wejścia liczbę całkowitą n i wypisuje na standardowym wyjściu jej wartość bezwzględną, użyj w tym celu „własnoręcznie” skonstruowanej funkcji:

int w_bezwzgledna (int wartosc);

- Napisz program, który wczytuje ze standardowego wejścia nieujemną liczbę całkowitą n i wypisuje na standardowym wyjściu liczbę $n!$, skonstruuj w tym celu funkcję:

int silnia (int wartosc);

- Napisz program wyświetlający na standardowym wyjściu tabliczkę mnożenia, skonstruuj w tym celu funkcję:

void t_m (int x_min, int y_min, int x_max, int y_max);

Modularyzacja programu – funkcje, podział na pliki

Podsumowanie :

- Deklaracja funkcji
- Definicja funkcji
- Argumenty funkcji
- Przeciążanie funkcji
- Wskaźnik do funkcji
- Funkcja inline
- Przesłanki do utworzenia funkcji
- Projektowanie funkcji

Łańcuchy tekstowe

Łańcuchy tekstowe

Plan zajęć :

- Pojęcie łańcucha tekstowego
- Tablica kodów ASCII
- Operacje na tekście
- Argumenty wywołania programu – parametry funkcji main()

Łańcuch tekstowy

- Łańcuch tekstowy to zbiór kodów znaków tekstowych przechowywanych zazwyczaj w spójnym obszarze pamięci
- Łańcuch tekstowy można przechowywać w tablicy zmiennych typu char
- Inicjalizacja tablicy złożonej ze zmiennych typu char może zostać dokonana przy pomocy łańcucha tekstowego
- Łańcuch tekstowy w języku C kończy się specjalnym kodem kończącym (\0 – wartość zero)

```
1  #include <stdio.h>
2
3  int main(int argc, char const *argv[]) {
4      char tekst[] = "Ala ma kota.";
5
6      printf("%s\n", tekst);
7      tekst[3] = 0;
8      printf("%s\n", tekst);
9
10     return 0;
11 }
12
13
```

Tablica kodów ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
80			,	f	„	...	†	‡	^	%	Š	<	Œ			
90		‘	’	“	”	•	—	—	~	™	š	>	œ			ÿ
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	­	®	¯
B0	°	±	²	³	´	µ	¶	·	,	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Łańcuch tekstowy

- Sprawdzenie jaki znak odpowiada kodowi ASCII o wartości 65

```
1  #include <stdio.h>
2
3  int main(int argc, char const *argv[]) {
4      char asciiVal = 65;
5      printf("%c\n", asciiVal);
6
7      return 0;
8  }
```

- Wyświetlenie kodu ASCII danego znaku

```
1  #include <stdio.h>
2
3  int main(int argc, char const *argv[]) {
4      char asciiChar = 'A';
5      printf("%d\n", asciiChar);
6
7      return 0;
8  }
```

Zmiana wielkości liter w łańcuchu znakowym

```
1  void male_litery(char *ciag)
2  {
3      while (*ciag)
4      {
5          if ('A' <= *ciag && *ciag <= 'Z')
6          {
7              *ciag += ('a' - 'A');
8          }
9          ciag++;
10     }
11 }
12
13 void wielkie_litery(char *ciag)
14 {
15     while (*ciag)
16     {
17         if ('a' <= *ciag && *ciag <= 'z')
18         {
19             *ciag -= ('a' - 'A');
20         }
21         ciag++;
22     }
23 }
24
```



Funkcja main() – parametry wywołania programu

- Argumenty funkcji main – dwa równoważne sposoby

`int main (int argc, char* argv[])`

`int main (int argc, char** argv)`

- Zawartość tablicy argv

argv[0] – wskaźnik do nazwy programu

argv[1], . . . , argv[argc - 1] – wskaźniki do kolejnych argumentów wywołania programu

argv[argc] – wskaźnik NULL

```
1  #include <stdio.h>
2
3  int main(int argc, char const *argv[]) {
4      int i;
5      printf("Podano %d argumentow:\n", argc);
6      for (i = 0; i < argc; i++) {
7          printf("Argument #%d\t-\t%s\n", i, argv[i] );
8      }
9      return 0;
10 }
11
```

Funkcja main() – parametry wywołania programu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void usage(char *program_name) {
5      printf("Sposob uzycia: %s <komunikat> <#powtorzen>\n", program_name);
6      exit(1);
7  }
8
9  int main(int argc, char const *argv[]) {
10     int i, count;
11
12     if(argc < 3)
13         usage(argv[0]);
14
15     count = atoi(argv[2]);
16     printf("Powtarzam %d razy..\n", count);
17
18     for( i = 0; i < count; i++)
19         printf("%3d - %s\n", i, argv[1]);
20     return 0;
21 }
22
23
```



Łańcuchy tekstowe

Zadania :

Napisz następujące funkcje:

- Obliczającą ilość znaków w łańcuchu
- Porównującą 2 łańcuchy znakowe pod kątem zawartości
- Dokonującą połączenia 2 łańcuchów znakowych

Łańcuchy tekstowe

Podsumowanie :

- Pojęcie łańcucha tekstowego
- Tablica kodów ASCII
- Operacje na tekście
- Argumenty wywołania programu – parametry funkcji main()

Operacje na plikach

Operacje na plikach

Plan zajęć :

- pliki o dostępie sekwencyjne
- pliki o dostępie swobodnym
- pliki tekstowe
- pliki binarne

Pliki o dostępie sekwencyjnym (ang. sequential-access)

Dane z takiego pliku należy odczytywać w takiej kolejności, w jakiej zostały zapisane

Z plikiem o dostępie sekwencyjnym robi się tylko trzy rzeczy:

- Tworzy się go
- Odczytuje się z niego dane
- Dodaje się do niego dane

```
1  #include <stdio.h>
2
3  FILE *fptr; // definicja wskaźnika do pliku
4  int main()
5  {
6      fptr = fopen("c:\\cprograms\\cdata.txt", "w");
7      // główna część programu
8      fclose (fptr);
9      return 0;
10 }
11
```

Pliki o dostępie sekwencyjnym – podstawowe funkcje

- `fopen()`
- `fclose()`
- `fprintf()`
- `fputs()`
- `fgets()`
- `feof()`

Podstawowe tryby otwarcia pliku funkcji `fopen()`

- „w” – tryb zapisu, w którym tworzony jest nowy plik
- „r” – tryb odczytu istniejącego pliku
- „a” – tryb, w którym dane są dodawane na końcu pliku lub tworzony jest nowy plik, jeśli podany plik nie istnieje

Pliki o dostępie sekwencyjnym - przykład

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  FILE * fptr;
5
6  int main()
7  {
8      fptr = fopen("plik.txt","a");
9      if (fptr == 0)
10     {
11         printf("Bład otwierania pliku!\n");
12         exit (1);
13     }
14
15     fprintf(fptr, "\nNapis dodawany na koncu pliku.\n");
16
17     fclose(fptr);
18
19     return(0);
20 }
21
22
```



Pliki o dostępie sekwencyjnym - przykład

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  FILE * fptr;
5
6  int main()
7  {
8      char fileLine[100];
9      fptr = fopen("plik.txt","r");
10
11     if (fptr != 0)
12     {
13         while (!feof(fptr))
14         {
15             fgets(fileLine, 100, fptr);
16             if (!feof(fptr))
17             {
18                 puts(fileLine);
19             }
20         }
21     }
22     else
23     {
24         printf("\nBlad otwierania pliku!\n");
25     }
26
27     fclose(fptr);
28
29     return(0);
```

Podstawowe zasady podczas pracy z plikami

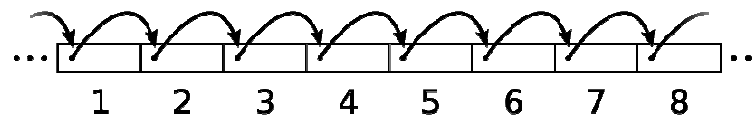
- Przed użyciem plik należy otworzyć za pomocą funkcji `fopen()`
- Po zakończeniu pracy z plikiem należy go zamknąć przy użyciu funkcji `fclose()`
- Podczas odczytywania danych z pliku należy używać funkcji `feof()`, aby uniemożliwić programowi odczyt danych spoza granic pliku
- Po otwarciu pliku nie posługujemy się jego nazwą tylko wskaźnikiem do pliku

Pliki o dostępie swobodnym (ang. random-access)

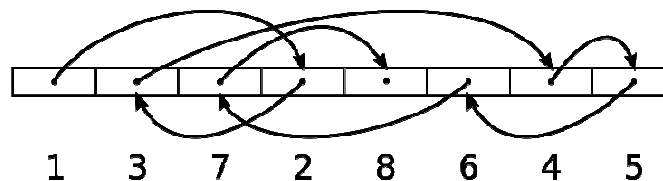
W przypadku takiego pliku informacje można odczytywać i zapisywać w dowolnie wybranych miejscach

- O typie (sekwencyjny lub swobodny) pliku nie decyduje jego fizyczna forma.
- Plik można utworzyć sekwencyjnie, a następnie odczytywać i zapisywać swobodnie.
- Dla C plik jest tylko strumieniem bajtów, a sposób jego używania nie ma związku z żadnym formatem.

Sequential access



Random access



Pliki o dostępie swobodnym – podstawowe funkcje

- `fopen()`
- `fclose()`
- `fseek()`
- `fputc()`
- `fsacnf()`
- `feof()`

Podstawowe tryby otwarcia pliku funkcji `fopen()`

- „r+” – otwiera istniejący plik z możliwością odczytu i zapisu
- „w+” – otwiera nowy plik do zapisu i odczytu
- „a+” – otwiera plik z możliwością dodawania danych (wskaźnik plikowy wskazuje na koniec tego pliku), ale umożliwia też przejście wstecz oraz odczytanie i zapisanie danych „po drodze”.

Pliki o dostępie swobodnym – poruszanie się po pliku

Do poruszania się po zawartości pliku służy funkcja `fseek()`.

Funkcja `fseek()` przesuwa wskaźnik plikowy, dzięki czemu możliwe jest odczytywanie i zapisywanie danych w miejscach, które w przypadku dostępu sekwencyjnego byłyby nieosiągalne.

`fseek(wskaznikPlikowy, wartoscLong, początek);`

Wartości początku, których można używać w funkcji `fseek()`:

- `SEEK_SET` – początek pliku
- `SEEK_CUR` – bieżąca pozycja w pliku
- `SEEK_END` – koniec pliku

Pliki o dostępie swobodnym - przykład

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  FILE * fptr;
5
6  int main()
7  {
8      char letter;
9      int i;
10     fptr = fopen("letters.txt", "w+");
11     if (fptr == 0)
12     {
13         printf("Wystapil blad podczas otwieraniapliku.\n");
14         exit(1);
15     }
16
17     for (letter = 'A'; letter <= 'Z'; letter++)
18     {
19         fputc(letter, fptr);
20     }
21
22     puts("Zapisano litery od A do Z.");
23
24     fseek(fptr, -1, SEEK_END);
25
26     printf("Oto zawartosc pliku wydrukowana wstecz:\n");
27
28     for (i = 26; i > 0; i--)
29     {
30         letter = 'A' + i;
```

Podstawowe zasady podczas pracy z plikami

- Aby zmieniać dane znajdujące się w pliku, należy w wywołaniu funkcji `fopen()` użyć łańcucha określającego tryb otwarcia pliku ze znakiem plus
- Po zakończeniu pracy z plikiem należy go zamknąć przy użyciu funkcji `fclose()`
- Funkcja `fseek()` służy do ustawiania wskaźnika plikowego w różnych miejscach pliku
- Nie wolno używać pliku, jeśli funkcja `fopen()` zgłosi błąd (poprzez zwrócenie zera)

Pliki tekstowe

Pliki tekstowe są przede wszystkim czytelne dla człowieka – można je otworzyć dowolnym edytorem tekstowym i bez problemu na nich operować. Ta sama cecha sprawia, że dane w tej postaci są zasadniczo niezrozumiałe dla procesora, przez co przed wykonaniem faktycznych, programowych operacji na zapisanych za ich pomocą danych konieczne jest wykonanie szeregu analiz i konwersji.

Gynvael Coldwind „Zrozumieć programowanie”

Pliki binarne

W **plikach binarnych** dane przechowywane są w sposób zakodowany, bardzo często w sposób identyczny jak ma to miejsce w pamięci operacyjnej programu, dzięki czemu zapis i odczyt danych może być bardzo szybki – często wystarczy jedynie wczytać bajty do pamięci i można od razu na nich operować. Z drugiej strony, zawartość pliku jest dla człowieka nieczytelna, a więc wykonywanie ręcznych modyfikacji jest utrudnione i wymaga odpowiedniego podejścia i narzędzi.

Gynvael Coldwind „Zrozumieć programowanie”

Pliki testowe a pliki binarne

W przypadku niektórych systemów operacyjnych (np. z rodziny Windows) podczas otwierania pliku za pomocą pewnych standardowych funkcji (np. ze standardowej biblioteki języka C i C++) należy zadeklarować, czy plik ma zostać otwarty jako tekstowy (wywołanie `fopen(..., "r")`), czy binarny (`fopen(..., "rb")`). Tryb otwarcia ma wpływ na zachowanie kilku funkcji operujących na danym pliku.

Istnieją odrębne zestawy funkcji do odczytu i zapisu danych charakterystycznych dla obu typów tych plików. W przypadku plików binarnych funkcje te są zorientowane na pojedyncze bajty, podczas gdy w przypadku plików tekstowych operują one na conceptach „znaków” oraz „linii”.

Gynael Coldwind „Zrozumieć programowanie”

Pliki testowe a pliki binarne

W Linuxie za koniec linii uznawany jest znak LF.

W Windowsie za koniec linii uznawane jest CR LF.

„W praktyce wszystkie funkcje odczytujące dane z pliku otwartego w trybie tekstowym dokonują automatycznej konwersji sekwencji CR LF na pojedynczy znak LF. Co więcej, wszystkie funkcje zapisujące dane do pliku w tym trybie dokonują konwersji znaku LF na sekwencję CR LF, dzięki czemu programista, tworząc kod, nie musi zajmować się różnicami wynikającymi z używanej na danym systemie konwencji. Należy jednak zaznaczyć, że tego typu konwersja w przypadku plików binarnych jest absolutnie niepożądana i prowadzi do uszkodzeń danych podczas odczytu i zapisu. Konieczne jest więc zachowanie szczególnej ostrożności podczas otwierania plików, korzystając z API, które wspiera oba tryby pracy z ich zawartością.

Druga różnica pomiędzy trybem binarnym a tekstowym dotyczy znaku o kodzie 0x1A; w trybie tekstowym jest to tzw. miękki koniec pliku (*soft EOF*).”

Gynael Coldwind „Zrozumieć programowanie”

Operacje na plikach

Zadania :

Napisz program...

Operacje na plikach

Podsumowanie :

- pliki o dostępie sekwencyjne
- pliki o dostępie swobodnym
- pliki tekstowe
- pliki binarne

Dynamiczna alokacja pamięci

Dynamiczna alokacja pamięci

Plan zajęć :

- funkcje malloc, calloc, realloc i free
- ramka stosu
- segmenty pamięci w języku C
- zmienne statyczne – zagadnienia wybrane

Dynamiczna alokacja pamięci

Dynamiczna alokacja (dynamiczne przydzielanie) pamięci to mechanizm przydzielenia ciągłego obszaru pamięci do danego programu

- W trakcie deklaracji tablicy trzeba podać jej wielkość jako stałą dostępną na etapie kompilacji
- Często wielkość tablicy nie jest znana, aż do czasu uruchomienia programu ponieważ zależy od danych wejściowych
- Jednym z możliwych rozwiązań powyższego problemu jest deklarowanie tak dużej tablicy, by zawsze wystarczyła ona do obsługi danych
- Eleganckie podejście polega na przydzielaniu do programu większej ilości pamięci w miarę zwiększających się potrzeb i zwalnianie przez program obszaru pamięci który już nie jest wykorzystywany

Funkcja malloc

Funkcja malloc służy do dynamicznego przydzielania pamięci.

Argumentem funkcji malloc jest potrzebna ilość bajtów pamięci.

Jeżeli żądana ilość pamięci jest dostępna to funkcja malloc zwraca wskaźnik do początku przydzielonego bloku, w przeciwnym przypadku zwracany jest wskaźnik NULL.

```
void *malloc( size_t rozmiar );
```

Przykład alokacji pamięci dla 10 liczb całkowitych za pomocą funkcji malloc():

```
int *temps;
```

```
temps = (int *) malloc( 10 * sizeof( int ) );
```

Funkcja calloc

Funkcja calloc podobnie jak malloc służy do dynamicznego przydzielania pamięci.

Główną różnicą pomiędzy malloc i calloc jest to, że calloc po przydzieleniu inicjuje pamięć wartościami 0 i dopiero wtedy zwraca ją do wywołującego programu – taka operacja, choć bardzo wygodna, jest jednak bardziej czasochłonna.

Funkcja calloc wymaga podania ilości elementów i ilości bajtów na każdy element.

Jeżeli żądana ilość pamięci jest dostępna to funkcja calloc zwraca wskaźnik do początku przydzielonego bloku, w przeciwnym przypadku zwracany jest wskaźnik NULL.

```
void *calloc( size_t ilosc_elementow, size_t wielkosc_elementu );
```

Przykład alokacji pamięci dla 10 liczb całkowitych za pomocą funkcji calloc():

```
int *temps;
```

```
temps = (int *) calloc( 10, sizeof( int ) );
```

Funkcja realloc

Funkcja realloc wykorzystywana jest do zmiany wielkości poprzednio przydzielonego bloku pamięci.

Korzystając z tej funkcji można zwiększyć lub zmniejszyć wielkość bloku.

Jeżeli blok jest zwiększany, stara zawartość nie jest zmieniana, a na koniec bloku dodawany jest nowy obszar pamięci (nowy obszar pamięci nie jest w żaden sposób inicjalizowany).

Jeżeli blok jest zmniejszany, pamięć jest zabierana z końca. Zawartość pozostałej części pozostaje bez zmian.

Jeżeli oryginalny blok nie może być zwiększony, funkcja realloc przydziela nowy blok o właściwej wielkości i kopiuje zawartość starego bloku do nowego. Z tego powodu nie można korzystać ze wskaźników wskazujących na stary obszar pamięci. Po wykonaniu realloc należy korzystać tylko z nowej wartości wskaźnika, zwracanej przez funkcję.

```
void *realloc( void *wskaznik, size_t nowy_rozmiar );
```

Funkcja free

Gdy przydzielony fragment pamięci nie jest już potrzebny, wywoływana jest funkcja free, która zwraca obszar do puli wolnej pamięci.

Argumentem free musi być NULL lub wartość zwrócona poprzednio przez malloc, calloc lub realloc. Przekazanie do free wartości NULL nie daje żadnych efektów.

```
void free( void *wskaznik );
```

Dynamiczna alokacja pamięci – najczęstsze błędy

W programach korzystających z dynamicznego przydzielania pamięci może występować wiele różnych błędów. Są to:

- Dereferencja wskaźnika NULL
- Wyjście poza obszar przydzielonej pamięci
- Zwolnienie bloku, który nie został przydzielony w sposób dynamiczny
- Próba zwolnienia części bloku dynamicznego – za pomocą free
- Korzystanie z pamięci dynamicznej po jej zwolnieniu

Dynamiczna alokacja pamięci - przykład

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  main()
6  {
7      int i, aSize;
8      int * randomNums;
9
10     time_t t;
11
12     srand(time(&t));
13
14     printf("Ile liczb losowych chcesz zapisac w tablicy? ");
15     scanf(" %d", &aSize);
16
17     randomNums = (int *) malloc(aSize * sizeof(int));
18
19     if (!randomNums)
20     {
21         printf("Nie udalo się alokowac tablicy!\n");
22         exit(1);
23     }
24
25     for (i = 0; i < aSize; i++)
26     {
27         randomNums[i] = (rand() % 500) + 1;
28     }
29
30 }
```

Dynamiczna alokacja pamięci - przykład

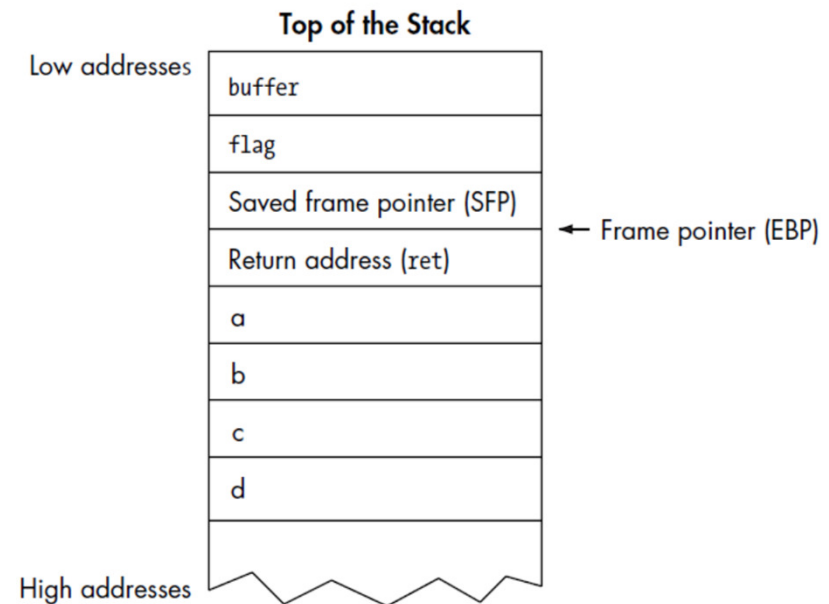
```
1  ...
2  int biggest = 0, smallest = 500;
3  double total = 0;
4  float average = 0;
5
6
7  for (i = 0; i < aSize; i++)
8  {
9      total += randomNums[i];
10     if (randomNums[i] > biggest)
11     {
12         biggest = randomNums[i];
13     }
14
15     if (randomNums[i] < smallest)
16     {
17         smallest = randomNums[i];
18     }
19 }
20
21 average = ((float)total)/((float)aSize);
22
23 printf("Najwieksza liczba: %d.\n", biggest);
24 printf("Najmniejsza liczba: %d.\n", smallest);
25 printf("Srednia: %.2f.\n", average);
26
27 free(randomNums);
28
29 return(0);
```

Ramka stosu

```

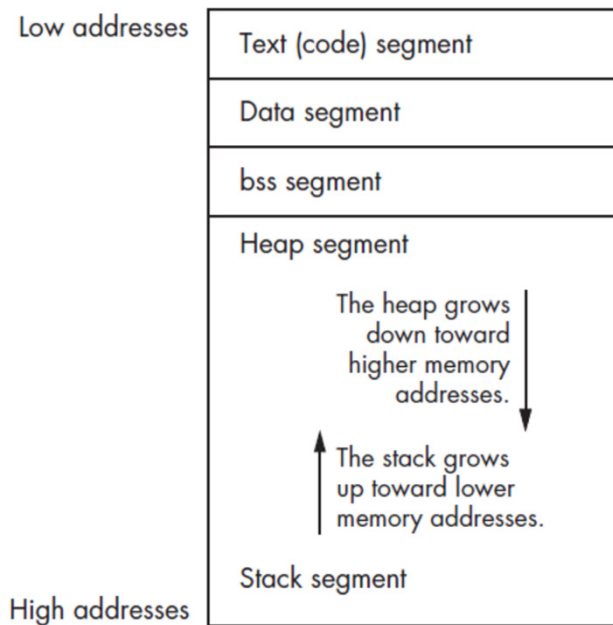
1 void test_function(int a, int b, int c, int d) {
2     int flag;
3     char buffer[10];
4
5     flag = 31337;
6     buffer[0] = 'A';
7 }
8
9 int main() {
10     test_function(1, 2, 3, 4);
11 }
12

```



JON ERICKSON
„HACKING: THE ART OF EXPLOITATION”

Segmenty pamięci



JON ERICKSON
„HACKING: THE ART OF EXPLOITATION”

Zmienne statyczne – zagadnienia wybrane

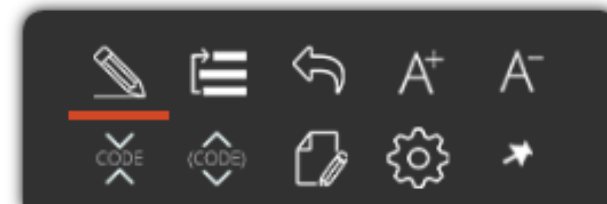
- Zasięg i „czas życia” zmiennych
- Zmienne o charakterze stałych
- Zmienne o charakterze ulotnym
- Zmienne o charakterze statycznym

Zasięg i „czas życia” zmiennych

- Zmienne ze względu na zasięg i „czas życia” dzieli się na zmienne lokalne i zmienne globalne
- Zmienne globalne dostępne są w całym programie, podczas gdy zmienne lokalne jedynie w bloku, w którym zostały zadeklarowane
- Zmienna globalna rezerwuje pamięć na cały czas działania programu, podczas gdy zmienna lokalna jedynie na czas jej dostępności w bloku, w którym została zadeklarowana

```
1 int main(){  
2   float r, o;  
3   ...  
4  
5 }  
6
```

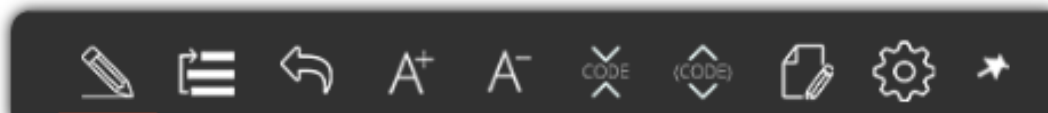
```
1 float r, o;  
2 int main(){  
3   ...  
4  
5 }  
6
```



Zmienne o charakterze stałych

- Wartość stałej ustalana jest jednorazowo na etapie deklaracji
- Nie można deklarować stałej w programie
- Stała przechowywana jest zwykle w tzw. pamięci programu w odróżnieniu od pozostałych zmiennych przechowywanych w tzw. pamięci danych

```
1  const float PI = 3.14;  
2
```



Zmienne o charakterze ulotnym

- Wyłącza procesy optymalizacji kodu wynikowego dla zmiennej
- Każde odwołanie do zmiennej (odczyt lub zapis) jest wykonywane bezpośrednio na komórce pamięci, w której zmienna jest przechowywana
- Umożliwia kontrolę kolejności odwoływania się do pamięci przy operacjach na zmiennych
- Umożliwia bezpieczną analizę zmiennej modyfikowanej poza blokiem programu, w którym jest analizowana

```
1 volatile int is_busy = 1;  
2  
3 while( is_busy ) {}  
4  
5
```



Zmienne o charakterze statycznym

- Nadaje zmiennej lokalnej właściwość zachowywania wartości, po opuszczeniu przez program bloku, w którym została zadeklarowana
- Czas życia statycznej zmiennej lokalnej jest taki sam jak zmiennej globalnej
- Inicjalizacja zmiennej statycznej występuje jednokrotnie w trakcie całego działania programu – analogicznie jak zmiennej globalnej

```
1 {  
2   static int i = 0;  
3   i++;  
4 }  
5  
6
```



Dynamiczna alokacja pamięci

Zadania :

Napisz program...

Dynamiczna alokacja pamięci

Podsumowanie :

- funkcje malloc, calloc, realloc i free
- ramka stosu
- segmenty pamięci w języku C
- zmienne statyczne – zagadnienia wybrane