



Programowanie zaawansowane FM i NI

Wykład 3-4

Referencje, typy, operatory, wskaźniki

Krzysztof Piasecki

Semestr letni roku akad. 2023-24



- **Referencja** – to nowa nazwa na istniejącą zmienną
- Składnia na prostym przykładzie: (uwaga: &)

```
int A = 5 ;  
int& Aref = A ;  
Aref = 7;      (sprawi, że zarazem A = 7 )
```

Nie da się zadeklarować 'pustej' referencji:

```
--int& Bref--;
```

Nie da się 'przełączyć' referencji na inną zmienną. To:

```
int C = 5 ;   Aref = C ;
```

będzie znaczyć tylko tyle, że do A przypiszemy 5.

- Główna użyteczność:
przy przekazywaniu zmiennej do podfunkcji.

Bez referencji - wywołana funkcja działa na **kopii** zmiennej podanej w wywołaniu. Zmiany w tej kopii nie zmieniają oryginału.

Działając na referencji - działamy na **oryginale**. Tylko tak zmiana wartości zmiennej w funkcji będzie się liczyć po powrocie do miejsca wywołania.

```
10  
12
```

```
1  #include<iostream>  
2  using namespace std;  
3  
4  void Add_pass_via_Copy (int X) {  
5      X    += 2;  
6  }  
7  
8  void Add_pass_via_Ref (int& Xref) {  
9      Xref += 2;  
10 }  
11  
12 int main () {  
13     int x = 10 ;  
14     Add_pass_via_Copy ( x );  
15     cout << x << endl;  
16  
17     x = 10 ;  
18     Add_pass_via_Ref ( x );  
19     cout << x << endl;  
20 }
```

- **Arytmetyczne typy danych** [[użyteczny link](#)]

① **bool**: 1 (true) lub 0 (false)

② Typy **całkowite**. Zmienna zajmuje 1-kilka bajtów (uwaga: **1 bajt = 8 bitów**).

Pierwszy bit pierwszego bajtu może służyć jako **znak** liczby. Wtedy typ ma cechę **signed**.

To sytuacja domyślna. Ale można zażądać **unsigned**: liczba jest ≥ 0 , ale jej maksimum $2 \times$ większe.

Typ	Rozmiar [bity]		Zakres (dla LP64)	
	Standard C	Model danych LP64	Min	Max
(signed) char	≥ 8	8	-128	127
unsigned char	≥ 8	8	0	255
(signed) short	≥ 16	16	-32768	32767
unsigned short	≥ 16	16	0	65535
(signed) int	≥ 16	32	$-2.14 \cdot 10^9$	$2.14 \cdot 10^9$
unsigned int	≥ 16	32	0	$4.29 \cdot 10^9$
(signed) long	≥ 32	64	$-9.22 \cdot 10^{18}$	$9.22 \cdot 10^{18}$
unsigned long	≥ 32	64	0	$1.84 \cdot 10^{19}$
(signed) long long	≥ 64	64	$-9.22 \cdot 10^{18}$	$9.22 \cdot 10^{18}$
unsigned long long	≥ 64	64	0	$1.84 \cdot 10^{19}$

Słowa kluczowe signed / unsigned to tzw. **modyfikatory typu** (type modifiers).

- **Sprawdzanie typów:**

Rozmiar zmiennej w bajtach:

→ `sizeof`

Identyfikator typu zmiennej:

→ `typeid` w `<typeinfo>`

(Typy można porównywać.)

Wartość minimalna i maksymalna:

→ stałe w `<climits>`

```
1 4 1 4
int has type i
c has type c
[types compared]: i has type int
short= [-32768 : 32767]
int = [-2147483648 : 2147483647]
```

```
1 #include <iostream>
2 #include <typeinfo>
3 #include <climits>
4 using namespace std;
5
6 int main () {
7     char c = 65 ;
8     int i = -1234 ;
9
10    cout << sizeof (char) << ' '
11         << sizeof (int ) << ' '
12         << sizeof ( c ) << ' '
13         << sizeof ( i ) << "\n";
14
15    cout << "\nint has type " << typeid (int).name();
16    cout << "\nc has type " << typeid ( c ).name();
17
18    if ( typeid (i) == typeid (int) )
19        cout << "\n[types compared]: i has type int\n";
20
21    cout << "\nshort= [" << SHRT_MIN
22         << " : " << SHRT_MAX << ']' ;
23    cout << "\nint = [" << INT_MIN
24         << " : " << INT_MAX << ']' ;
25 }
```

- **Reprezentacja** liczby całkowitej w pamięci

Użyteczna biblioteka `<bitset>`
 Daje obiekty N-bitowe do łatwego wypisywania w systemie dwójkowym.

Potestujmy np. reprezentację zmiennych typu `short` (w LP64 mieści 2 bajty).
 Jest to typ ze znakiem (signed short)
 → lewy bit 1. bajtu opisuje znak.

Największa wartość to: $2^{15} - 1 = 32767$
 Najmniejsza to: $-2^{15} = -32768$

```
(signed) shorts:
0      |_2 = 0000000000000000
1      |_2 = 0000000000000001
2      |_2 = 0000000000000010
32767  |_2 = 0111111111111111
Attempt to assign int > SHRT_MAX:
-32768 |_2 = 1000000000000000

-1     |_2 = 1111111111111111
-2     |_2 = 1111111111111110
-32768 |_2 = 1000000000000000
```

```
1  #include <iostream>
2  #include <bitset>
3  #include <climits>
4  using namespace std;
5
6  void bitprint (short S) {
7      bitset<16> bits (S) ;
8      cout << S << "\t|_2 = " << bits << endl;
9  }
10
11 int main () {
12     short s0 = 0, s1 = 1, s2 = 2,
13           s_1 = -1, s_2 = -2;
14     cout << "(signed) shorts: \n";
15     bitprint ( s0 );
16     bitprint ( s1 );
17     bitprint ( s2 );
18     bitprint ( SHRT_MAX );
19     cout << "Attempt to assign int > SHRT_MAX:\n";
20     bitprint ( 32768 ); cout << endl;
21     bitprint ( s_1 );
22     bitprint ( s_2 );
23     bitprint ( SHRT_MIN );
24 }
```

- **Reprezentacja** liczby całkowitej c.d.

Rozważmy teraz `unsigned short`
(nadal w modelu LP64).

Najmniejsza wartość to 0

Największa to $2^{16} - 1 = 65535$

```
unsigned shorts:
0      |_2 = 0000000000000000
1      |_2 = 0000000000000001
2      |_2 = 0000000000000010
65535  |_2 = 1111111111111111
Attempt to assign int > USHRT_MAX:
1      |_2 = 0000000000000001
Attempt to assign -1 :
65535  |_2 = 1111111111111111
```

```
1  #include <iostream>
2  #include <bitset>
3  #include <climits>
4  using namespace std;
5
6  void bitprint_unsigned (unsigned short US) {
7      bitset<16> bitUS (US) ;
8      cout << US << "\t|_2 = " << bitUS << endl;
9  }
10
11 int main () {
12     unsigned short us0 = 0, us1 = 1, us2 = 2,
13         us_overmax = 65536, us_minus1 = -1;
14     cout << "\nunsigned shorts:\n";
15     bitprint_unsigned ( us0 );
16     bitprint_unsigned ( us1 );
17     bitprint_unsigned ( us2 );
18     bitprint_unsigned ( USHRT_MAX );
19     cout << "Attempt to assign int > USHRT_MAX:\n";
20     bitprint_unsigned ( 65536 );
21     cout << "Attempt to assign -1 :\n";
22     bitprint_unsigned ( -1 );
23 }
```

- Wstawianie do zmiennej całkowitej wartości w reprezentacjach o bazie 2, 8, 16 (i wypis na ekran w tych formatach).

- ▶ Wartość poprzedzona przez **0** to liczba w systemie 8-kowym.
Np.:

$$0100 = 64_{(10)}$$

- ▶ Wartość poprzedzona przez **0x** to liczba w systemie 16-tkowym.
Np.:

$$0x100 = 256_{(10)}$$

- ▶ Wartość poprzedzona przez **0b** to liczba w systemie binarnym.
Np.:

$$0b1000 = 8_{(10)}$$

```
Hexadecimal rep.: 40
Decimal rep.: 64
Octal rep.: 100
Binary 8bit rep.: 01000000

For 0100, decimal rep.: 64
For 0x40, decimal rep.: 64
For 0b1000000, decimal rep.: 64
```

```
1 #include <iostream>
2 #include <bitset>
3 using namespace std;
4
5 int main () {
6     int x = 64;
7     cout << "Hexadecimal rep.: " << hex << x << endl;
8     cout << "Decimal rep.: " << dec << x << endl;
9     cout << "Octal rep.: " << oct << x << endl;
10    cout << "Binary 8bit rep.: " << bitset<8> (x) << "\n\n";
11
12    int octX = 0100;
13    cout << "For 0100, decimal rep.: "
14         << dec << octX << endl;
15
16    int hexX = 0x40;
17    cout << "For 0x40, decimal rep.: "
18         << dec << hexX << endl;
19         // Guaranteed since C++14
20    int binX = 0b1000000;
21    cout << "For 0b1000000, decimal rep.: "
22         << dec << binX << endl;
23 }
```

[Link]

- Casus typu **char**. Zmiennej tego typu można używać:

- do przechowywania i arytmetyki liczb w zakresie $\in [-128, 127]$ (przy unsigned char: $[0, 255]$)
- do przechowywania i wyświetlania znaków z **tablicy kodów ASCII** :

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

- Casus typu **char**. Zmiennej tego typu można używać:
 - do przechowywania i arytmetyki liczb w zakresie $\in [-128, 127]$ (przy `unsigned char`: $[0, 255]$)
 - do przechowywania i wyświetlania znaków z **tablicy kodów ASCII**.

Rozważmy np. `char c = 65, d = 50, e = 't';`

Efekty uzależnione od sytuacji:

- ▶ `e = c + d;` → `{ e = 115 }` (każdy operator algebraiczny: działania na liczbach)
- ▶ `if (e == 115)` → `{ true }` (porównywanie z liczbą działa)
- ▶ `cout << e;` → `s` (podanie na `cout`: wyświetli się znak wg tablicy ASCII)
- ▶ `if (e == 's')` → `{ true }` (porównywanie ze znakiem też działa)
- ▶ `cin >> e;` (gdy użytkownik poda znak, zmienna przyjmie wartość wg ASCII)

Uwaga: znak do zmiennej typu `char` przypisujemy zawsze przez `' '` (nigdy przez `" "`)

- **Kody ucieczki (escape sequences)**: to specjalne kody sterujące. Pełna lista **tu**, a poniżej – najważniejsze:

<code>\n</code>	przejdź na początek nowej linii	<code>\"</code>	znak graficzny cudzysłowu
<code>\r</code>	przejdź na początek aktualnej linii	<code>\'</code>	znak graficzny apostrofu
<code>\v</code>	przejdź do nowej linii; ta sama kolumna	<code>\u...</code>	znak Unicode o danym kodzie (hex)
<code>\t</code>	wykonaj tabulator	<code>\a</code>	daj sygnał dźwiękowy

Enum(-eracja) : typy wyliczeniowe

Czasem potrzeba zmiennej mieszczącej tylko kilka możliwości, np. dni tygodnia, miesiące, główne kolory itp.

Poprzez **enum** definiujemy odrębny typ, przypisując nazwy własne liczbom (tu: typ `WeekDay`). Można też wylistować nazwy nie przypisując liczb – wtedy komputer przypisze im kolejne liczby całkowite, od 0.

Zmienna typu `WeekDay` może przyjąć dowolną z tych możliwości. Można ją też przestać do funkcji.

Dostęp do nazw naszego enum-a :

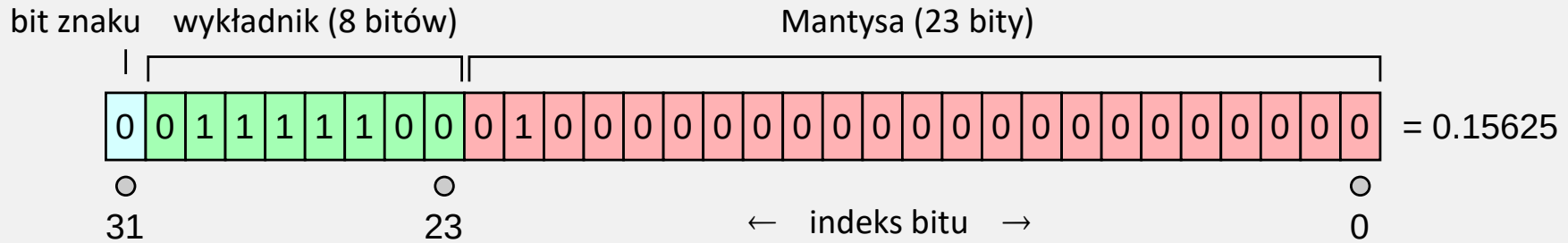
- ▶ przy braku konfliktów : Nazwa
 - ▶ aby uniknąć konfliktów : Typ::Nazwa
- Takie wartości może zwracać funkcja lub można nimi manipulować.

Każdą z nazw komputer skonwertuje do liczby. Gdyby porównywane były nazwy z odmiennych enum'ów, kompilator wyda ostrzeżenie.

```
1 #include <iostream>
2 using namespace std;
3
4 enum WeekDay {
5     Monday    = 1, Tuesday   = 2,
6     Wednesday = 3, Thursday  = 4,
7     Friday    = 5, Saturday  = 6,
8     Sunday    = 7
9 };
10 enum Logic { Untrue , True };
11
12 Logic IsWeekend (WeekDay day) {
13     if (day <= Friday) return Logic::Untrue;
14     else                return Logic::True ;
15 }
16
17 int main () {
18     WeekDay Today = Friday;
19     cout << IsWeekend ( Today ) << endl;
20
21     switch ( IsWeekend(Today) ) {
22     case Logic::True : cout << "Today's weekend!\n";
23                       break;
24     default          : cout << "Work harder :P  \n";
25     }
26     return 0;
27 }
```

③ Typy zmiennoprzecinkowe są (zwykle) oparte o standard IEEE-754.

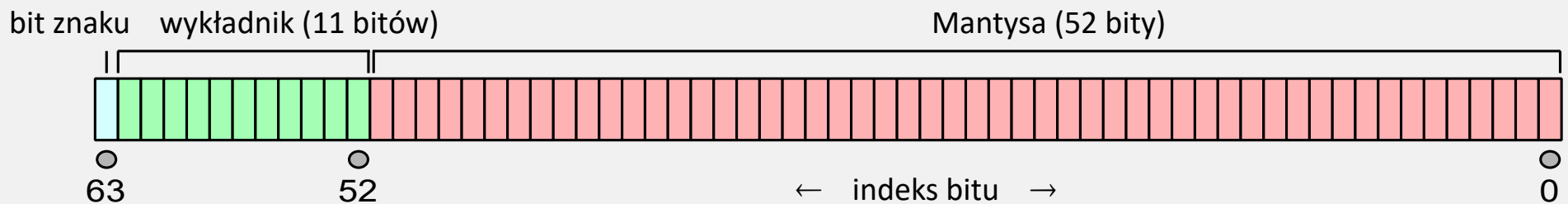
► Notacja dla typu 32-bitowego (**float**) :



Nb. sposób rozkodowania w „typowym przypadku”:

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$$

► Notacja dla typu 64-bitowego (**double**) :



► W każdym przypadku zdefiniowane są: **inf** ($+\infty$), **-inf** ($-\infty$) i **NaN** (not a number).

Np. zwykle: $1./0. = \text{inf}$ $-1./0. = \text{-inf}$ $0./0. = \text{NaN}$

- Możliwości typów zmiennoprzecinkowych:

Typ	Rozmiar [bajty]		Zakres (model LP64)		Precyzja [cyfry]
	Standard C	Model danych LP64	min. x	max. x	
float	'single prec.'	4	$1.4 \cdot 10^{-45}$	$3.4 \cdot 10^{38}$	6
double	'double prec'	8	$4.9 \cdot 10^{-324}$	$1.8 \cdot 10^{308}$	15
long double	'extended prec'	16	$6.5 \cdot 10^{-4966}$	$1.2 \cdot 10^{4932}$	18

(**Precision**: gwarantowana liczba cyfr znaczących, które można wstawić do zmiennej i odzyskać bezstratnie)

Nb.: parametry typów zmiennoprzecinkowych wpisane są w bibliotekę `<cmath>`.

- **Konwersje między typami zmiennej (type cast)**

[\[Link\]](#)

Konwersje **niejawne (implicit)** :

gdy komputer sam skonwertuje typ zmiennej.

Np.: obliczając wyrażenia algebraiczne,
podając argumentom funkcji wartości
czy zwracając z funkcji.

Konwersje **jawne (explicit)**: gdy my polecimy.

- ▶ w stylu C : `(typ) zmienna`
`typ (zmienna)`
- ▶ w stylu C++ : `static_cast<typ> (zmienna)`

(nie wyczerpuje listy, ale obecnie wystarczy)

```
1 #include <iostream>
2 using namespace std;
3
4 int PoorPi () {
5     return 3.141592;
6 }
7
8 int main () {
9     int i = 1.2 * 3 ;
10    cout << i << '\t' << PoorPi() << endl;
11
12    cout << 1/2 << '\t'
13         << (float) 1 / 2 << '\t'
14         << float (1) / 2 << '\t'
15         << static_cast<float> (1) / 2
16         << endl;
17 }
```

3	3		
0	0.5	0.5	0.5

- **Dedukcja typu: auto i decltype**

Słowo **auto** w deklaracji:
kompilator będzie dedukował rzeczywisty typ
deklarowanej zmiennej.

Uwaga do auto w nagłówkach funkcji:

- ▶ auto w typie zwracanym - od standardu C++14
- ▶ auto w argum. wejścia - od standardu C++20

Słowo **decltype (a) b** w deklaracji:
zadeklaruj zmienną b o takim typie,
jaki ma zmienna a.

Uwaga: na zajęciach będziemy unikać dedukcji,
aby się dobrze nauczyć typów.

```
i d d
i d
```

```
1 #include <iostream>
2 #include <typeinfo>
3 using namespace std;
4
5 auto myfun (auto x) {
6     return x;
7 }
8
9 int main () {
10     auto ix = 5;
11     auto dy = 2 + 0.5;
12     decltype ( dy ) dz = -123.45;
13     cout << typeid ( ix ).name() <<
14         ' ' << typeid ( dy ).name() <<
15         ' ' << typeid ( dz ).name() << endl;
16     cout << typeid ( myfun(ix) ).name() <<
17         ' ' << typeid ( myfun(dy) ).name() ;
18 }
```

- Jak kompilować kod w ramach danego standardu C++ :
\$ g++ -std=c++20 mycode.C -o mycode.exe

- Operatory (in/de)krementacji

```
++a  preinkrementacja
--b  predekrementacja
```

} gdy zmiana ma zajść natychmiast.

Np. dla `int i = 5; cout << (++i) * 5 ;` → 30

```
a++  postinkrementacja
a--  postdekrementacja
```

} gdy zmiana ma zajść już po obliczeniu wyrażenia.

Np. dla `int i = 5; cout << (i++) * 5` → 25
`<< ' ' << i;` → 6

- Operator = (direct assignment)

lvalue → `A = 5;` ← *rvalue*

- to obiekt mający adres.
 Tu operator = wpisze *rvalue*.

- to obiekt tymczasowy,
 który operator = wpisze do *lvalue*.

- Operatory przypisania (compound assignment)

```
a += x;   ↔   tożsame a = a + x;
a -= x;
a *= x;
a /= x;
a %= x;
```

...

Ale również bitowo: (omówimy wkrótce)

```
a &= x;    a <<= x;
a |= x;    a >>= x;
a ^= x;
```

- **Operator `,`** ← w wyrażeniu `a, b`; wykonywane są działania `a` i `b`, ale zwracane tylko `b`.

`int a = 1, b = 2, c = 3;` ← tu `,` służy tylko za separator.

`int i = (a += 2 , a + b) ;` ← tu zadziała operator `,`
Skutek: `a = 3, b = 2, c = 3, i = 5`

`for (a = 1, b = 5 ; a<=b ; a++ , b--)` ← tu operator `,` wykona po 2 działania w polach `for`
`cout << a << ':' << b << ' ' ;`
Wypis na ekranie: `1:5 2:4 3:3`

☞ **Komentarz:** szereg operatorów pozwala na „trickowe” zapisanie działań, ale im więcej „trików”, tym gorsza czytelność. Czytelność kodu przez człowieka jest bardzo ważna.

- **Operator `? :` (ternary/conditional)** ← służy za „jednolinijkowy if”

```
int x = 3;  
double result = (x % 2 == 0) ? x*x : -x ;
```

Testowane
wyrażenie

gdy prawda,
zwróć to

w przeciwnym
razie – to.

Operatory bitowe (bitwise operators)

a & b bitowe and
a | b bitowe or
a ^ b bitowe exor

~a bitowa negacja
a << n przesunięcie a o n bitów w lewo
a >> n przesunięcie a o n bitów w prawo

[\[Link\]](#)

```
a = 10101010
b = 00001111

AND a & b = 00001010
OR  a | b = 10101111
XOR a ^ b = 10100101
NOT ~ b = 11110000

b << 5 = 11100000
b >> 2 = 00000011
```



```
1 #include <iostream>
2 #include <bitset>
3 using namespace std;
4
5 int main () {
6     char a = 0b10101010 ,
7           b = 0b00001111 ;
8     cout << "a = " << bitset<8> (a) << endl;
9     cout << "b = " << bitset<8> (b) << "\n\n";
10
11     cout << "AND a & b = " << bitset<8> (a & b) << endl;
12     cout << "OR  a | b = " << bitset<8> (a | b) << endl;
13     cout << "XOR a ^ b = " << bitset<8> (a ^ b) << endl;
14     cout << "NOT  ~ b = " << bitset<8> ( ~ b) << "\n\n";
15     cout << "b << 5 = " << bitset<8> (b << 5) << endl;
16     cout << "b >> 2 = " << bitset<8> (b >> 2) << "\n\n";
17 }
```

- **Tabela priorytetów operatorów** dostępna [[tutaj](#)] .

- ① To reguły, gdy pierwszeństwa nie wymuszają (...)
- ② Im wyżej, tym wyższy priorytet
- ③ W przypadku zbitki operatorów o tym samym priorytecie, kierunek działań definiuje **Associativity**.

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right →
2	a++ a--	Suffix/postfix increment and decrement	Right-to-left ←
	type() type{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
3	. ->	Member access	Right-to-left ←
	++a --a	Prefix increment and decrement	
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of ^[note 1]	
	co_await	await-expression (C++20)	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right →
5	a*b a/b a%b	Multiplication, division, and remainder	Left-to-right →
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	

8	<<=>	Three-way comparison operator (since C++20)	Right-to-left ←
9	< <= > >=	For relational operators < and ≤ and > and ≥ respectively	
10	== !=	For equality operators = and ≠ respectively	
11	a&b	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13		Bitwise OR (inclusive or)	
14	&&	Logical AND	
15		Logical OR	
16	a?b:c	Ternary conditional ^[note 2]	
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
17	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= =	Compound assignment by bitwise AND, XOR, and OR	
	,	Comma	Left-to-right →



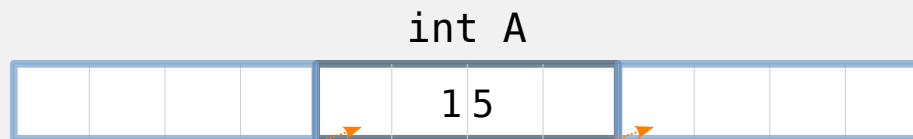
Komentarz

To kolejny "materiał na tricki".
Kodujmy tak, byśmy byli zrozumiani.
Np. stosujemy () w wyrażeniach.

- **Wskaźnik (pointer)**: to zmienna do przechowywania adresu w pamięci zmiennej danego typu.

Na wstępie poznajmy **operator adresu**, `&`.

```
int A = 15;
cout << &A           → 0x7fff1467aed4
cout << ( &A ) + 1   → 0x7fff1467aed8
```



Operator nie tylko zwraca adres, ale i typ. Widać, że `+1` przesuwa o 4 bajty, czyli o długość jednego `int`'a.

Uwaga: znak `&` używany jest też przy deklaracji referencji. Ale tam – należy do *lvalue*, a tu – do *rvalue*.

- Deklarujemy wskaźnik i wstawmy tam adres zmiennej A:

```
int* Aptr = &A ;
```

Wypiszmy zawartość wskaźnika:

```
cout << Aptr ; → 0x7fff1467aed4
```

- Poznajmy **operator wyłuskania (dereference)**, `*`.
Przekształca on adres zmiennej – w tą zmienną.

```
cout << *Aptr ; → 15
```

```
*Aptr = 17;
cout << A ; → 17
```

- Sam wskaźnik też jest zmienną i ma swój adres:

```
cout << &Aptr ; → 0x7ffc2a21d830
```

- Wskaźnik można tylko zadeklarować, a wypełnić później:
Można go też „przełączyć” na inną zmienną:

```
int* NewPtr ; NewPtr = &A ;
int B ;       NewPtr = &B ;
```

- **Wskaźnik (pointer)** . Ważna możliwość: komunikacja między funkcjami przez argumenty wejścia.

[Link]

Gdy wołana funkcja przyjmuje wskaźnik, a w miejscu wywołania podamy adres zmiennej, to **funkcja pracuje na „oryginalnej” zmiennej** (tej z miejsca wywołania).

Chcąc zmienić wartość zmiennej ze wskaźnika, używamy operatora wyłuskania, `*`.

Osiągamy efekt, jak przy referencjach.

Uwaga:

w podfunkcji próba przepięcia wskaźnika na inny adres, nie będzie miała skutku po powrocie do miejsca wywołania.

Powód: wskaźnik też jest zmienną
⇒ jego życie ograniczone jest do { }
Można ominąć problem, robiąc przekaz przez referencję (lub wskaźnik) na nasz wskaźnik.

```
10
12
```

```
1 #include<iostream>
2 using namespace std;
3
4 void Add_pass_via_Copy (int X ) {
5     X += 2;
6 }
7
8 void Add_pass_via_Pointer (int* Xptr) {
9     *Xptr += 2;
10 }
11
12 int main () {
13     int x = 10 ;
14     Add_pass_via_Copy ( x );
15     cout << x << endl;
16
17     x = 10 ;
18     Add_pass_via_Pointer ( &x );
19     cout << x << endl;
20 }
```

- **Wskaźnik funkcyjny**: obiekt do przechowywania funkcji. Musi znać typy, które funkcja przyjmuje i zwraca.

Deklaracja wskaźnika funkcyjnego (*lvalue*) z przypisaniem (*rvalue*):

```
Typ (*NazwaWskaznika) (Typ, Typ, ...) = NazwaFunkcji;
```

Użycie gdzieś tego wskaźnika
– to wpisanie jego nazwy. Np.:

```
Myfun( NazwaWskaznika );
```

Nazwy funkcji – są zarazem
wskaźnikami funkcyjnymi na siebie. Np:

```
Myfun( sqrt );
```

Funkcja może przyjmować
wskaźnik funkcyjny. W ten sposób
funkcja będzie zdolna do pracy
na dowolnej funkcji (zadanych typów),
którą się jej poda w miejscu wywołania.

```
0.5  
15  
0.5
```



```
1 #include <iostream>  
2 #include <cmath>  
3 using namespace std;  
4  
5 double dzialanie ( double (* MyFunPtr) (double)  
6     , double x) {  
7     return MyFunPtr ( x ) ;  
8 }  
9  
10 int main () {  
11     cout << dzialanie ( sin , 30.*M_PI/180.) << endl;  
12     cout << dzialanie ( sqrt , 225. ) << endl;  
13  
14     double (*tryg) (double) = cos ;  
15  
16     cout << dzialanie (tryg, 60 * M_PI/180.) << endl;  
17 }
```

[\[Link\]](#)