Podstawy programowania

Grzegorz Jabłoński

Katedra Mikroelektroniki i Technik Informatycznych

tel. (631) 26-48

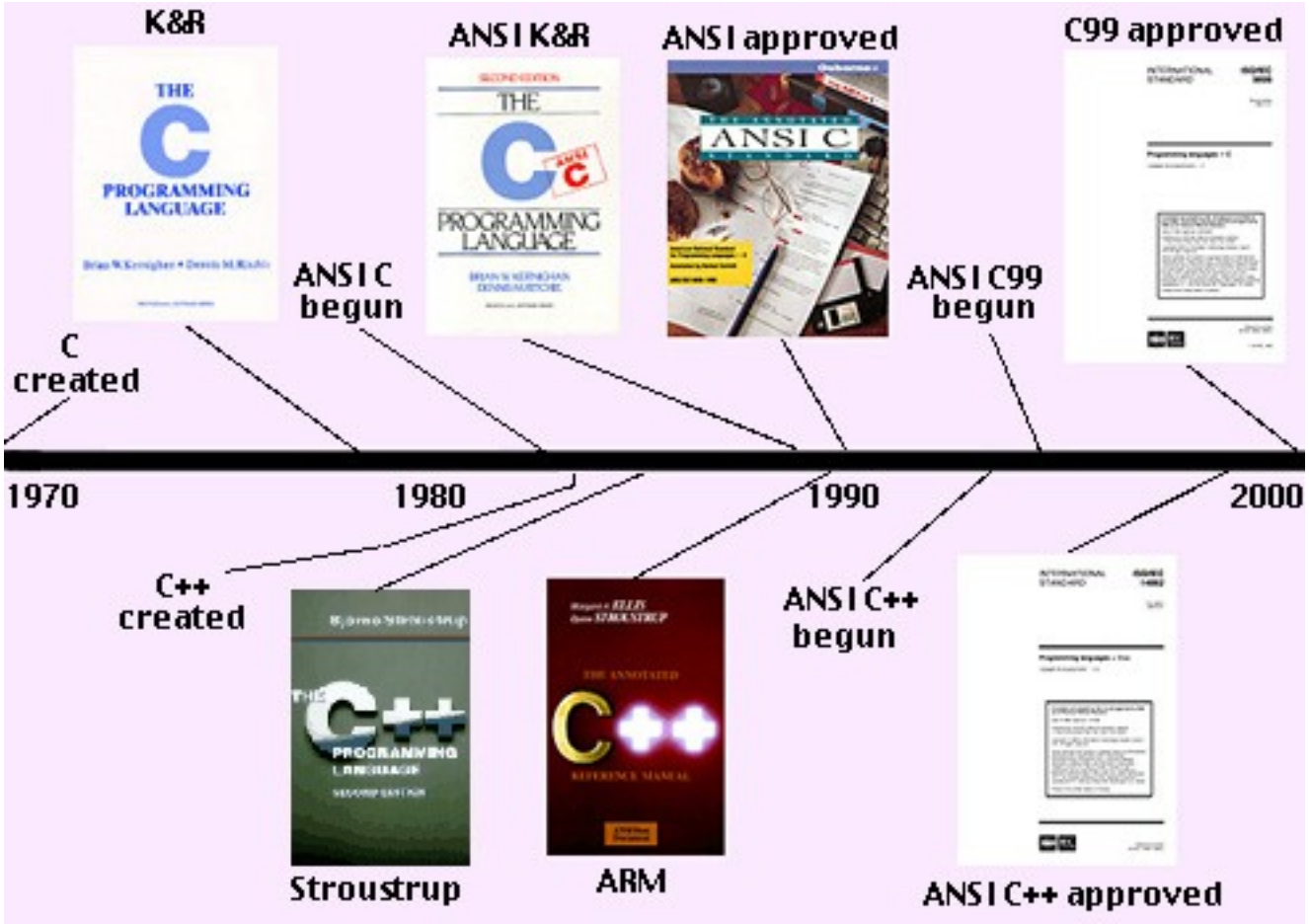
gwj@dmcs.p.lodz.pl

http://neo.dmcs.p.lodz.pl/pp

Historia języka C

- 1969 - Ken Thompson projektuje Unix, język B z BCPL
- 1970 - Thompson & Ritchie przekształcają B na C
- 1978 - K&R's "The C Programming Language"
- 1982 - ANSI tworzy komitet standaryzacyjny języka C
- 1988 - K&R uaktualniony zgodnie z wersją roboczą ANSI C
- 1989 - ANSI zatwierdza C (zwany "C89")
- 1990 - ISO zatwierdza C (zwany "C90")
- 1995 - Uformowany nowy komitet dla "C9X"
- 1999 - Nowy standard ISO (zwany "C99")
- 2000 - ANSI zatwierdza "C99"

Historia języka C

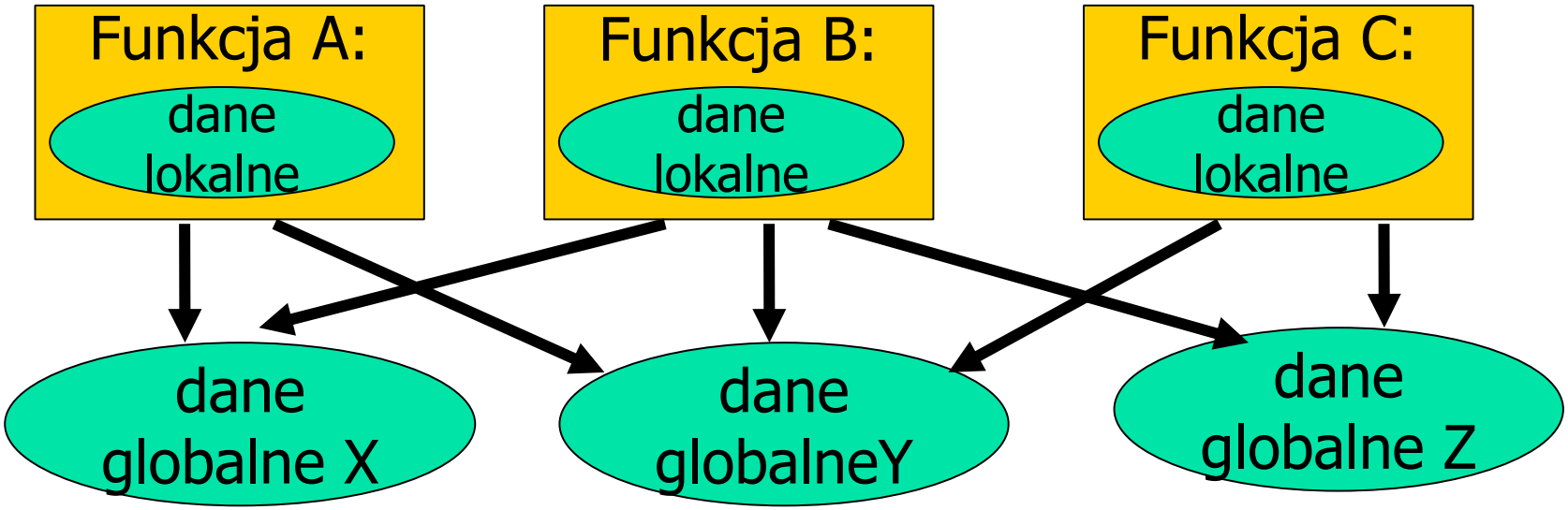


Programowanie strukturalne

- C, Pascal, Fortran są proceduralnymi językami programowania.
- Program w języku proceduralnym składa się z listy instrukcji, pętli i skoków warunkowych.
- Dla małych programów żadna inna zasada organizacyjna (paradygmat) nie jest potrzebna.
- Większe programy dzieli się na mniejsze jednostki.
- Program proceduralny jest podzielony na funkcje, które w idealnym przypadku mają ściśle określony cel i interfejs dla innych funkcji.
- Koncepcja podziału programu na funkcję można dalej rozszerzyć poprzez grupowanie pokrewnych funkcji w moduły.
- Podział programu na funkcje i moduły jest kluczową koncepcją programowania strukturalnego.

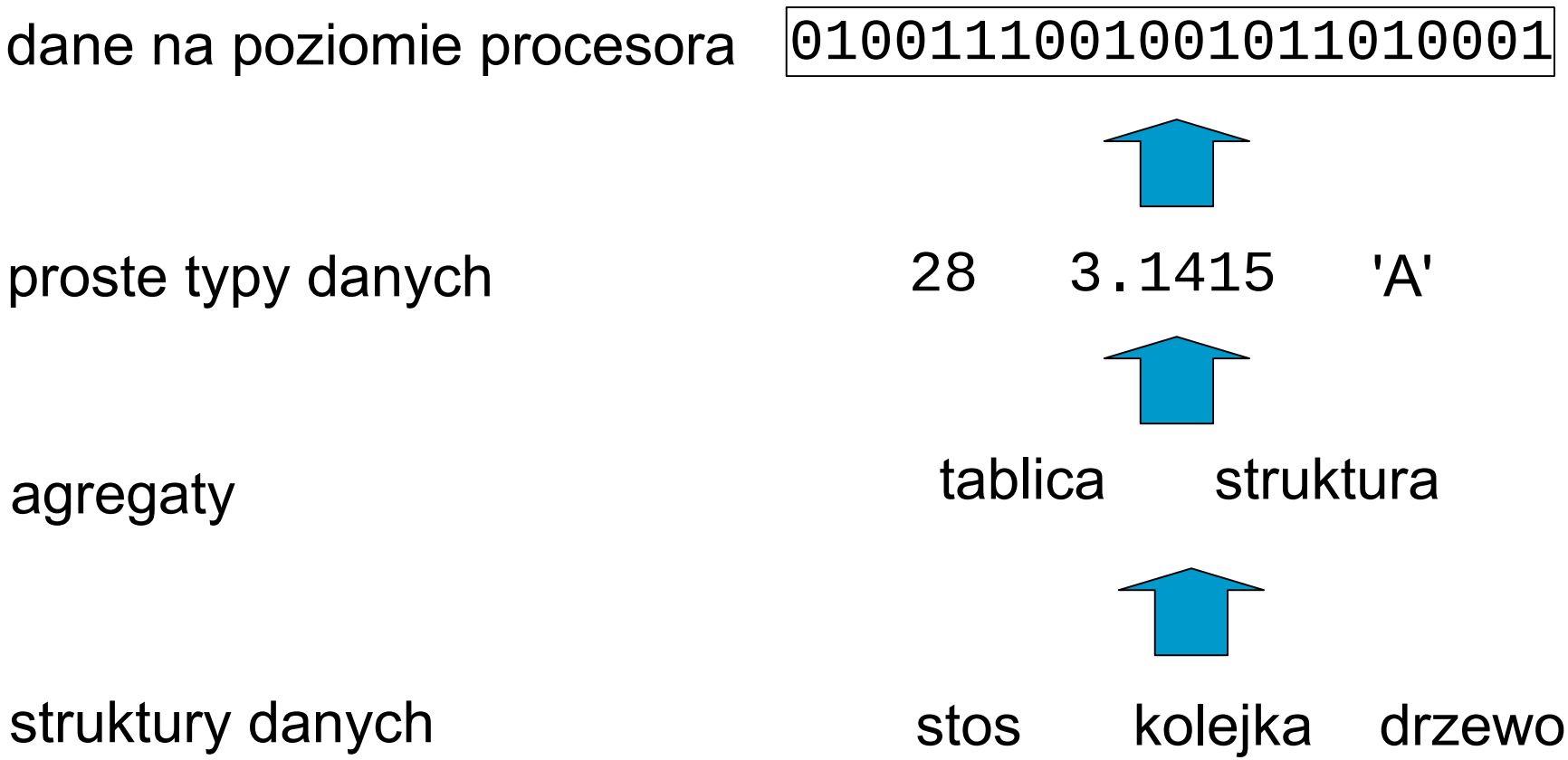
Problemy z programowaniem strukturalnym

- Funkcje mają nieograniczony dostęp do danych globalnych



- Duża liczba potencjalnych powiązań między funkcjami i danymi (wszystko jest powiązane z wszystkim, brak wyraźnych granic)
  - trudno wyrazić strukturę programu
  - trudno modyfikować i utrzymywać program
  - np. trudno stwierdzić, która funkcja operuje na których danych

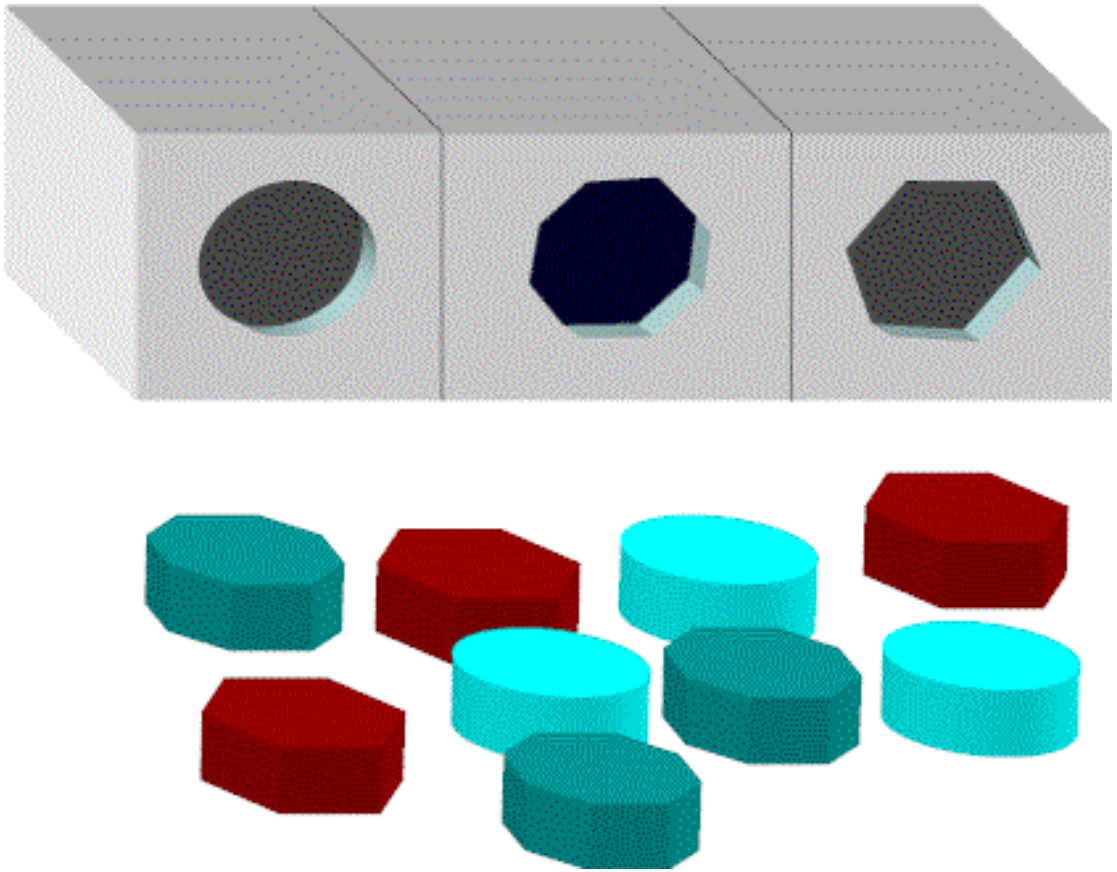
Od danych do struktur danych



Na każdym poziomie abstrakcji...

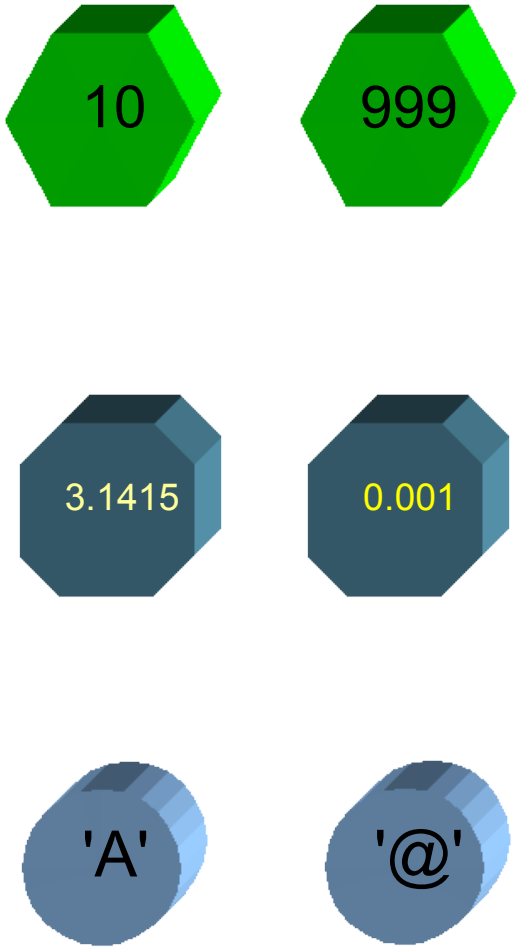
- Nie chcemy zajmować się budową obiektów niższego poziomu
- Chcemy zajmować się semantyką danych na bieżącym poziomie
- Co to jest ?
- Co z tym możemy zrobić ?

Proste typy danych



Proste typy danych

- Liczby całkowite
  - 1, 10, 999, 1000
- Liczby zmiennoprzecinkowe
  - 2.7128, 0.003, 7.0
- Znaki
  - 'A', 'B', '_', '@'



Reprezentacja liczb całkowitych – notacja pozycyjna

- Podstawa systemu liczbowego B $\Rightarrow$ B symboli na cyfrę:
  - Podstawa 10 (dziesiętny): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Podstawa 2 (dwójkowy): 0, 1
- Reprezentacja liczb:
  - $d_{31}d_{30} \ldots d_1d_0$ jest liczbą 32-bitową
  - wartość = $d_{31} \times B^{31} + d_{30} \times B^{30} + \ldots + d_1 \times B^1 + d_0 \times B^0$
- Binarnie: 0,1 (Używając cyfr dwójkowych, tzw. bitów)
  - $0b11010 = 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$
    $= 16 + 8 + 2$
    $= 26$
  - 5-cyfrowa liczba binarna zamienia się w dwucyfrową dziesiętną
  - Czy istnieje podstawa łatwo konwertowalna do podstawy 2?

liczby dwójkowe często zapisywane 0b… niestandardowe rozszerzenie

Liczby szesnastkowe – podstawa 16

- Cyfry szesnastkowe: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Normalne cyfry + 6 dodatkowych z alfabetu
  - W C zapisywane jako 0x… (np., 0xFAB5)
- Konwersja: liczba binarna⇔liczba szesnastkowa
  - 1 cyfra szesnastkowa reprezentuje 16 różnych wartości
  - 4 cyfry binarne reprezentują 16 różnych wartości
  - 1 cyfra szesnastkowa zastępuje 4 cyfry binarne
- Jedna cyfra szesnastkowa to “nibble”. Dwie to “bajt”.
- Przykład:
  - 1010 1100 0011 (binarnie) = 0x_____ ?

Liczby dziesiętne, szesnastkowe, binarne

- Przykłady:
  - 1010 1100 0011 (binarnie) = 0xAC3
  - 10111 (binarnie) = 0001 0111 (binarnie) = 0x17
  - 0x3F9 = 11 1111 1001 (binarnie)
- Jak konwertować między liczbami szesnastkowymi i dziesiętnymi?

ZAPAMIĘTAĆ !

00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Reprezentacja liczb ujemnych

- Najprostsze rozwiązanie: najbardziej znaczący bit określa znak liczby
  - 0 $\Rightarrow$ +, 1 $\Rightarrow$ -
  - Reszta bitów reprezentuje moduł liczby
- Reprezentacja “znak-moduł”
- dla 32 bitowych liczb $+1_{10}$ ma postać:

  0000 0000 0000 0000 0000 0000 0000 0001
- -1_{10} w reprezentacji znak-moduł ma postać:

  1000 0000 0000 0000 0000 0000 0000 0001

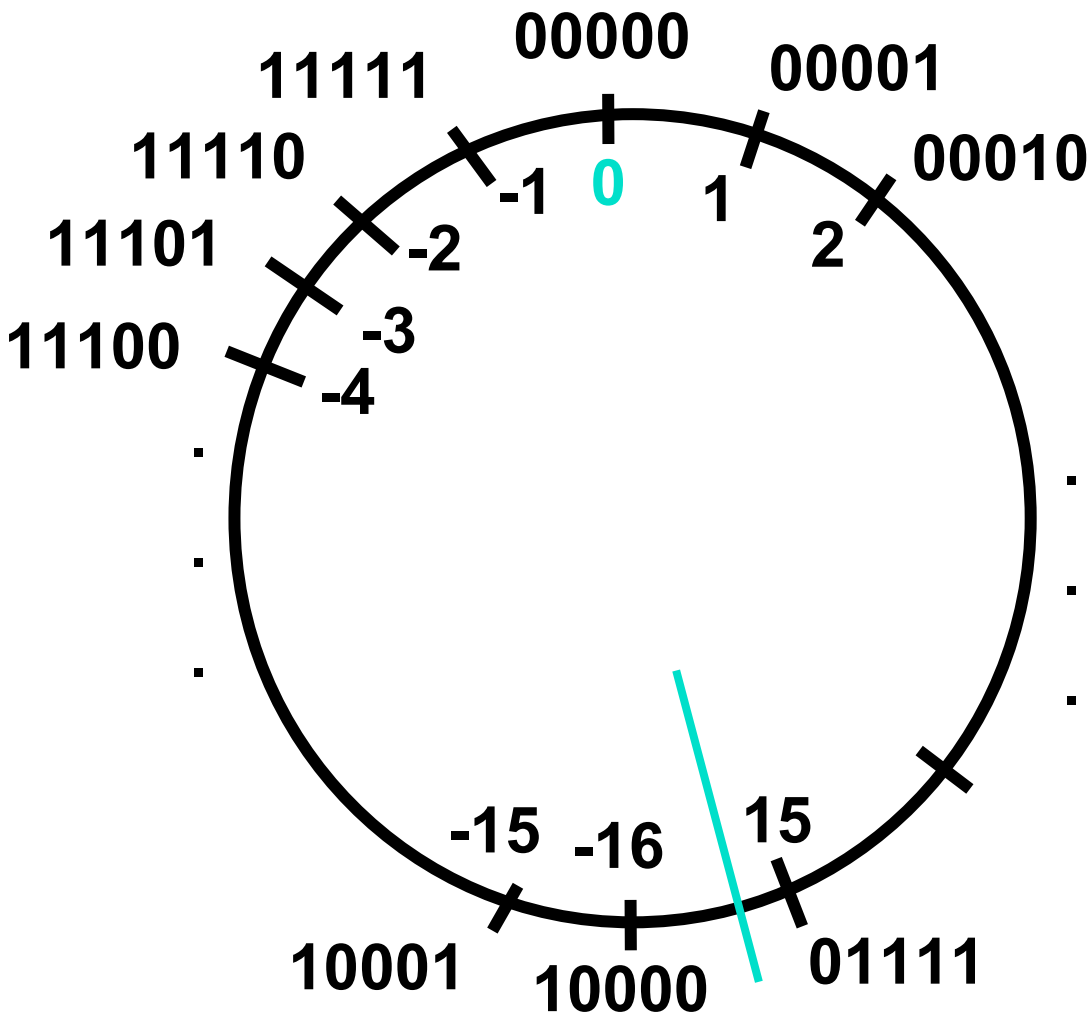
Wady reprezentacji znak-moduł

- Komplikacja układów arytmetycznych
  - Specjalne warunki określające, czy znaki są jednakowe, czy różne
- dwa zera
  - $0x00000000 = +0_{10}$
  - $0x80000000 = -0_{10}$
  - Co dwa zera oznaczają dla programisty?
- Dlatego reprezentacji znak-moduł nie stosuje się w praktyce

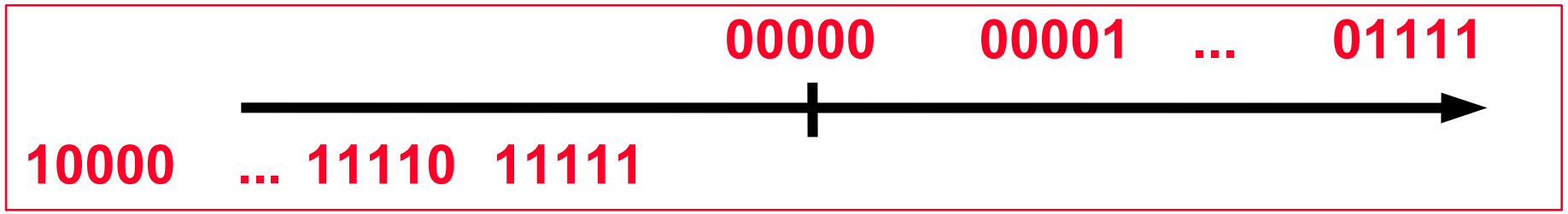
Standardowa reprezentacja liczb ujemnych

- Jaki będzie rezultat przy odejmowaniu liczby większej od mniejszej przy kodowaniu w naturalnym kodzie binarnym?
  - Nastąpi pożyczka od łańcucha wiodących zer, w wyniku czego rezultat będzie miał szereg wiodących jedynek
    - 3 - 4 $\Rightarrow$ 00…0011 - 00…0100 = 11…1111
  - Z braku lepszej alternatywy, wybierzmy reprezentację upraszczającą sprzęt
  - Jak w przypadku reprezentacji znak-moduł,
    - wiodące zera $\Rightarrow$ liczba dodatnia,
    - wiodące jedynki $\Rightarrow$ liczba ujemna
    - 000000...xxx ≥ 0, 111111...xxx < 0
    - 1…1111 to -1
- Ta reprezentacja zwana jest kodem uzupełnień do dwóch (U2)

“Oś” liczbowa w kodzie U2: N = 5



- 2^{N-1} liczb nieujemnych
- 2^{N-1} liczb ujemnych
- jedno zero
- ile liczb dodatnich?



Kod uzupełnień do dwóch dla N=32

$0000 \ldots 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
$0000 \ldots 0000\ 0000\ 0000\ 0001_2 = 1_{10}$
$0000 \ldots 0000\ 0000\ 0000\ 0010_2 = 2_{10}$

. . .

$0111 \ldots 1111\ 1111\ 1111\ 1101_2 = 2{,}147{,}483{,}645_{10}$
$0111 \ldots 1111\ 1111\ 1111\ 1110_2 = 2{,}147{,}483{,}646_{10}$
$0111 \ldots 1111\ 1111\ 1111\ 1111_2 = 2{,}147{,}483{,}647_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0000_2 = -2{,}147{,}483{,}648_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0001_2 = -2{,}147{,}483{,}647_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0010_2 = -2{,}147{,}483{,}646_{10}$

. . .

$1111 \ldots 1111\ 1111\ 1111\ 1101_{t2} = -3_{10}$
$1111 \ldots 1111\ 1111\ 1111\ 1110_2 = -2_{10}$
$1111 \ldots 1111\ 1111\ 1111\ 1111_2 = -1_{10}$

- Jedno zero, najstarszy bit zwany bitem znaku
- 1 “dodatkowa” liczba ujemna: brak dodatniego odpowiednika dla $2{,}147{,}483{,}648_{10}$

Wzór na kod uzupełnień do dwóch

- Może reprezentować dodatnie i ujemne liczby jako sumę wartości poszczególnych bitów pomnożonych przez potęgi dwójki:
  - $d_{31} \times -(2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$
  - Przykład: 1101_2

$= 1 \times -(2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= -2^3 + 2^2 + 0 + 2^0$

$= -8 + 4 + 0 + 1$

$= -8 + 5$

$= -3_{10}$

Kod uzupełnień do dwóch: liczba przeciwna

- Zamienić każde 0 na 1 oraz 1 na 0 (zanegować, uzupełnić do jedynki), a następnie dodać 1 do wyniku
- Dowód: Suma liczby i jej uzupełnienia jedynkowego wynosi $111...111_2$
  - Jednocześnie $111...111_2 = -1_{10}$
  - Niech x' $\Rightarrow$ dopełnienie jedynkowe reprezentacji liczby x
  - Wówczas x + x' = -1 $\Rightarrow$ x + x' + 1 = 0 $\Rightarrow$ x' + 1 = -x
- Przykład: -3 $\Rightarrow$ +3 $\Rightarrow$ -3

  x : 1111 1111 1111 1111 1111 1111 1111 1101_2

  x': 0000 0000 0000 0000 0000 0000 0000 0010_2

  +1: 0000 0000 0000 0000 0000 0000 0000 0011_2

  ()': 1111 1111 1111 1111 1111 1111 1111 1100_2

  +1: 1111 1111 1111 1111 1111 1111 1111 1101_2

Kod uzupełnień do dwóch: rozszerzenie bitem znaku

- Konwersja liczby w kodzie U2 zapisanej z użyciem n bitów na więcej niż n bitów
- Należy powielić najbardziej znaczący bit (bit znaku) wypełniając nim nowe bity
  - dodatnia liczba w kodzie U2 ma nieskończoną liczbę wiodących zer
  - ujemna liczba w kodzie U2 ma nieskończoną liczbę wiodących jedynek
  - Reprezentacja binarna ukrywa wiodące bity, rozszerzenie bitem znaku pokazuje niektóre z nich
  - Rozszerzenie 16-bitowej reprezentacji -4_{10} na 32-bity:

$$1111\ 1111\ 1111\ 1100_2$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$$

Kod U2: mnożenie i dzielenie przez 2

- Mnożenie przez 2 to przesunięcie w lewo (o ile nie wystąpi przepełnienie)

  $(-5_{10}) * 2_{10} = -10_{10}$

  $1111\ 1111\ 1111\ 1011_2 * 2_{10} = 1111\ 1111\ 1111\ 0110_2$

  $5_{10} * 2_{10} = 10_{10}$

  $0000\ 0000\ 0000\ 0101_2 * 2_{10} = 0000\ 0000\ 0000\ 1010_2$

- Dzielenie przez 2 wymaga wsunięcia z lewej strony kopii najbardziej znaczącego bitu

  $(-4_{10}) / 2_{10} = -2_{10}$

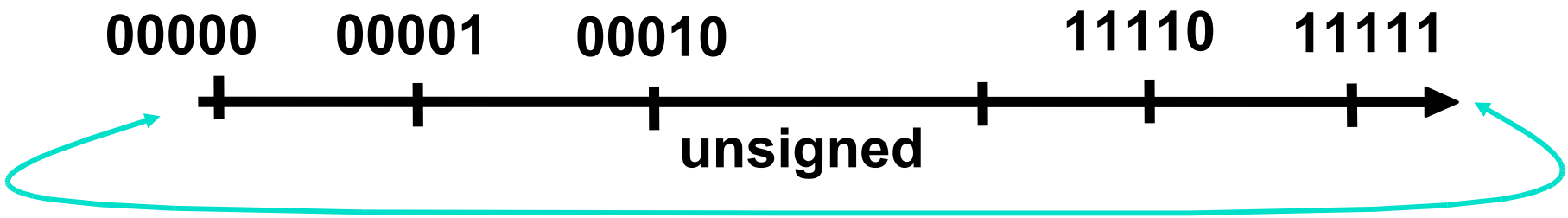
  $1111\ 1111\ 1111\ 1100_2 / 2_{10} = 1111\ 1111\ 1111\ 1110_2$

  $(4_{10}) / 2_{10} = 2_{10}$

  $0000\ 0000\ 0000\ 0100_2 / 2_{10} = 0000\ 0000\ 0000\ 0010_2$

Przepełnienie

- Układy bitów zaprezentowane uprzednio nie są liczbami całkowitymi, mogą reprezentować ograniczony zakres liczb.
- Liczby mają nieskończoną liczbę cyfr
  - prawie wszystkich jednakowych (00…0 or 11…1) z wyjątkiem pewnej liczby cyfr z prawej strony
  - Zazwyczaj nie pokazujemy cyfr wiodących
- Jezeli wynik dodawania (lub -, *, /) nie może być reprezentowany przez liczbę bitów zaimplementowaną sprzętowo, mówimy że wystąpiło przepełnienie.



Typy całkowite w języku C

- **`signed`** i **`unsigned`**
  - traktowane jako liczby odpowiednio ze znakiem i bez znaku
  - liczby ze znakiem zwykle implementowane w kodzie U2
  - ta sama liczba bitów, różny zakres
- dokładny rozmiar i zakres typów całkowitych nie jest zdefiniowany w standardzie, zależy od implementacji.
  - liczbę bajtów zajmowaną przez zmienną danego typu można odczytać za pomocą operatora **`sizeof()`**
  - zakres wartości które zmienna danego typu może przyjmować można odczytać używając makrodefinicji z pliku nagłówkowego **`limits.h`**

Typ `char`

- standard nie określa, czy to typ ze znakiem, czy bez znaku
- musi przechować każdy znak z zestawu znaków
- można go dodatkowo uściślić przy pomocy słowa kluczowego signed lub unsigned
- z definicji, **`sizeof(char) == 1`**
- ma co najmniej 8 bitów

```
char c1; /* signed or unsigned */
unsigned char c2;
signed char c3;

printf("%d\n", sizeof(c1)); /* prints 1 */
printf("%d\n", sizeof(char)); /* also prints 1 */
```

Typ `char` – makrodefinicje w `<limits.h>`

- **`CHAR_BIT`**
  - Makro podaje liczbę bitów używaną do reprezentacji obiektu typu char
- **`CHAR_MAX`**
  - Makro podaje maksymalną wartość dla typu char. Jest ona równa:
    - **`SCHAR_MAX`** jeżeli char może przechowywać liczby ujemne
    - **`UCHAR_MAX`** w przeciwnym przypadku
- **`CHAR_MIN`**
  - Makro podaje minimalną wartość dla typu char. Jest ona równa:
    - **`SCHAR_MIN`** jeżeli char może przechowywać liczby ujemne
    - zero w przeciwnym przypadku
- **`SCHAR_MAX`**
  - Makro podaje maksymalną wartość dla typu signed char
- **`SCHAR_MIN`**
  - Makro podaje minimalną wartość dla typu signed char
- **`UCHAR_MAX`**
  - Makro podaje maksymalną wartość dla typu unsigned char

Zestawy znaków

- ASCII
  - Sposób na reprezentację znaków alfabetu łacińskiego jako liczb, w którym każdej literze przypisuje się liczbę z zakresu; nie wszystkie znaki są drukowalne. Skrót określenia American Standard Code for Information Interchange.
  - Znaki sterujące ASCII przedstawia tabela po prawej
- EBCDIC
  - Extended Binary Coded Decimal Interchange Code
  - 8-bitowe rozszerzenie firmy IBM 4-bitowego kodowania cyfr 0-9 zwanego Binary Coded Decimal (0000-1001).

Char	Dec	Control-Key	Control Action
NUL	0	^@	NULl character
SOH	1	^A	Start Of Heading
STX	2	^B	Start of TeXt
ETX	3	^C	End of TeXt
EOT	4	^D	End Of Transmission
ENQ	5	^E	ENQuiry
ACK	6	^F	ACKnowledge
BEL	7	^G	BELl, rings terminal bell
BS	8	^H	BackSpace (non-destructive)
HT	9	^I	Horizontal Tab (move to next tab positior
LF	10	^J	Line Feed
VT	11	^K	Vertical Tab
FF	12	^L	Form Feed
CR	13	^M	Carriage Return
SO	14	^N	Shift Out
SI	15	^O	Shift In
DLE	16	^P	Data Link Escape
DC1	17	^Q	Device Control 1, normally XON
DC2	18	^R	Device Control 2
DC3	19	^S	Device Control 3, normally XOFF
DC4	20	^T	Device Control 4
NAK	21	^U	Negative AcKnowledge
SYN	22	^V	SYNchronous idle
ETB	23	^W	End Transmission Block
CAN	24	^X	CANcel line
EM	25	^Y	End of Medium
SUB	26	^Z	SUBstitute
ESC	27	^[	ESCape
FS	28	^\	File Separator
GS	29	^]	Group Separator
RS	30	^^	Record Separator
US	31	^_	Unit Separator

Znaki drukowalne ASCII

Dec	Description
32	Space
33	Exclamation mark
34	Quotation mark
35	Cross hatch (number sign)
36	Dollar sign
37	Percent sign
38	Ampersand
39	Closing single quote (apostrophe)
40	Opening parentheses
41	Closing parentheses
42	Asterisk (star, multiply)
43	Plus
44	Comma
45	Hyphen, dash, minus
46	Period
47	Slash (forward or divide)
48	Zero
49	One
50	Two
51	Three
52	Four
53	Five
54	Six
55	Seven
56	Eight
57	Nine
58	Colon
59	Semicolon
60	Less than sign
61	Equals sign
62	Greater than sign
63	Question mark

Char	Dec	Description
@	64	At-sign
A	65	Upper case A
B	66	Upper case B
C	67	Upper case C
D	68	Upper case D
E	69	Upper case E
F	70	Upper case F
G	71	Upper case G
H	72	Upper case H
I	73	Upper case I
J	74	Upper case J
K	75	Upper case K
L	76	Upper case L
M	77	Upper case M
N	78	Upper case N
O	79	Upper case O
P	80	Upper case P
Q	81	Upper case Q
R	82	Upper case R
S	83	Upper case S
T	84	Upper case T
U	85	Upper case U
V	86	Upper case V
W	87	Upper case W
X	88	Upper case X
Y	89	Upper case Y
Z	90	Upper case Z
[	91	Opening square bracket
\	92	Backslash (Reverse slant)
]	93	Closing square bracket
^	94	Caret (Circumflex)
_	95	Underscore

Char	Dec	Description
`	96	Opening single quote
a	97	Lower case a
b	98	Lower case b
c	99	Lower case c
d	100	Lower case d
e	101	Lower case e
f	102	Lower case f
g	103	Lower case g
h	104	Lower case h
i	105	Lower case i
j	106	Lower case j
k	107	Lower case k
l	108	Lower case l
m	109	Lower case m
n	110	Lower case n
o	111	Lower case o
p	112	Lower case p
q	113	Lower case q
r	114	Lower case r
s	115	Lower case s
t	116	Lower case t
u	117	Lower case u
v	118	Lower case v
w	119	Lower case w
x	120	Lower case x
y	121	Lower case y
z	122	Lower case z
{	123	Opening curly brace
|	124	Vertical line
}	125	Closing curly brace
~	126	Tilde (approximate)
DEL	127	Delete (rubout), cross-hatch box

Zestaw znaków EBCDIC

Dec	EBCDIC	
0	NUL	Null
1	SOH	Start of Heading
2	STX	Start of Text
3	ETX	End of Text
4	PF	Punch Off
5	HT	Horizontal Tab
6	LC	Lower Case
7	DEL	Delete
10	SMM	Start of Manual Message
11	VT	Vertical Tab
12	FF	Form Feed
13	CR	Carriage Return
14	SO	Shift Out
15	SI	Shift In
16	DLE	Data Link Escape
17	DC1	Device Control 1
18	DC2	Device Control 2
19	TM	Tape Mark
20	RES	Restore
21	NL	New Line
22	BS	Backspace
23	IL	Idle
24	CAN	Cancel
25	EM	End of Medium
26	CC	Cursor Control
27	CU1	Customer Use 1
28	IFS	Interchange File Separator
29	IGS	Interchange Group Separator
30	IRS	Interchange Record Separator
31	IUS	Interchange Unit Separator
32	DS	Digit Select
33	SOS	Start of Significance
34	FS	Field Separator
36	BYP	Bypass
37	LF	Line Feed

Dec	EBCDIC	
38	ETB	End of Transmission Block
39	ESC	Escape
42	SM	Set Mode
43	CU2	Customer Use 2
45	ENQ	Enquiry
46	ACK	Acknowledge
47	BEL	Bell
50	SYN	Synchronous Idle
52	PN	Punch On
53	RS	Reader Stop
54	UC	Upper Case
55	EOT	End of Transmission
59	CU3	Customer Use 3
60	DC4	Device Control 4
61	NAK	Negative Acknowledge
63	SUB	Substitute
64	SP	Space
74	¢	Cent Sign
75	.	Period, Decimal Point, "dot"
76	<	Less-than Sign
77	(	Left Parenthesis
78	+	Plus Sign
79	|	Logical OR
80	&	Ampersand
90	!	Exclamation Point
91	$	Dollar Sign
92	*	Asterisk, "star"
93	)	Right Parenthesis
94	;	Semicolon
95	¬	Logical NOT
96	-	Hyphen, Minus Sign
97	/	Slash, Virgule
107	,	Comma
108	%	Percent
109	_	Underline, Underscore

Dec	EBCDIC	
110	>	Greater-than Sign
111	?	Question Mark
122	:	Colon
123	#	Number Sign, Octothorp, "pound"
124	@	At Sign
125	'	Apostrophe, Prime
126	=	Equal Sign
127	"	Quotation Mark
129	a	a
130	b	b
131	c	c
132	d	d
133	e	e
134	f	f
135	g	g
136	h	h
137	i	i
145	j	j
146	k	k
147	l	l
148	m	m
149	n	n
150	o	o
151	p	p
152	q	q
153	r	r
162	s	s
163	t	t
164	u	u
165	v	v
166	w	w
167	x	x
168	y	y
169	z	z
185	`	Grave Accent

Dec	EBCI
193	A
194	B
195	C
196	D
197	E
198	F
199	G
200	H
201	I
209	J
210	K
211	L
212	M
213	N
214	O
215	P
216	Q
217	R
226	S
227	T
228	U
229	V
230	W
231	X
232	Y
233	Z
240	0
241	1
242	2
243	3
244	4
245	5
246	6
247	7
248	8
249	9

Typ `int`

- typ ze znakiem
- podstawowy typ całkowity, reprezentuje typ całkowity naturalny dla danego komputera
- co najmniej 16-bitowy
- można go uzupełnić słowem kluczowym signed lub unsigned

```
int i1; /* signed */
unsigned int i2;
signed int i3;

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

Typ `long int`

- typ ze znakiem
- co najmniej 32 bity, nie mniej niż **`int`**
- może być uzupełniony słowem kluczowym signed lub unsigned
- słowo kluczowe int może być pominięte w deklaracjach

```
long int i1; /* signed */
unsigned long int i2;
signed long int i3;
long i4; /* same type as i1 */
unsigned long i5; /* same type as i2 */
signed long i6; /* same type as i3 */

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

Typ `short int`

- typ ze znakiem
- co najmniej 16 bitów, nie więcej niż int
- może być uzupełniony słowem kluczowym signed lub unsigned
- słowo kluczowe int może być pominięte w deklaracjach

```
short int i1; /* signed */
unsigned short int i2;
signed short int i3;
short i4; /* same type as i1 */
unsigned short i5; /* same type as i2 */
signed short i6; /* same type as i3 */

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

Typ `long long int`

- dodany w standardzie C99
- typ ze znakiem
- co najmniej 64 bity, nie mniej niż long
- może być uzupełniony słowem kluczowym signed lub unsigned
- słowo kluczowe int może być pominięte w deklaracjach

```
long long int i1; /* signed */
unsigned long long int i2;
signed long long int i3;
long long i4; /* same type as i1 */
unsigned long long i5; /* same type as i2 */
signed long long i6; /* same type as i3 */

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

Typy całkowite – Makrodefinicje w <limits.h>

- **`INT_MAX`**
  - Makro zwraca maksymalną wartość dla typu **`int`**
- **`INT_MIN`**
  - Makro zwraca minimalną wartość dla typu **`int`**
- **`UINT_MAX`**
  - Makro zwraca maksymalną wartość dla typu **`unsigned int`**
- **`LONG_MAX, LONG_MIN, ULONG_MAX`**
  - Analogicznie dla typu **`long`**
- **`SHRT_MAX, SHRT_MIN, USHRT_MAX`**
  - Analogicznie dla typu **`short`**
- **`LLONG_MAX, LLONG_MIN, ULLONG_MAX`**
  - Analogicznie dla typu **`long long`**

Stałe całkowite (literały)

- Notacja dziesiętna:
  - `int`: `1234`
  - `long int`: `1234L, 1234l`
  - `unsigned int`: `1234U, 1234u`
  - `unsigned long int`: `1234UL, 1234ul, 1234Ul, 1234uL`
  - `long long int`: `1234LL, 1234ll`
  - `unsigned long long`: `1234ULL, 1234ull, 1234uLL, 1234Ull`
- Notacja ósemkowa (oktalna):
  - liczba zaczyna się od `0` (zero)
  - `031 == 25`
    - (31 Oct == 25 Dec, łatwo pomylić dwa święta)
  - dozwolone te same sufiksy co w poprzednim przypadku
- Notacja szesnastkowa (heksadecymalna):
  - liczba zaczyna się od `0x` (zero x)
  - `0x31 == 49`
  - dozwolone te same sufiksy co w poprzednich przypadkach

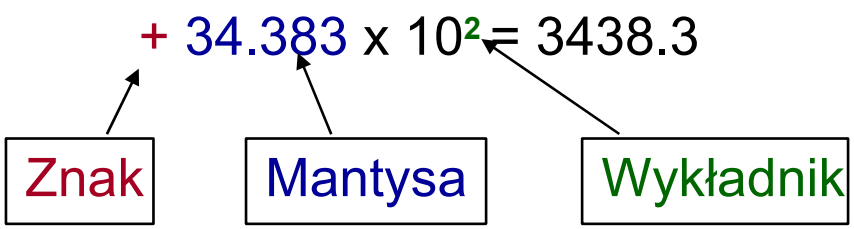
Stałe znakowe (literały)

- Notacja bezpośrednia:
  - `'a', 'b', ..., 'z', '0', ..., '9'`
- Znaki specjalne:
  - `'\n'` - znak nowego wiersza
  - `'\r'` - powrót karetki
  - `'\a'` - alert wizualny
  - `'\b'` - cofacz
  - `'\f'` - wysuw strony
  - `'\t'` - tabulator poziomy
  - `'\v'` - tabulator pionowy
  - `'\''` - apostrof
  - `'\"'` - cudzysłów
  - `'\?'` - pytajnik
  - `'\\'` - odwrotny ukośnik
- Notacja ósemkowa:
  - `'\077'`
  - `'\0'`
    (zwany **`NUL`** – przez jedno 'l')
  - Notacja szesnastkowa:
  - `'\x32'`

Liczby zmiennoprzecinkowe

- Liczby zmiennoprzecinkowe są używane do reprezentacji liczb rzeczywistych
  - 1.23233, 0.0003002, 3323443898.3325358903
- Liczby zmiennoprzecinkowe to podzbiór zbioru liczb rzeczywistych
  - Zakres przechowywanych liczb jest ograniczony
    - Zależy od zastosowanej liczby bitów
  - Ograniczona precyzja
    - 1234567890123456789 0 --> 1234567890000000000
    - Liczby zmiennoprzecinkowe to przybliżenie liczb rzeczywistych, podczas gdy liczby całkowite są reprezentowane dokładnie

Notacja wykładnicza



+ 3.4383 x 10^3 = 3438.3

Forma znormalizowana: jedna cyfra przed przecinkiem dziesiętnym

+3.4383000E+03 = 3438.3

Notacja zmiennoprzecinkowa

8-cyfrowa mantysa może reprezentować jedynie 8 cyfr znaczących

Binarne liczby zmiennoprzecinkowe

+ 101.1101

$= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$

$= 4 + 0 + 1 + 1/2 + 1/4 + 0 + 1/16$

$= 5.8125$

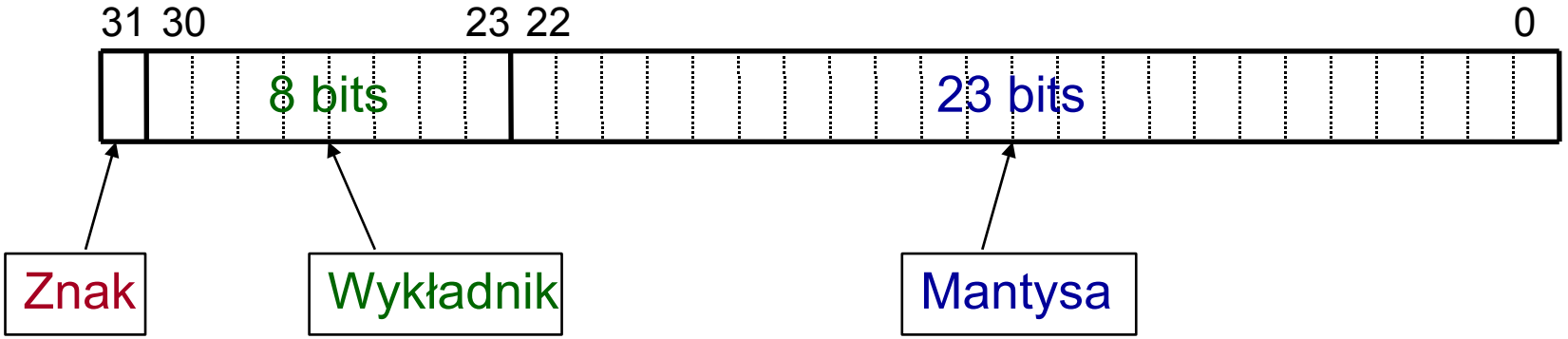
+1.011101 E+2

Notacja znormalizowana, czyli taka, w której przecinek dwójkowy występuje zaraz za pierwszą cyfrą

Uwaga: pierwsza cyfra jest zawsze niezerowa
--> pierwsza cyfra jest zawsze jedynką

Format zmiennoprzecinkowy IEEE

- The Institute of Electrical and Electronics Engineers
- Wymawiany "I-triple-E"
- Najbardziej znany z projektowania standardów dla przemysłu komputerowego i elektronicznego



0: Dodatni
1: Ujemny

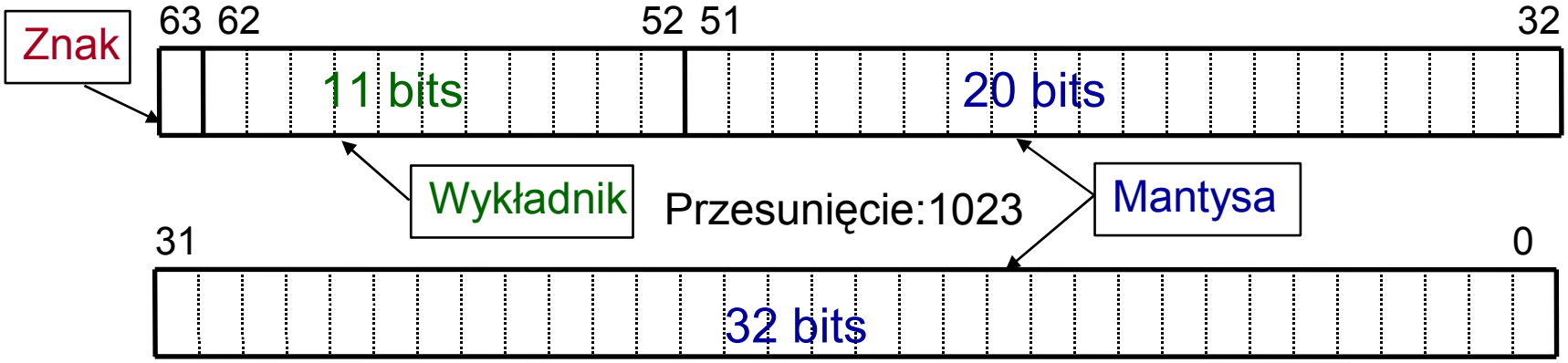
Przesunięty o 127.

Wiodące ‘1’ występuje niejawnie, nie jest reprezentowane w pamięci

$$\text{Liczba} = -1^{S} * (1 + M) \times 2^{E-127}$$

- Pozwala na reprezentację liczb w zakresie 2^{-127} do 2^{+128} ($10^{\pm 38}$)
- Ponieważ mantysa zawsze zaczyna się od ‘1’, nie trzeba tego jawnie zapisywać
  - Mantysa ma efektywnie 24 bity

Format IEEE o podwójnej precyzji



$$\text{Liczba} = -1^{S} * (1 + M) \times 2^{E-1023}$$

- Pozwala na reprezentację liczb w zakresie od 2^{-1023} do $2^{+1024}(10^{\pm 308})$
- Większa mantysa pozwala na większą precyzję

Format IEEE o rozszerzonej precyzji

- Opcjonalna rekomendacja IEEE dla dokładności większej niż float/double
  - Zaimplementowana sprzętowo (Intel 80-bit)
  - Pojedyncza precyzja
    - Co najmniej p = 32
    - Co najmniej 11 bitów na wykładnik
  - Podwójna precyzja
    - p >= 64
    - Wykładnik >= 15 bitów

Typy zmiennoprzecinkowe w języku C

- Trzy typy zmiennoprzecinkowe w C
  - **`float`**
  - **`double`**
  - **`long double`**
- Najczęściej przechowywane z użyciem standardu IEEE
  - niekoniecznie, mogą nawet używać podstawy innej niż 2
  - charakterystyka typów opisana w pliku **`<float.h>`**
- Dodatkowo trzy typy zespolone w C99
- Stałe:
  - **`1234.3`** – stała typu **`double`**
  - **`12345.5e7`** – stała typu **`double`**
  - **`123.4f`** – stała typu **`float`**
  - **`123.4F`** – stała typu **`float`**
  - **`123.4l`** – stała typu **`long double`**
  - **`123.4L`** – stała typu **`long double`**

Problemy z liczbami zmiennoprzecinkowymi

- Wiele liczb nie może być przedstawionych dokładnie
  - Reprezentacja 1/3 to 0.3333
    - 3 * “1/3” ≠ 1
  - Podobny problem 1/10 w przypadku ułamków binarnych
- Wyniki operacji zmiennoprzecinkowych prawie nigdy nie są równe dokładnie odpowiadające wartości matematycznej
- Wyniki obliczeń mogą się nieco różnić dla różnych systemów komputerowych i wszystkie być poprawne. Jeżeli komputery przestrzegają tego samego standardu, różnice znacznie się zmniejszają.
- Wyniki mogą zależeć od włączonych opcji optymalizacji
  - Wartości mogą być przechowywane z większą precyzją w rejestrach procesora, niż w pamięci

Problemy z liczbami zmiennoprzecinkowymi

```
int main()
{
  float a = 2.501f;
  a *= 1.5134f;
  if (a == 3.7850134)
          printf("Expected value\n");
  else
          printf("Unexpected value\n");
  return 0;
}
```

- Nigdy nie należy porównywać liczb zmiennoprzecinkowych
  - nie używać `if (a == b) ...`
  - zamiast tego używać `if( fabs(a - b) < error) ...`

Identyfikatory

- Nazwy obiektów w programie (zmienne, funkcje, itd.)
  - `int nMyPresentIncome = 0;`
  - `int DownloadOrBuyCD();`
- Do 31 znaków (litery, cyfry, znak _)
- Musi zaczynać się od litery
- Wielkość liter jest ważna! (“Url” to coś innego niż “URL”)

Style nazewnictwa

- Style:
  - lower_case
  - CAPITAL_CASE
  - camelCase
  - PascalCase (zwany również TitleCase)
  - szHungarianNotation
- Notacja węgierska:
  - Wymyślona przez Charlesa Simonyi'ego, Węgra, urodzonego w Budapeszcie w 1948

Niejawna konwersja typów

- Niejawne konwersje
  - `char b = '9'; /* Converts '9' to 57 */`
  - `int a = 1;`
  - `int s = a + b;`
- Promocja całkowita przed operacją: **`char/short`** $\Rightarrow$ **`int`**
- Przy wywoływaniu niezadeklarowanych funkcji, również promocja zmiennoprzecinkowa: **`float`** $\Rightarrow$ **`double`**
- Jeżeli jeden operand ma typ **`double`**, drugi jest konwertowany do **`double`**
- w przeciwnym wypadku jeżeli jeden operand ma typ **`float`**, drugi jest konwertowany do **`float`**

```
int a = 3;
float x = 97.6F;
double y = 145.987;
y = x * y;
x = x + a;
```

Jawna konwersja typów

- Zwana również rzutowaniem
- Czasami nie odpowiadają nam konwersje niejawne

```
float x = 97.6;
x = (int)x + 1;
```

- Czasami musimy pomóc kompilatorowi

```
float x = 97.6f;
printf("%d\n", x);          ⇒ 1610612736
printf("%d\n", (int) x);    ⇒ 97
```

- Prawie każda konwersja jest wykonalna - ale niekoniecznie oznacza to, co mamy na myśli!!

Niewłaściwa konwersja typów

- Przykład:

```
int x = 35000;
short s = x;
printf("%d %d\n", x, s);
```

- Otrzymujemy:

```
35000 -30536
```

Stałe

- Do deklaracji każdej zmiennej możemy dodać słowo kluczowe **`const`**
  - **`const int base = 345;`**
- Ta zmienna zostaje stałą
- Stała musi mieć przypisaną wartość w momencie deklaracji
- Próba modyfikacji stałej spowoduje błąd kompilacji

```
int main()
{
  const int a = 20;
  a = 31; /* error */
  return 0;
}
```

Wartości logiczne w języku C

- C89 nie ma typu logicznego
- C99 definiuje typ **`_Bool`**
- Możemy zaemulować przez typ int lub char, używając wartości 0 (fałsz) i 1 lub wartość różna od zera (prawda)
- Dozwolone w instrukcjach sterujących:

```
if ( success == 0 ) {
    printf( "something wrong" );
}
```

- Możemy zdefiniować własne stałe logiczne:

```
#define FALSE 0
#define TRUE  1
```

Wartości logiczne w języku C

- Nie zawsze jest to dobrym pomysłem:

```
if ( success == TRUE ) {
     printf( "everything is a-okay" );
}
```

- **`success`** może być niezerowa, ale nie równa 1; powyższe to NIE to samo, co:

```
if ( success ) {
  printf( "Something is rotten in the state of "
          "Denmark" );
}
```

Typ wyliczeniowy

- Typy wyliczeniowe pozwalają na pogrupowanie logicznie powiązanych ze sobą stałych
  - `enum color {BLACK, RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW, WHITE, COLOR_MAX};`
- Oto kolejny sposób na zdefiniowanie typu logicznego:

```
enum boolean { FALSE, TRUE };
enum boolean eAnswer = TRUE;
```

- Stałe wyliczeniowe traktowane są jak typ całkowity

Typ wyliczeniowy

- Zaczyna się od 0 chyba że jawnie napisano inaczej
  - `enum boolean { FALSE, TRUE };`
  - `enum genre { TECHNO, TRANCE=4, HOUSE };`
- Można równiez przypisać wszystkie wartości
  - `enum channel { TVP1=1, HBO=32, RTL=44 };`
- Nazwy stałych muszą być różne, ale wartości mogą się powtarzać
  - `enum boolean { FALSE=0, TRUE=1, NO=0, YES=1 };`

Typ wyliczeniowy a konstrukcja `typedef`

- Można użyć konstrukcji **`typedef`** w celu zaoszczędzenia miejsca

```
enum boolean { FALSE, TRUE };
typedef enum boolean Eboolean;
EBoolean eAnswer = TRUE;
```

- Jeszcze lepiej jest połączyć **`typedef`** z anonimowym typem wyliczeniowym

```
typedef enum { FALSE, TRUE } Eboolean;
EBoolean eAnswer = TRUE;
```

- Konstrukcje **`typedef`** przydadzą się jeszcze przy okazji struktur i wskaźników do funkcji

Operatory arytmetyczne

- Podstawowe: **`x+y, x-y, x*y, x/y`**
- Uwaga:
  - Niezgodne operandy są konwertowane do "większego" typu:
    `char/short` ⇒ **`int`** ⇒ **`float`** ⇒ **`double`**
  - Dzielenie całkowite obcina część ułamkową:
    - `5/2` ⇒ `2`
    - `5.0/2` ⇒ `2.5`
    - `(float)5/2` ⇒ `2.5`

Modulo (%)

- Znane również jako reszta z dzielenia
  - Powinno być stosowane jedynie dla dodatnich liczb całkowitych
  - Przykłady:
    - `13 / 5 == 2`
    - `13 % 5 == 3`
    - czy x jest nieparzyste?
    - czy x jest podzielne przez y?
    - odwzorowanie kąta 765˚ do zakresu 0˚ - 360˚
    - konwersja 18:45 do formatu 12-godzinnego
    - symulacja rzutu sześcienną kostką
  - Czy rok 2000 był przestępny?
    - Rok jest przestępny, jeżeli jest podzielny przez 4 i niepodzielny przez 100, chyba że jest podzielny przez 400
  - Jak to zapisać?

Przypisania (= i <op>=)

- Przypisanie jest wyrażeniem – jego wartość jest wartością lewej strony po przypisaniu
  - Zwykłe przypisanie: **`x = x + y`**
  - Równoważny sposób: **`x += y`**
  - Więcej: **`x += y, x -= y, x *= y, x /= y, x %= y`**
- Lewa strona operatora przypisania jest obliczana tylko raz

```
(c=getchar()) += 1;
```

to coś innego niż

```
(c=getchar()) = (c=getchar()) + 1;
```

Inkrementacja/dekrementacja

- Pre-inkrementacja/dekrementacja: **`++x, --x`**
- Post-inkrementacja/dekrementacja: **`x++, x--`**
- **`++x`** i **`x++`** działają jak **`x = x + 1`** lub **`x += 1`**
- Różnica uwidacznia się w wyrażeniach
  - **`++x`** najpierw zwiększa, a potem zwraca nową wartość **`x`**
  - **`x++`** zwraca **`x`** najpierw, a potem inkrementuje

```
int x = 0;
assert(x == 0);
assert(++x == 1);
assert(x == 1);
assert(x++ != 2);
assert(x == 2);
```

Operatory bitowe

- Kiedy chcemy zmieniać lub czytać pojedyncze bity
  - Tylko dla typów całkowitych (**`char, short, int, long, unsigned/signed`**)
  - Operatory bitowe:
    - **`&`** Bitowy AND
    - **`|`** Bitowy OR
    - **`^`** Bitowy exclusive OR (XOR)
    - **`<<`** Przesunięcie w lewo
    - **`>>`** Przesunięcie w prawo
    - **`~`** Dopełnienie jedynkowe (operator unarny)
  - Z przypisaniem: **`x &= y, x |= y, x ^= y, x <<= y, x >>= y`**

Operatory bitowe

Przykłady:

- `&` Bitowy AND `0110 & 0011` ⇒ `0010`
- `|` Bitowy OR `0110 | 0011` ⇒ `0111`
- `^` Bitowy XOR `0110 ^ 0011` ⇒ `0101`
- `<<` Przesunięcie w lewo `01101110 << 2` ⇒ `10111000`
- `>>` Przesunięcie w prawo `01101110 >> 3` ⇒ `00001101`
- `~` Dopełnienie jedynkowe `~0011` ⇒ `1100`

- Uwaga: << i >> mnożą/dzielą przez 2^n
- operator >> może nie działać zgodnie z oczekiwaniami dla typów ze znakiem - może wykonywać przesunięcie logiczne lub arytmetyczne (z powieleniem bitu znaku)
- Nie mylić operatorów bitowych `&` `|` z op. logicznymi `&&` `||`

Upakowanie informacji o kolorach w 32 bity

```
/*
 * Format of RGBA colors is
 *  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 *  |    alpha      |      red      |     green     |     blue      |
 *  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 */
#define GET_ALPHA(val)      ((val) >> 24)
#define GET_RED(val)        (((val) >> 16) & 0xff)
#define GET_GREEN(val)      (((val) >> 8) & 0xff)
#define GET_BLUE(val)       ((val) & 0xff)
#define MAKE_ARGB(a,r,g,b) (((a) << 24) | ((r) << 16) | ((g) << 8) | (b))
```

Flagi bitowe

- Można traktować pojedyncze bity jako znaczniki (1=włączone 0=wyłączone)
  - Pozwala to na upakowanie do 32 znaczników w jedną liczbę całkowitą bez znaku
  - Przykład:

```
#define READONLY        0x00000010
#define NOSYSLOCK       0x00000800
#define NOOVERWRITE     0x00001000
#define DISCARD         0x00002000
#define NO_DIRTY_UPDATE 0x00008000
```

  - Używamy `|` aby włączyć flagę
    - `int flags = READONLY | DISCARD;`
  - Używamy `&` aby sprawdzić flagę
    - `if (flags & READONLY) ...`

Operatory logiczne i porównań

- Logiczne:
  - `x == y` Równy
  - `x != y` Nie równy
  - `x && y` logiczne AND
  - `x || y` logiczne OR
  - `!x` NOT
- Porównania:
  - `x < y` Mniejszy niż
  - `x <= y` Mniejszy niż lub równy
  - `x > y` Większy niż
  - `x >= y` Większy niż lub równy

Inne operatory

- **`sizeof`** – podaje rozmiar w bajtach

```
int x = 0;
unsigned size = sizeof(int);     ⇒  4
size = sizeof(x);                ⇒  4
```

- operator wyboru
  - **`x ? y : z`**
  - to notacja skrócona od:
    - if (x) y else z
  - np.: **`z=(a>b)?a:b; /* z = max(a,b) */`**
- przecinek
  - **`x, y`**

Łączność i priorytet

- Dodawanie i odejmowanie są lewostronnie łączne
  - `4 + 5 + 6 + 7` jest równoważne `(((4 + 5) + 6) + 7)`
- Mnożenie, dzielenie i modulo są lewostronnie łączne
  - `4 * 5 * 6 * 7` jest równoważne `(((4 * 5) * 6) * 7)`
- Operatory przypisania są prawostronnie łączne
  - `a = b = c = d` jest równoważne `(a=(b=(c=d)))`
- Dla złożonych wyrażeń z różnymi operatorami, priorytet operatorów określa kolejnośc operacji:

  np.: `c = getchar() != EOF`

  Ponieważ `!=` ma wyższy priorytet niż `=`, powyższe jest równoważne

  `c = (getchar() != EOF)`

  Zdecydowanie nie to, o co nam chodziło!
- W przypadku wątpliwości lub skomplikowanych wyrażeń używamy nawiasów
  - `(c = getchar()) != EOF`

Łączność i priorytet

Operatory	Łączność
`() [] -> .`	lewostronna
`! ~ ++ -- + - * (type) sizeof`	prawostronna
`* / %`	lewostronna
`+ -`	lewostronna
`<< >>`	lewostronna
`< <= > >=`	lewostronna
`== !=`	lewostronna
`&`	lewostronna
`^`	lewostronna
`|`	lewostronna
`&&`	lewostronna
`||`	lewostronna
`?:`	prawostronna
`= += -= *= /= %= &= ^= |= <<= >>=`	prawostronna
`,`	lewostronna

Efekty uboczne i kolejność obliczeń

- Wywołania funkcji, zagnieżdżone instrukcje przypisania oraz operatory inkrementacji i dekrementacji mają efekty uboczne - pewne zmienne ulegają zmianie przy okazji obliczania wyrażenia.
- Jeżeli wyrażenie ma efekty uboczne, mogą wystąpić subtelne zależności wyniku operacji od kolejności uaktualniania zmiennych występujących w wyrażeniu.
- Standard C nie specyfikuje kolejności obliczania argumentów operatorów z wyjątkiem operatorów **`&&`**, **`||`**, **`?:`**, i **`','`**. W wyrażeniu jak

```
x = f() + g();
```

`f` może być wywołane przed **`g`** lub odwrotnie.

- W celu zapewnienia określonej kolejności obliczeń należy przechować wyniki pośrednie w zmiennych tymczasowych obliczanych oddzielnie.
- Kolejność obliczania argumentów funkcji nie jest określona, a więc instrukcja

```
printf("%d %d\n", ++n, power(2, n));/* WRONG */
```

może dać różne wyniki przy kompilacji różnymi kompilatorami.

- Inna typowa sytuacja tego rodzaju jest reprezentowana przez wyrażenie:

```
a[i] = i++;
```

Instrukcje sterujące - przegląd

- Wyrażenia, instrukcje i bloki
- **`if`**, **`else`**
- **`switch`**
- Pętle
  - **`while`**
  - **`do-while`**
  - **`for`**
  - **`break`** i **`continue`**
- **`goto`** i etykiety

Wyrażenia, instrukcje i bloki

- Widzieliśmy już wiele przykładów
  - Wyrażenia zwracają wartość: **`x + 1`**, **`x == y`**, itd.
  - Instrukcje to wyrażenia zakończone ;
  - Nawiasy klamrowe **`{ }`** są używane do grupowania instrukcji w bloki
  - Bloki mogą również być użyte jako ciała funkcji i w instrukcjach **`if`**, **`else`**, **`while`**,**`for`**, itd.

Instrukcja `if`

- Prosta instrukcja if

```
if (eDay == eMONDAY)
 printf("I hate Mondays!\n");
```

- if-else

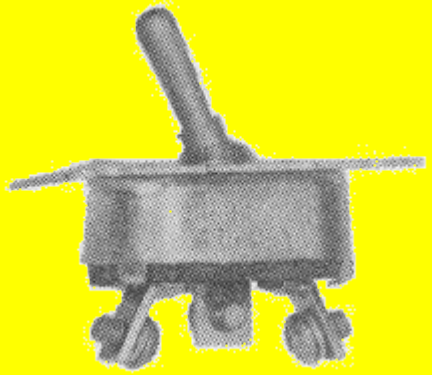
```
if (eDay == eMONDAY)
 printf("I hate Mondays!\n");
else
 printf("How soon 'till the weekend?\n");
```

- if-else-if-else

```
if (eDay == eMONDAY)
 printf("I hate Mondays!\n");
else if (eDay == eWEDNESDAY)
 printf("The weekend is in sight!\n");
else
 printf("How soon 'till the weekend?\n");
```

Instrukcja `switch`

```
int c = getchar();
switch (c)
{
 case '?':
  printf("Please answer Y or N\n");
  break;
 case 'y': case 'Y':
  printf("Answer is yes\n");
  break;
 case 'n': case 'N':
  printf("Answer is no\n");
  break;
 default:
  printf("By default, the answer is maybe\n");
  break;
}
```



- Wybór z wielu możliwości
- Uwaga: Przypadki z kilkoma instrukcjami nie wymagają nawiasów klamrowych
- **`default`** jest opcjonalne, ale przeważnie występuje
- Nie zapomnij break!

while i do-while

- Widzieliśmy już przykład

```
while((c = getchar()) != EOF)
  ...
```

  - **while** sprawdza warunek i wykonuje ciało instrukcji
  - **do-while** wykonuje ciało instrukcji i dopiero wtedy sprawdza warunek

```
int nDone = 0;
do {
  ...
} while (!nDone);
```

Instrukcja `for`

- Zwięzła instrukcja pętli

```
for(expr1; expr2; expr3)
{
  statements
}
```

  - Jest to równoważne

```
expr1;
while (expr2)
{
  statements
  expr3;
}
```

  - **`expr1, expr2, expr3`** są opcjonalne

Instrukcja `for` – przykłady

- Wypisz 4 spacje

```
for(i = 0; i < 4; ++i)
  putchar(' ');
```

- Wypisz alfabet

```
for(c = 'a'; c <= 'z'; ++c)
  printf("%c ", c);
```

- Wypisz liczby parzyste między 0 i 100 włącznie

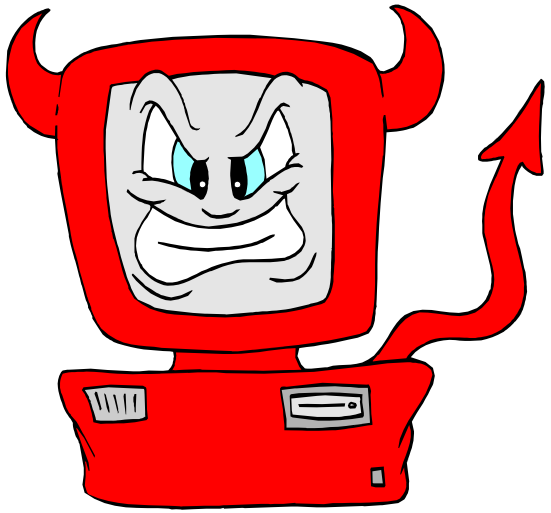
```
for(n = 0; n <= 100; n += 2)
  printf("%d ", n);
```

- Kiedy używać **`while`**, **`do-while`**, **`for`**?

`break` i `continue`

- **`break`**
  - Używamy **`break`** aby zakończyć pętlę (**`while`**, **`do-while`**, **`for`**)
  - Przejście do pierwszej instrukcji za pętlą
- **`continue`**
  - Pomija pozostałe instrukcje w ciele pętli
  - Przechodzi do sprawdzenia warunku pętli (**`while`** i **`do-while`**) lub do **`expr3`** (**`for`**)

Instrukcja `goto` i etykiety



```
goto label;
...
label:
```

- Przejście do instrukcji po etykiecie
- Używane w nadmiarze jest szkodliwe i prowadzi do tzw. *spaghetti code*
- Dwa przypadki, gdy stosowanie jest dozwolone:
  - Wyjście z zagnieżdżonej pętli
  - Wykonywanie kodu "sprzątającego" przed wyjściem z funkcji
- Nie wykonywać skoków do tyłu! Do tego służą instrukcje pętli.

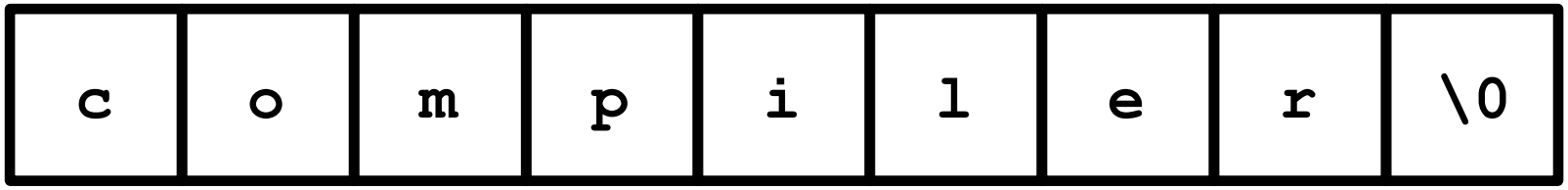
Tablice

- Najprostsze agregaty
- Stała długość (tablice dynamiczne omówimy później)
  - Wszystkie elementy tego samego typu
    - Powinny oznaczać wielkości jednego rodzaju
  - Rodzaje tablic
    - Znakowe (łańcuchy)
    - Tablice innych typów
    - Wielowymiarowe

Tablice znakowe (“łańcuchy”)

```
const char szMsg[] = "compiler";
```

- Zmienna jest przechowywana w tablicy znaków zakończonej znakiem `'\0'` (**NUL**)



- Pierwszy element tablicy ma indeks 0
  - **`szMsg[3]`** odnosi się do czwartego znaku (nie 3.) ⇒ **`'p'`**
  - **`sizeof(szMsg)`** = rozmiar tablicy w bajtach = **9**
    (nie zapominajmy o `'\0'`!)
- Liczba elementów
  - = rozmiar tablicy / rozmiar elementu
  - = **`sizeof(szMsg)/sizeof(char)`**
  - = **`sizeof(szMsg)/sizeof(*szMsg)`**

Tablice znakowe

- Utwórzmy inny łańcuch

```
char szMyMsg[4];
szMyMsg[0] = 'f';
szMyMsg[1] = 'o';
szMyMsg[2] = 'o';
szMyMsg[3] = '\0'; /* Did you forget this? */
```

  - Poniżej jeszcze jedna metoda inicjalizacji łańcucha

```
char szMyMsg[] = { 'f', 'o', 'o', '\0' };
```

Inne tablice i ich inicjalizacja

- Tablice mogą zawierać elementy inne niż znaki, włączając także tablice!

```
int aryDigitCount[10];  /* uninitialized array */
```

  - Możemy zainicjalizować tablicę używając notacji ={}

```
int aryDays[]= { 31, 28, 31, 30, 31, 30, 31,
31, 30, 31, 30, 31};
```

  - W tym przypadku możemy pominąć liczbę elementów, bo kompilator sam może ją określić.
  - Jeżeli liczba elementów jest jawnie określona i liczba inicjalizatorów jest mniejsza niż liczba elementów, kompilator zainicjalizuje pozostałe elementy zerami. Pozwala to na wygodną inicjalizację całej tablicy zerami:

```
int aryDigitCount[10] = { 0 };
```

  - Powinniśmy zawsze inicjalizować automatyczne tablice, nie możemy zakładać, ze zostaną one zainicjalizowane zerami.

Rozmiar tablicy

- Mając łańcuch, jak określić jego długość?
  - Mając tablice dowolnego typu, jak określić liczbę elementów?
  - Nie możemy użyć **`sizeof`** jeżeli tablica jest parametrem funkcji
  - Liczbę elementów tablicy zazwyczaj określamy przy pomocy:
    - elementu terminującego (**`'\0'`** dla łańcuchów, **`0`** dla **`argv`**)
    - osobnej zmiennej licznika (np. **`argc`**)
    - licznika zakodowanego w danych (np. **`BSTR`**)
    - stałej (np. **`MAX_SIZE`**)
  - Jak napisać **`strlen()`**?
  - Jaka jest wada użycia elementu terminującego?

Tablice dwuwymiarowe

```
char arySmiley[4][8] = {
" -- -- ",
" @    @ ",
"    +    ",
" |---/ ", /* trailing comma is legal */
};
```

- To tablica czterech łańcuchów ośmioelementowych (nie zapominajmy o `\0`!)
- Tablica dwuwymiarowa jest tablicą jednowymiarową, której każdy element jest tablicą
- Jaki jest rozmiar w bajtach?

Tablice dwuwymiarowe

- Pokolorujmy nasz symbol
  - Przechowajmy wartość składowych RGB, upakowanych w liczbie typu **`int`** w postaci **`0x0rgb`**, dla każdego elementu

```
/* Initialize all colors to black */
unsigned long arySmileyColors[4][7] = { 0L };

/* Paint eyebrows, nose, and chin white */
arySmileyColors[0][1] = 0xFFFFFFL;
arySmileyColors[0][2] = 0xFFFFFFL;
arySmileyColors[0][4] = 0xFFFFFFL;
arySmileyColors[0][5] = 0xFFFFFFL;
arySmileyColors[2][3] = 0xFFFFFFL;
arySmileyColors[3][1] = 0xFFFFFFL;
arySmileyColors[3][5] = 0xFFFFFFL;
```

  - Jaki kolor mają oczy i usta?
  - Dlaczego tylko 7 **`int`**ów, jeżeli było 8 **`char`**ów?

Problemy z tablicami

- Wolno odwoływać się jedynie do poprawnych elementów tablicy!
  - Czytanie/pisanie elementów które są poza zakresem tablicy ma w języku C niezdefiniowane konsekwencje!
  - **`ary[-1]`** i **`ary[999]`** nie wygenerują żadnych błędów kompilacji
  - Jeżeli mamy szczęście(!), program zakończy się z komunikatem:

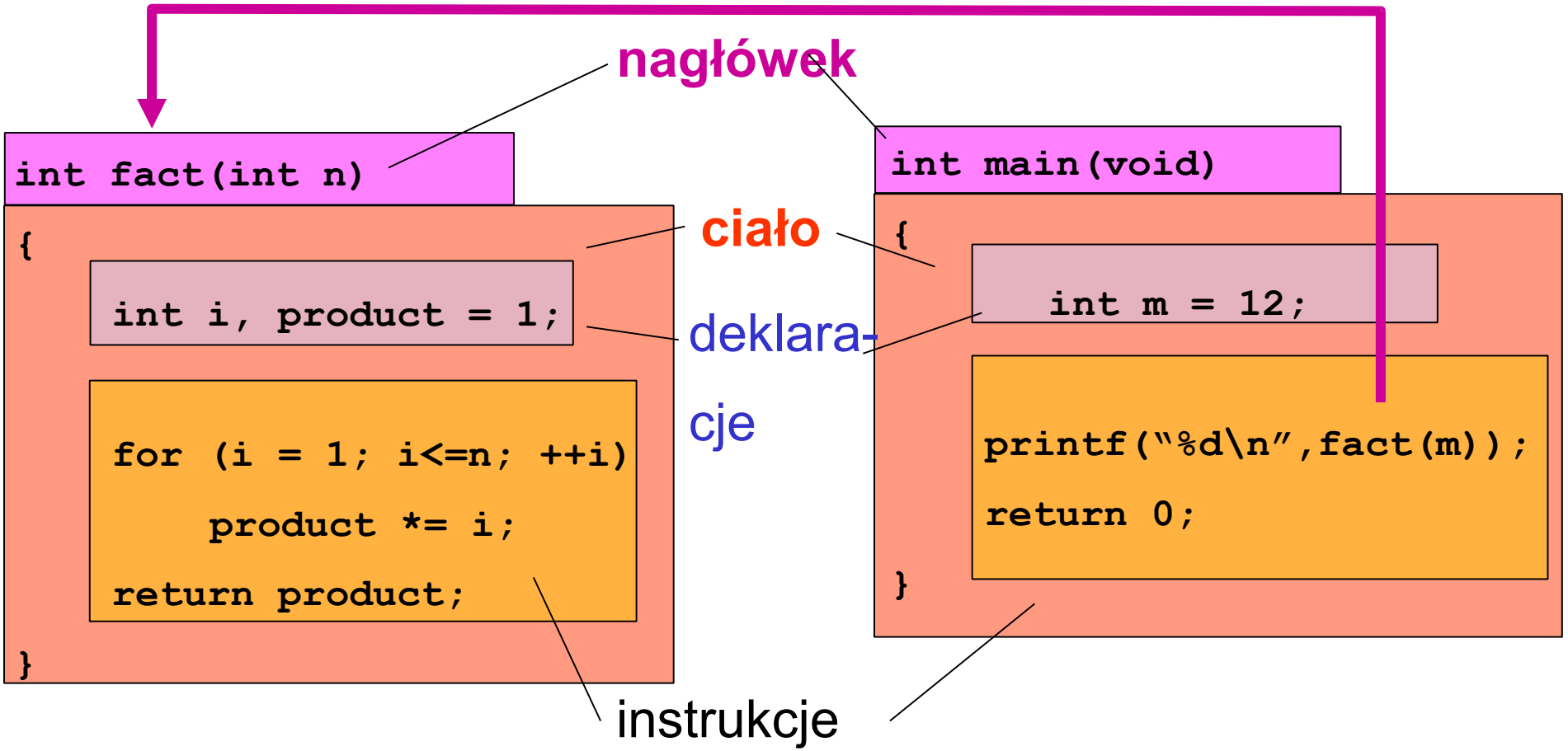
    `Segmentation fault (core dumped)`
  - Co jest nie tak w kodzie poniżej?

```
int aryChrCount[26] = { 0 };   /* A-Z */
char c = '\0';
while ((c = getchar()) != EOF)
  ++aryChrCount[c];
```

Definicja funkcji

format
definicji funkcji:

```
type func_name( parameter_list )
{
   declarations
   statements
}
```



Nagłówek funkcji

```
type   func_name( parameter_list )
```

nazwa funkcji

lista argumentów:

```
type parameter_name
```

typ zwracany przez funkcję

(**`void`** jeżeli funkcja nie zwraca wartości)

argumenty oddziela się przecinkami

`void` jeżeli brak parametrów

Przykłady:

```
int fact(int n)

void error_message(int errorcode)

double initial_value(void)

int main(void)
```

Użycie:

```
a = fact(13);

error_message(2);

x=initial_value();
```

Po co funkcje?

- Piszemy program w postaci zbioru niewielkich funkcji w celu jego modularyzacji.
  - programowanie strukturalne
  - kod łatwiejszy do uruchomienia
  - łatwiejsza modyfikacja
  - możliwość ponownego użycia w innych programach

Prototypy funkcji

- Jeżeli funkcja nie jest zdefiniowana przed jej uzyciem, musi zostać zadeklarowana poprzez wyspecyfikowanie typu wartości zwracanej i typów argumentów

```
double sqrt(double);
```

  - informuje kompilator, że funkcja **`sqrt()`** pobiera argument typu **`double`** i zwraca wartość typu **`double`**.
  - Powoduje to, że zmienne będą rzutowane do właściwego typu, a więc **`sqrt(4)`** zwróci poprawną wartość mimo iż **`4`** jest typu **`int`**, a nie **`double`**.
- Prototypy funkcji są umieszczane zazwyczaj na początku programu albo w oddzielnym pliku nagłówkowym, **`file.h`**, dołączanym poprzez **`#include "file.h"`**
- Nazwy zmiennych na liście argumentów w deklaracji funkcji są opcjonalne:

```
void f (char, int);
void f (char c, int i); /*equivalent but makes code more readable */
```

- Jeżeli wszystkie funkcje są zdefiniowane przed ich użycie, nie potrzeba żadnych prototypów funkcji. W tym przypadku **`main()`** jest ostatnią funkcją w programie.

Wywołanie funkcji

- W momencie wywołania funkcji:
  - wyrażenia na liście parametrów są obliczane (bez zachowania żadnej szczególnej kolejności!)
  - wyniki obliczeń są przekształcane do właściwych typów
  - parametry są kopiowane do lokalnych zmiennych funkcji
  - ciało funkcji jest wykonywane
  - po napotkaniu instrukcji return funkcja jest zakończona, a wynik (podany w instrukcji return) jest przekazywany do funkcji wywołującej (na przykład main)

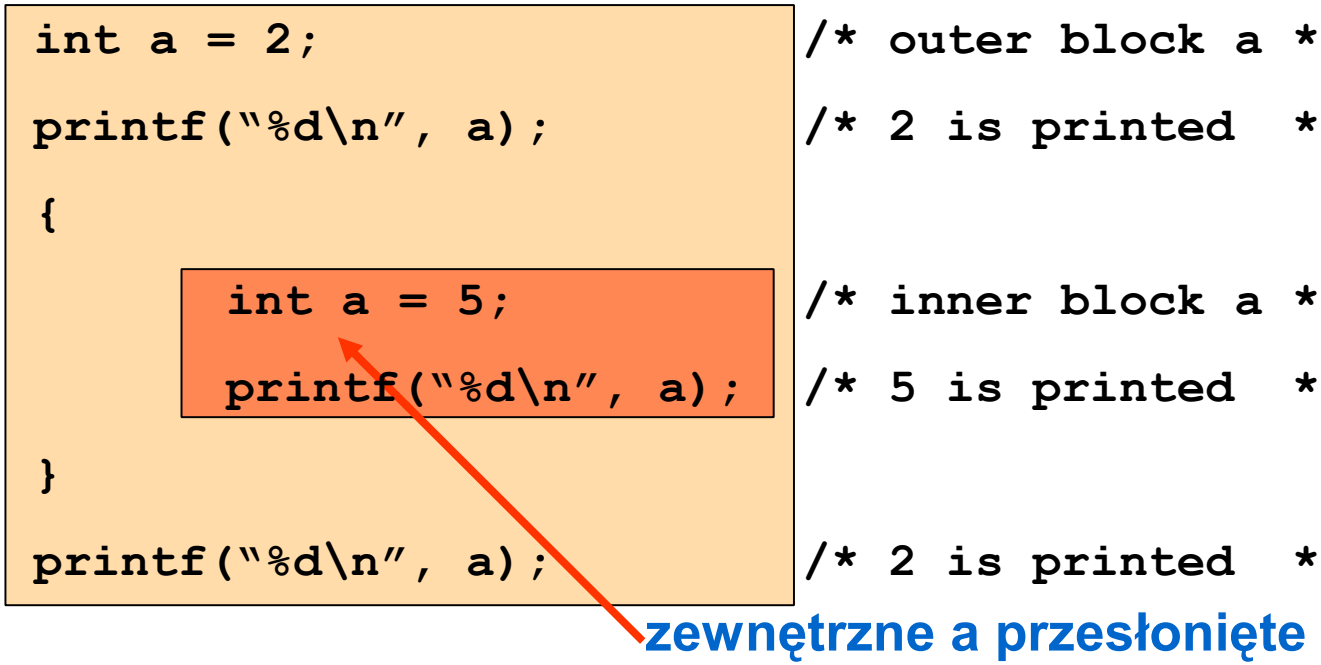
```
int fact (int n)
{
    int i, product = 1;
    for (i = 2; i <= n; ++i)
                product *= i;
    return product;
}
```

```
int main (void)
{
    int i = 12;
    printf("%d",fact(i));
    return 0;
}
```

Reguły zasięgu dla bloków

- Identyfikatory (tj. zmienne itd.) są dostępne tylko wewnątrz bloku, w którym są zadeklarowane.

```
{
    int a = 2;                  /* outer block a */
    printf("%d\n", a);          /* 2 is printed  */
    {
        int a = 5;              /* inner block a */
        printf("%d\n", a);      /* 5 is printed  */
    }
    printf("%d\n", a);          /* 2 is printed  */
}
/* a no longer defined */
```



- Zmienna zadeklarowana w zewnętrznym bloku jest dostępna w wewnętrznym bloku o ile nie jest ponownie zadeklarowana. W tym przypadku zewnętrzna deklaracja jest przesłonięta.
- Unikaj przesłaniania nazw zmiennych!
  Używaj różnych nazw zmiennych w celu ułatwienia uruchamiania!

Reguły zasięgu dla funkcji

- Zmienne zdefiniowane wewnątrz funkcji (włączając main) są lokalne dla tej funkcji i niedostępne z innych funkcji!
  - zmienne można przekazywać do funkcji jako parametry
  - jedyną metodą przekazania (pojedynczej) zmiennej z funkcji jest uzycie instrukcji return

```
int func (int n)
{
   printf("%d\n",b);

   return n;
}
```

```
int main (void)
{
   int a = 2, b = 1, c;

   c = func(a);

   return 0;
}
```

b nie zdefiniowane lokalnie!

- Wyjątki:
  - Zmienne globalne
  - Wskaźniki

Zmienne globalne

- Zmienne zdefiniowane poza blokami są globalne, tj. dostępne dla wszystkich późniejszych bloków i funkcji
- Unikaj używania zmiennych globalnych w celu przekazania parametrów do funkcji!
- Tylko kiedy wszystkie zmienne w funkcji są lokalne może ona być używana w różnych programach
- Zmienne globalne są mylące w długim programie

```
#include <stdio.h>

int a = 1, b = 2; /* global variables */

int main (void)
{
   int b = 5;   /* local redefinition */
   printf("%d", a+b); /* 6 is printed */
   return 0;
}
```

Przekazywanie parametrów przez wartość

- Oblicza się argumenty funkcji i kopie obliczonych wartości - nie zmienne podane jako argumenty - są przekazywane do funkcji
  - Zaleta: chroni to zmienne w funkcji wywołującej
  - Wada: kopiowanie nieefektywne, np. dla dużych tablic ⇒ wskaźniki

```
#include <stdio.h>
int     compute_sum (int n);   /* function prototype          */
/* ------------------------------------------------------------ */
int main(void)
{
        int n = 3, sum;
        printf("%d\n", n);     /* 3 is printed                */
        sum = compute_sum(n);  /* pass value 3 down to func */
        printf("%d\n", n);     /* 3 is printed -  unchanged */
        printf("%d\n",sum);    /* 6 is printed                */
        return 0;
}
/* ------------------------------------------------------------ */
int compute_sum (int n)        /* sum integers 1 to n         */
{
        int sum = 0;
        for ( ; n > 0; --n)    /* local value of n changes   */
                sum += n;
        return sum;
}
```

n niezmienione

lokalna kopia n, niezależna od n w funkcji wywołującej

Klasy pamięci

- Każda zmienna i funkcja w C ma dwa atrybuty:
  - typ (**`int, float`**, ...)
  - klasę pamięci
- Klasa pamięci jest związana z zasięgiem zmiennej
- Są cztery klasy pamięci:
  - **`auto`**
  - **`extern`**
  - **`register`**
  - **`static`**
- **`auto`** jest domyślna i najpowszechniejsza
- Pamięć dla zmiennych automatycznych jest alokowana po wejściu do bloku lub funkcji. Są one lokalne w bloku. Po wyjściu z bloku, system zwalnia pamięć zaalokowaną dla zmiennych **`auto`** a ich wartość jest tracona.
- Deklaracja:
  - **`auto type variable_name;`**
- Zazwyczaj jawnie nie używa się słowa kluczowego **`auto`**

extern

- Zmienne globalne (zdefiniowane poza funkcjami) i wszystkie funkcje mają klasę pamięci **`extern`** lub **`static`** i przypisany na stałe obszar pamięci
- Aby uzyskać dostęp do zmiennej zewnętrznej, zdefiniowanej gdzie indziej, używamy następującej deklaracji:

  `extern type variable_name;`

  - informuje to kompilator, że zmienna **`variable_name`** o klasie pamięci **`extern`** jest zdefiniowana gdzieś w programie
- Zmienne zdefiniowane w pliku poza funkcjami mają klasę pamięci **`extern`**
- Program może składać się z wielu plików kompilowanych oddzielnie.
  `extern` jest używany dla zmiennych globalnych używanych w kilku różnych plikach

`extern` w projektach złożonych z wielu plików

```
/*file1.c*/
#include <stdio.h>
int a =1, b = 2, c = 3; /* external variables */
int f(void);
int main (void)
{
    printf("%3d\n", f( ));
    printf("%3d%3d%3d\n", a, b, c);
    return 0;
}

/*file2.c*/
int f(void)
{
    extern int a;  /* look for it elsewhere */
    int b, c;
    a = b = c = 4;
    return (a + b + c);
}
```

wypisuje 4, 2, 3

a jest globalne i zmieniane przez f

b i c są lokalne i nie przeżywają

zwraca 12

- kompilacja: **`gcc file1.c file2.c –o prog`**

static

- Zmienne statyczne to zmienne lokalne które zachowują swoją poprzednią wartość po ponownym wejściu do bloku. Deklaracja
  - `static int cnt = 0;`

  ustawi cnt na zero kiedy funkcja będzie użyta po raz pierwszy; za kolejnymi razami zmienna zachowa wartość z poprzedniego wywołania.
- Może być to przydatne np. podczas uruchamiania programu, w dowolnym miejscu możemy wstawić kod taki jak poniżej bez wpływu na resztę programu

```
{ /* debugging starts here */
  static int cnt = 0;
  printf("*** debug: cnt = %d, v = %d\n",++cnt, v);
}
```

  Zmienna cnt jest lokalna dla bloku i nie będzie kolidować z żadną inną zmienną o tej samej nazwie w bloku zewnętrznym; będzie się ona zwiększać o jeden przy każdym wejściu do bloku.
- **`static`** można również zastosować do zmienych globalnych; oznacza to, że są one lokalne dla pliku i niedostępne z innych plików.
- Jeżeli nie zainicjalizowano ich jawnie, zmienne globalne i statyczne są inicjalizowane wartością 0

Rekursja

- Aby zrozumieć rekursję, trzeba najpierw zrozumieć rekursję.
- Funkcja jest nazywana rekursywną, jeżeli wywołuje sama siebie, bezpośrednio lub pośrednio.
- W C wszystkie funkcje mogą być wywołane rekursywnie.
  - Przykład:

```
int sum(int n)
{
  if (n <= 1)
   return n;
  else
   return (n + sum(n - 1));
}
```

- Jeżeli nie chcemy wygenerować pętli nieskończonej, musimy dostarczyć warunek zakończenia rekursji (tutaj **`n<=1`**), który będzie kiedyś spełniony.
- Rekursja jest często nieefektywna, gdyż wymaga wielu wywołań funkcji.

Przykład: liczby Fibonacciego

- Funkcja rekursywna obliczająca liczby Fibonacciego (0,1,1,2,3,5,8,13...)

```
int fibonacci(int n)
{
    if (n <= 1)
       return n;
    else
       return (fibonacci(n-1) + fibonacci(n-2));
}
```

- Potrzeba 1.4 x 10^9 wywołań funkcji aby obliczyć 43. liczbę Fibonacciego! (która ma wartość 433494437)
- Jeżeli to możliwe, lepiej napisać funkcję iteracyjną

```
int factorial (int n)   /* iterative version */
{
    for ( ; n > 1; --n)
           product *= n;
    return product;
}
```

Asercje

Po zastosowaniu dyrektywy

`#include <assert.h>`

można używać makra “assert”: kończy ono program jeżeli warunek asercji (niezmiennik algorytmu) nie jest spełniony.

Można wyłączyć asercje poprzez zdefiniowanie stałej **`NDEBUG`** (**`#define NDEBUG`**).

```
#include <assert.h>
#include <stdio.h>
int f(int a, int b);
int g(int c);

int main(void)
{
int a, b, c;
.....
scanf("%d%d", &a, &b);
.....
c = f(a,b);
assert(c > 0);/* an assertion */
.....
}
```