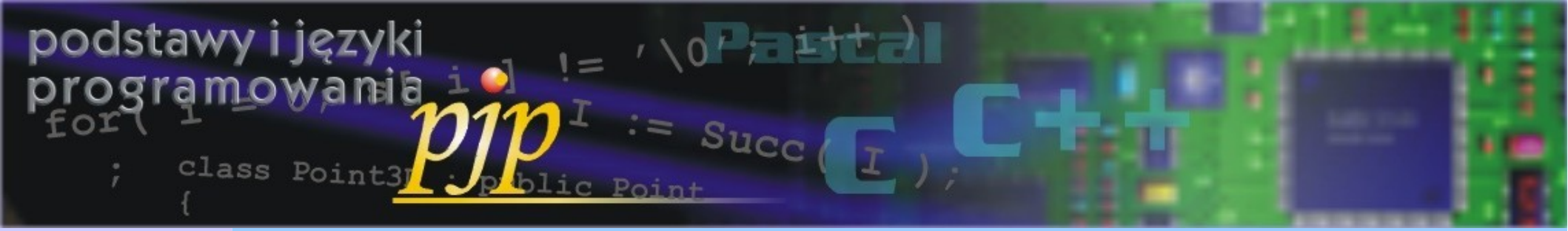


Podstawy programowania

Część szósta

Typy proste

Operatory i wyrażenia

Autor

Roman Simiński

Kontakt

roman.siminski@us.edu.pl

www.programowanie.siminskionline.pl

Syntaktyka — z czego zbudowany jest program?

- W trakcie procesu kompilacji kod źródłowy dzielony jest na elementy zwane *jednostkami leksykalnymi* (ang. *tokens*).
- Rozróżnia się sześć klas jednostek leksykalnych:
  - identyfikatory (ang. *identifiers*),
  - słowa kluczowe (ang. *keywords*),
  - stałe (ang. *constants*),
  - napisy (ang. *string-literals*),
  - operatory (ang. *operators*),
  - separatory (ang. *punctuators*).

Identyfikatory

- Identyfikator to ciąg liter, cyfr i znaków podkreślenia rozpoczynający się od litery, przy czym znak podkreślenia traktowany jest jako litera. Duże i małe litery są rozróżniane.
- Rozróżnia się 31, 32 pierwsze znaki, liczbę znaczących znaków można zwykle w poszczególnych implementacjach redukować.
- Uwaga – polskie znaki nie są traktowane jako litery!

Poprawne identyfikatory:

```
J23, J_23, Pi, wartosc_maksymalna, WartoscMaksymalna, _2Pi, _maksDl, _MAKSDL
```

Niepoprawne identyfikatory:

```
J 23, 007_James_Bond, 2Pi, wartosc maksymalna, Wartosc-Maksymalna,
wartość_maksymalna
```

Identyfikatory, cd. ...

Uwaga – identyfikatory są arbitralnie wybranymi nazwami dla zmiennych, funkcji, definiowanych przez programistę typów danych itp. Nie mogą być jednak *słowami kluczowymi*.

- Nie ma normatywnych zaleceń odnośnie konwencji pisania identyfikatorów. Istnieją tylko umowne konwencje, tak jak np. *notacja węgierska* dla programów pisanych na poziomie WinAPI.
- Tradycyjnie jednak, w programach pisanych w języku C nazwy zmiennych i funkcji pisze się małymi literami, czasem ze znakiem podkreślania w identyfikatorach będących zlepkami.
- W języku C++ znacznie częściej wykorzystuje się małe i duże litery.

```
charcounter, getline, maxline lub char_counter, get_line, max_line
```

C

```
charCounter, getLine, maxLine lub CharCounter, GetLine, MaxLine
```

C++

Słowa kluczowe — dziedzictwo języka C

Słowa kluczowe to identyfikatory zastrzeżone i nie mogą być inaczej stosowane niż określa to standard języka.

Słowa kluczowe winny być pisane tak jak je podano, a więc wyłącznie z wykorzystaniem małych liter. Słowa kluczowe wg. normy ANSI C89:

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

Słowa kluczowe — nowe słowa kluczowe w C++

Do języka C++ zostały przeniesione słowa kluczowe istniejące w języku C, dodano nowe:

asm	dynamic_cast	namespace	reinterpret_cast	try
bool	explicit	new	static_cast	typeid
catch	false	operator	template	typename
class	friend	private	this	using
const_cast	inline	public	throw	virtual
delete	mutable	protected	true	wchar_t

Zestaw słów kluczowych może być rozszerzany w zależności od kompilatora i środowiska programistycznego.

Wybrane separatory

- Nawiasy kwadratowe (ang. *brackets*) [ ] wykorzystywane są do deklarowania i odwoływania się do jedno i wielowymiarowych tablic.
- Nawiasy okrągłe (ang. *parentheses*) ( ) wykorzystywane są do grupowania wyrażeń, izolowania wyrażeń warunkowych, wskazują wywołanie funkcji i jej parametry.
- Nawiasy klamrowe (ang. *braces*) { } oznaczają początek i koniec instrukcji złożonej, zwanej również blokiem.
- Przecinek (ang. *comma*) , rozdziela zwykle elementy na liście parametrów funkcji, występuje również w wyrażeniach przecinkowych.
- Średnik (ang. *semicolon*) ; jest znakiem kończącym instrukcję. Każde legalne wyrażenie w języku C (również wyrażenie puste) zakończone znakiem średnika jest interpretowane jako instrukcja wyrażeniowa.

Komentarze

- Komentarze to fragmenty tekstu spełniające funkcje dowolnych objaśnień robionych przez programistów dla programistów.
- Nie mogą występować w napisach i stałych znakowych. Komentarze są usuwane z tekstu źródłowego programu.

```
/*  To jest komentarz jednoliniowy */

/*
Ten komentarz obejmuje
kilka linii
kodu
*/
```

Standard ANSI C nie dopuszcza komentarzy zagnieżdżonych, choć niektóre kompilatory na to zezwalają.

```
/*
  Ten komentarz obejmuje niedozwolony w ANSI C
  /*  komentarz zagnieżdżony */
  powodujący błąd syntaktyczny
*/
```

Komentarze, cd. ...

W języku C++ można używać komentarzy takich jak w C oraz komentarzy jednoliniowych, rozpoczynających się od pary // i rozciągających się aż do końca linii.

```
int licznik; // Ta zmienna będzie licznikiem wystąpień wzorca
```

Głupie komentarze, które w niczym nie pomagają:

```
int counter;  /* Ta zmienna będzie licznikiem */
float paliwo; /* Zmienna rzeczywista o nazwie paliwo */
int i, j;     /* Zmienne indeksowe tablicy*/
```

Te są wyraźnie lepsze:

```
int counter;  /* Licznik tych linii pliku, które zawierają wzorzec */
float paliwo; /* Zmienna przechowuje ilość paliwa zużytego przez pojazd */
float delta;  /* Wyróżnik trójmianu kwadratowego */
```

Typy proste w języku C++ dzielimy następująco:

- Typy arytmetyczne:
  - całkowite (podstawowe):
    - *char* – znakowy,
    - *int* – całkowity,
    - *bool* — logiczny
    - wyliczeniowe;
  - zmiennopozycyjne (podstawowe):
    - *float* – pojedyncza precyzja,
    - *double* – podwójna precyzja.
- Typ *void*.
- Typy wskaźnikowe.

Typ znakowy char

- Zmienne zadeklarowane jako znakowe – *char* – są dostatecznie duże aby pomieścić dowolny element zbioru znaków dla danej maszyny bądź systemu operacyjnego.
- Wartość zmiennej znakowej to liczba całkowita równa kodowi danego znaku.
- Zmienna typu *char* jest zatem *krótką liczbą całkowitą* i tak może być traktowana, można zmiennych tego typu używać w wyrażeniach.

Typ *char* przeznaczony jest do reprezentowania informacji o *znakach* wykorzystywanych do komunikacji człowiek—komputer.

Ponieważ informacja *o znaku* jest reprezentowana *przez liczbę*, określającą jego *kod*, zmienna *char* ma zatem charakter *numeryczny* i *może* być wykorzystywana do *obliczeń*, jednak *nie* jest to jej *właściwe przeznaczenie*.

- Operacje arytmetyczne na danych typu *char* zwykle nie są szybsze niż te same operacje wykonywane na typach całkowitoliczbowych.

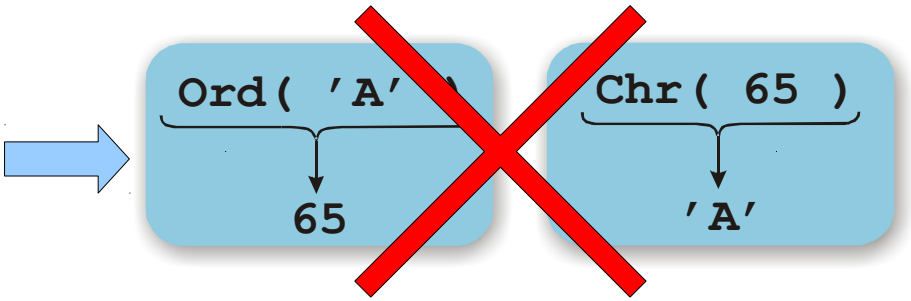
Typ znakowy char

- To jakie liczby odpowiadają poszczególnym znakom określa *system kodowania znaków* obowiązujący na danej maszynie lub w danym systemie operacyjnym.
- Języki C i C++ wywodzą się z USA, tam od wielu lat powszechnie wykorzystywanym systemem kodowania jest ASCII.

```
char c = 'A'; // Literał 'A' ma wartość kodu litery A, w ASCII to 65
char d;

d = c + 1;

cout << c; // Wyprowadza literę A
cout << d; // Wyprowadza literę B
```

```
AB
```

Pascal' owe kombinacje z typem **Char** nie są w C/C++ potrzebne!



Typ znakowy char, literały znakowe wg ASCII

- Do reprezentowania kodów znaków w języku C i C++ służą *literały znakowe*.
- Dzięki nim *nie trzeba* posługiwać się liczbami określającymi kody znaków.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

'A'

'&'

'k'

Źródło: http://www.asciitable.com

Typ znakowy char, literały znakowe

- Literał znakowy jest ciągiem złożonym z jednego lub więcej znaków, zawartych w apostrofach.
- Wartością literału znakowego, zawierającego tylko jeden znak jest numeryczna wartość tego znaku, w zbiorze znaków maszyny wykonującej program.
- Wartość literału wieloznakowego jest zależna od implementacji.
- W języku C literał znakowy reprezentowany jest jako wartość typu *całkowito-liczbowego int*.
- W języku C++ literał znakowy reprezentowany jest przez wartość typu *char*, *literał wieloznakowa* natomiast przez wartość typu *int*.

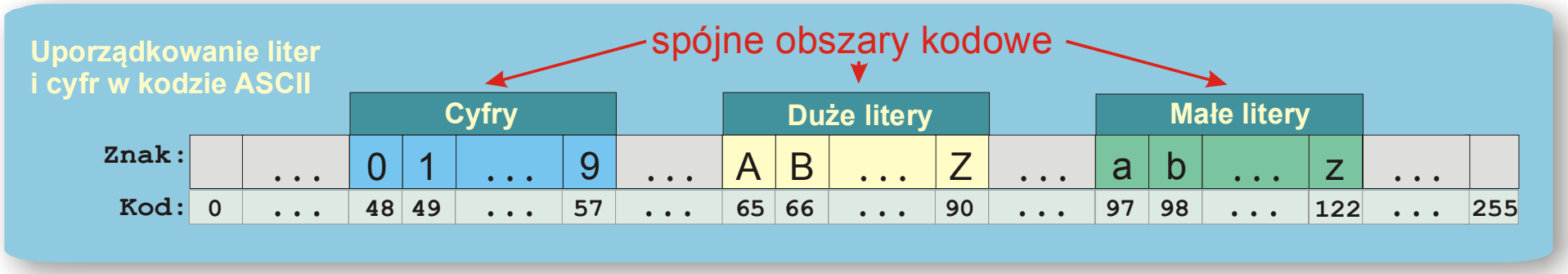
Specjalne literały znakowe

- Sekwencje specjalne pozwalają na reprezentowanie znaków nie posiadających swoich legalnych symboli graficznych.
- Dodatkowo sekwencje specjalne są wykorzystywane do zapisu pewnych „niewygodnych” stałych znakowych.

Sekwencja	Wartość	Znak	Znaczenie
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Apostrof
\"	0x22	"	Cudzysłów
\?	0x3F	?	Pytajnik
\O		any	O = łańcuch ósemkowych cyfr
\xH		any	H = łańcuch szesnastkowych cyfr
\XH		any	H = łańcuch szesnastkowych cyfr

Typ znakowy char, cd...

Zwyczajowo dane typu **`char`** reprezentowany jest na jednym bajcie i służą do reprezentowania znaków kodowanych wg. **ASCII**. Do przechowywania kodów znaków wg. kodowania międzynarodowego wykorzystuje się typ **`wchar_t`**.



Dla kodu ASCII przesunięcie pomiędzy dużymi a małymi literami wynosi 32.
Zamiana zmiennej znakowej *c* z litery dużej na małą (i odwrotnie):

```
if( c >= 'A' && c <= 'Z' )
  c = c + 32;   // lepiej: c += 32
```

```
if( c >= 'a' && c <= 'z' )
  c = c - 32;  // lepiej: c -= 32
```

Specjalne literały znakowe, zastosowania znaku \b

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
  float cena;

  cout << "\nPodaj cene netto: ____ PLN" << "\b\b\b\b\b\b\b\b";
  cin >> cena;
  cout << "Cena brutto: " << cena * 1.22 << " PLN";

  cout << "\n\nEnter=Koniec";
  cin.ignore();
  cin.get();

  return EXIT_SUCCESS;
}
```

```
Podaj cene netto: ____ PLN
```

```
Podaj cene netto: 100_ PLN
```

```
Podaj cene netto: 100_ PLN
Cena brutto: 122 PLN

Enter=Koniec
```

Uwaga – znak *backspace* \b czasem powoduje wymazanie znaku w trakcie cofania kursora.

Rozszerzone zbiory znaków

Rozszerzone zbiory znaków nie mogą być odwzorowywane przez typ *char*. Standard ANSI wprowadza typ całkowity *wchar_t*, jest to typ całkowity zdefiniowany w pliku nagłówkowym *stddef.h*.

Stałe rozszerzonego zbioru znaków zapisuje się z prefiksem *L*, np.:

```
x = L'A';  //  Przypisanie do x literału znakowego reprezentującego literę A
```

- W języku C++ wprowadzono *uniwersalne nazwy znaków*, taka nazwa zaczyna się od \u lub \U i zawiera cyfry szesnastkowe określające kod znaku wg ISO 10646.

- Uwaga – to czy właściwy znak się pojawi, zależy nie tylko od języka, ale od jego bibliotek i tego, czy środowisko systemowe obsługuje dany zestaw kodowania.

Więcej o Unicode i wykorzystaniu w C/C++:

- `http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF`
- `http://www.cl.cam.ac.uk/~mgk25/unicode.html#c`
- `http://pl.wikibooks.org/wiki/C/Napisy`

Typ całkowitoliczbowy int

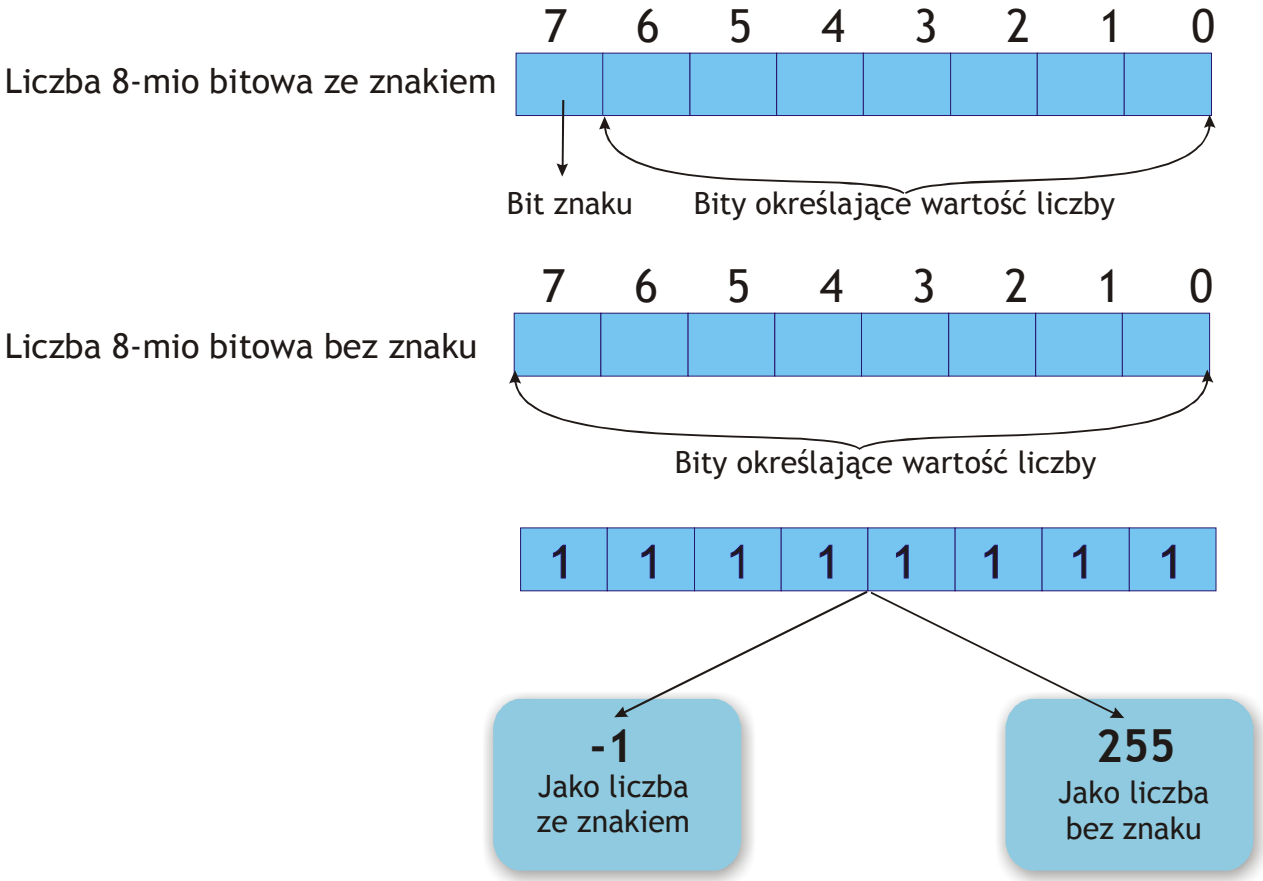
- Zmienne typu całkowitego – *int* – mają zwykle naturalny rozmiar wynikający z architektury maszyny lub środowiska systemowego.
- Zwykle w środowiskach 16-bitowych rozmiar danej typu *int* to dwa bajty, w środowiskach 32-bitowych to 4 bajty.
- Domyślnie typ *int* reprezentuje liczbę ze znakiem (wartości dodatnie i ujemne).

Rozmiar i zakres typu *int* zmienia się, wraz ze zmianą architektury sprzętowej, oprogramowania systemowego i kompilatorów. Standardy zakładają, że typ *int* będzie reprezentowany *minimalnie* na 16-tu bitach (z uwzględnieniem bitu znaku), co odpowiada zakresowi –32768 do 32767.

Typy pochodne typów całkowitych — modyfikatory unsigned i signed

- Modyfikatory *signed* i *unsigned* mogą być stosowane do typów *char* i *int*. Zmieniają one sposób traktowania najstarszego bitu liczby.
- Modyfikatory pozwalają na tworzenie specyfikacji typów pochodnych:
  - *unsigned int* – typ całkowity służący do reprezentacji liczb całkowitych bez znaku. Najstarszy bit liczby jest uznawany za jeden z bitów wartości.
  - *signed int* - typ całkowity służący do reprezentacji liczb całkowitych ze znakiem. Najstarszy bit liczby jest bitem przechowującym informację o znaku liczby, nie wchodzi do bitów wartości.
  - *unsigned char* i *signed char* analogicznie jak dla typu *int*.

Signed, unsigned — o co chodzi?



Ten sam układ bitów, różna interpretacja:

```
signed   char sc = 255;
unsigned char uc = 255;

cout << ( int ) sc << ( int ) uc;
```

```
-1 255
```

Typy pochodne typów całkowitych — modyfikatory short i long a typ int

- Modyfikator *short* sygnalizuje chęć *skrócenia danej w stosunku do rozmiaru typu int*.
- Modyfikator *long* sygnalizuje chęć posłużenia się daną *dłuższą w stosunku do rozmiaru typu int*.
- Modyfikatory short i long mogą być stosowane do typu *int*:
  - *short int* – typ całkowity służący do reprezentowania liczb o potencjalnie „krótszej” reprezentacji wewnętrznej niż typ int, zatem potencjalnie o mniejszym zakresie wartości.
  - *long int* – to typ całkowity służący do reprezentowania liczb o potencjalnie „dłuższej” reprezentacji wewnętrznej niż typ int, zatem potencjalnie o większym zakresie wartości.
  - *long long int* — to typ wprowadzony w C99, służy do reprezentowania bardzo dużych liczb całkowitych (ma być reprezentowany na 64 bitach).

Jak jest naprawdę z tymi długościami różnych typów całkowitych?

- Standard ANSI zakłada, że int oraz short int są co najmniej 16-to bitowe, long int jest co najmniej 32-bitowy.
- Modyfikatory *short* i *long* wprowadzono po to, by umożliwić posługiwanie się różnymi zakresami liczb całkowitych tam, gdzie programiście może się to przydać.
- Dodatkowo mówi się, że:

```
sizeof( char ) <= sizeof( short int ) <= sizeof( int ) <= sizeof( long int )
```

- *unsigned char* traktowany jest jak odpowiednik typu *byte* (Pascal, języki symboliczne),
- *unsigned short* int traktowany jest odpowiednik typu *word* (Pascal, języki symboliczne).

Co warto pamiętać odnośnie typów całkowitych

- Każdy kompilator powinien posiadać dokumentację określającą szczegółowy zakres poszczególnych typów.
- Czasem warto skompilować i uruchomić program, wykorzystujący stałe zdefiniowane w pliku nagłówkowym *limits.h* i *float.h* — definiują one ograniczenia zakresów liczb, przykład:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main()
{
  printf( "                         char: %d..%d\n",      CHAR_MIN, CHAR_MAX );
  printf( "                  short int: %hd..%hd\n",     SHRT_MIN, SHRT_MAX );
  printf( "                        int: %d..%d\n",       INT_MIN, INT_MAX );
  printf( "                   long int: %ld..%ld\n",     LONG_MIN, LONG_MAX );
  printf( "(C99)          long  long int: %lld..%lld\n", LONG_LONG_MIN, LONG_LONG_MAX );
  printf( "                unsigned char: 0..%u\n",      UCHAR_MAX );
  printf( "           unsigned short int: 0..%hu\n",     USHRT_MAX );
  printf( "                 unsigned int: 0..%u\n",      UINT_MAX );
  printf( "            unsigned long int: 0..%lu\n",     ULONG_MAX );
  printf( "(C99) unsigned long long int: 0..%llu\n",    ULONG_LONG_MAX );
  return EXIT_SUCCESS;
}
```

```
                         char: -128..127
                  short int: -32768..32767
                        int: -2147483648..2147483647
                   long int: -2147483648..2147483647
(C99)         long  long int: -9223372036854775808..9223372036854775807
                unsigned char: 0..255
           unsigned short int: 0..65535
                 unsigned int: 0..4294967295
            unsigned long int: 0..4294967295
(C99) unsigned long long int: 0..18446744073709551615
```

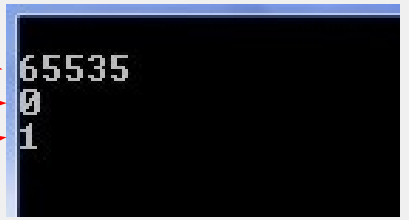
Warto znać przybliżone zakresy

Wybierając typ np. dla zmiennej trzeba oszacować jej typowy, minimalny i maksymalny zakres wartości. Źle dobrane zakresy grożą postaniem *przepełnienia* zmiennych całkowitoliczbowych.

- Typ *char* to ok. 128 na plus i minus, *unsigned char* to 255 na plus.
- Typ *short int* to ok. 32 tyś. na plus i minus, *unsigned short int* to ok. 65 tyś. na plus.
- Typ *int* (16 bitów) jak *short int*.
- Typ *int* (32 bity) to ok. 2 *miliardy* na plus i minus, *unsigned int* to ok. 4 *miliardy* na plus (miliard to rząd wielkości odpowiadający komputerowemu *giga*).
- Typ *long* jak *int* (32 bity).
- Typ *long long int* (64 bity) to ok. 9 *trylionów* na plus i minus, *unsigned long long* to ok. 18 *trylionów* na plus (*trylion* to rząd wielkości odpowiadający komputerowemu e*ksabajtowi*, skrót EB).

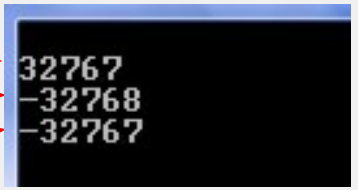
Przekroczenie zakresu dla liczb unsigned – przepełnienie

```
#include <iostream>
#include <climits>
using namespace std;

int main()
{
  unsigned short int ui = USHRT_MAX; // Maks. wartosc

  cout << endl << ui;

  ui++;
  cout << endl << ui;

  ui++;
  cout << endl << ui;

  return EXIT_SUCCESS;
}
```



Przekroczenie zakresu dla liczb signed – przepełnienie

```
#include <iostream>
#include <climits>
using namespace std;

int main()
{
  signed short int si = SHRT_MAX; // Maks. wartosc

  cout << endl << si;

  si++;
  cout << endl << si;

  si++;
  cout << endl << si;


  return EXIT_SUCCESS;
}
```



Czym grozi nieznajomość zakresów wartości?

Załóżmy, że programista to sknera, oszczędzający każdy bajt pamięci...

```
char counter = 0;          // Zmienna char jako krótka liczba całkowita
do
{
  // Jakieś iterowane instrukcje...
  counter++;
}
while( counter < 150 );
```


```
short int counter = 0;          // Teraz krótka zmienna int
do
{
  // Jakieś iterowane instrukcje...
  counter++;
}
while( counter < 50000 );
```

Czym grozi nieznajomość zakresów wartości?

Czasem pozornie niewinne obliczenia mogą być ryzykowne:

```
short int a, b = 20000;
a = 3 * b;

a == ?
```


```
char a, b = 50;
a = 3 * b;

a == ?
```

Nawet dodawanie może być czasem niebezpieczne:

```
char a, b, c;
a = b = 70;

c = a + b;

c == ?
```

Programista może definiować własne synonimy typów

```
typedef unsigned char        byte;
typedef unsigned short int   word;
typedef unsigned long int    counter_t;
```

Specyfikacja *typedef* przypisuje symboliczną nazwę <*identyfikator*> do istniejącej wcześniej definicji typu <*definicja_typu*>.

```
typedef <definicja typu> <identyfikator>;
```

Modyfikatory *short* i *long* a typy zmiennopozycyjne:

- Można stosować modyfikatory *short* i *long* z typami *float* i *double*, jednak tylko kombinacja *long double* ma sens.
- Typ *double* naturalnie rozszerza typ *float* zatem zapis *long float* to po prostu przestarzały synonim typu *double*.
- Z kolei typu *double* nie można skrócić, zatem specyfikacja *short double* nie ma sensu. Nie można również skrócić typu *float*, zatem specyfikacja *short float* nie ma sensu.

Typy danych a systemy 64-bitowe

- W architekturze 64-bitowej typ *int* powinien teoretycznie być 64-bitowy.
- Wyróżnia się różne modele odwzorowania standardowych typów C/C++ w fizyczną reprezentację:

Typ danych	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
pointer	64	64	64	32	32

- "I" to skrót reprezentujący typ *int*, "L" reprezentujący typ *long*, "P" reprezentujący typ *pointer* — *wskaźnikowy*, 64 and 32 odpowiadają rozmiarowi danych liczonemu w bitach.
- Aktualnie preferuje się wykorzystanie modelu LP64.

Krótka dygresja — typy danych C a rozwój sprzętu

TABLE 1 Common C DataTypes

int	long	ptr	long long	Label	Examples
16	--	16	--	IP16	PDP-11 Unix (early, 1973)
16	32	16	--	IP16L32	PDP-11 Unix (later, 1977); multiple instructions for long
16	32	32	--	I16LP32	Early MC68000 (1982); Apple Macintosh 68K Microsoft operating systems (plus extras for X86 segments)
32	32	32	--	ILP32	IBM 370; VAX Unix; many workstations
32	32	32	64	ILP32LL or ILP32LL64	Amdahl; Convex; 1990s Unix systems Like IP16L32, for same reason; multiple instructions for long long.
32	32	64	64	LLP64 or IL32LLP64 or P64	Microsoft Win64
32	64	64	*(64)	LP64 or I32LP64	Most 64/32 Unix systems
64	64	64	*(64)	ILP64	HAL; logical analog of ILP32

*In these cases, LP64 and ILP64 offer 64-bit integers, and long long seems redundant, but in practice, most proponents of LP64 and ILP64 included long long as well, for reasons given later. ILP64 uniquely required a new 32-bit type, usually called _int32.

- Źródło: John R. Mashey, The Long Road to 64 Bits, dostępne online http://queue.acm.org/detail.cfm?id=1165766, październik 2006.

Literały całkowitoliczbowe

- Literał całkowity może być zapisywana *dziesiętnie*, *ósemkowo*, *szesnastkowo*.
- Wszystkie literały rozpoczynające się *od zera* traktowane są jako ósemkowe.
- Wszystkie literały rozpoczynające się od przedrostka `0x` lub `0X` są traktowane jako szesnastkowe.

```
int i = 10;    // Stała dziesiętna
int o = 077;   // Stała ósemkowa
int h = 0xff;  // Stała szesnastkowa
```

Dozwolone cyfry ósemkowe to:

0, 1, 2, 3, 4, 5, 6, 7

Dozwolone cyfry szesnastkowe to:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Literały całkowitoliczbowe, cd. ...

- Literał całkowitoliczbowy może być zakończona przyrostkiem *u* lub *U* co oznacza że liczba jest bez znaku (dokładniej – najstarszy bit liczby jest częścią wartości).
- Literał całkowitoliczbowy może być zakończona przyrostkiem *l* lub *L* co oznacza że liczba jest długa.
- Wartość literału całkowitoliczbowego nie może przekraczać zakresu typu liczba całkowita długa bez znaku (unsigned long int). Wartości większe są obcinane.
- Dla implementacji zakładającej 32-bitową długość liczby długiej bez znaku, wartość maksymalna wynosi odpowiednio:

4 294 967 295 dec 037777777777 oct 0xFFFFFFFF hex

Typy wyliczeniowe

- Typy wyliczeniowe nie występowały we wczesnych implementacjach języka C.
- W tych implementacjach brakowało sposobu przedstawienia uporządkowanej listy takich elementów, które można przedstawić jedynie nazwami. Przykładem mogą być np. dni tygodnia, miesiące, kolory.
- Typ wyliczeniowy to tak na prawdę, lista nazwanych stałych całkowitych.

```
enum RGB_colors
{
  RED,
  GREEN,
  BLUE
};

enum boolean
{
  false,
  true
};
```

Typy wyliczeniowe, cd. ...

- Stałe wyliczeniowe, są typu *int*, mogą wystąpić w każdym miejscu dozwolonym dla danej całkowitej.
- Identyfikatory stałych wyliczeniowych powinny być unikatowe w ramach danego wyliczenia.
- Każda stała wyliczeniowa ma swoją wartość całkowitą. Pierwsza stała na liście otrzymuje wartość 0, następna 1 itd.
- Każda stała występująca w wyliczeniu może posiadać swój *inicjalizator*, przypisujący mu wartość (również ujemną) wyznaczoną przez programistę.
- Każdy element wyliczenia nie posiadający inicjalizatora otrzymuje wartość o jeden większą od swojego poprzednika na liście

```
enum months
{
  JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEPT, OCT, NOV, DEC
};
```

Stałe wyliczeniowe a typ int

Deklarowanie zmiennych wyliczeniowych w języku C/C++ spotyka się sporadycznie, można tak:

```
months m = MAY;
. . .
```

Ale zwyczajowo pisze się tak:

```
int m = MAY;
. . .
```

Przykład wykorzystania typu wyliczeniowego

Przypomnijmy sobie przykład ilustrujący wykorzystanie instrukcji *switch*:

```
. . .
int nadwozie;

cout << "\nJaki typ nadwozia lubisz?";
cout << "\n1. Sedan" << "\n2. SUV" << "\n3. Coupe";
cout << "\nWpisz 1, 2 lub 3: ";
cin >> nadwozie;

switch( nadwozie )
{
  case 1 : cout << "\nChyba lubisz eleganckie limuzyny!";
           break;
  case 2 : cout << "\nWidze, ze ciagnie Cie w teren!";
           break;
  case 3 : cout << "\nTy to pewnie lubisz szybka jazde!";
           break;
}
. . .
```

Trzeba pamiętać, jaki numer przypisaliśmy każdemu typowi nadwozia

Przykład wykorzystania typu wyliczeniowego

```
enum TYP_NADWOZIA
{
  SEDAN = 1,
  SUV,
  COUPE
};
```

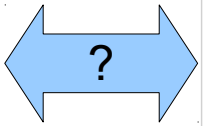
Numery przypisane typom nadwozia są nazwane identyfikatorami wyliczeniowymi

```
. . .
int nadwozie;

cout << "\nJaki typ nadwozia lubisz?";
cout << "\n1. Sedan" << "\n2. SUV" << "\n3. Coupe";
cout << "\nWpisz 1, 2 lub 3: ";
cin >> nadwozie;

switch( nadwozie )
{
  case SEDAN : cout << "\nChyba lubisz eleganckie limuzyny!";
               break;
  case SUV   : cout << "\nWidze, ze ciagnie Cie w teren!";
               break;
  case COUPE : cout << "\nTy to pewnie lubisz szybka jazde!";
               break;
}
. . .
```

Czytelny i samokomentujący się kod programu

Typy wyliczeniowe, alternatywa dla stałych symbolicznych

```
enum ctrl_key_codes
{
  UP    =  0x48,
  DOWN  =  0x50,
  LEFT  =  0x4b,
  RIGHT =  0x4d,
  DEL   =  0x53,
  INS   =  0x52,
  HOME  =  0x47,
  END   =  0x4f,
  PGUP  =  0x49,
  PGDN  =  0x51
};
```




```
#define UP    0x48
#define DOWN  0x50
#define LEFT  0x4b
#define RIGHT 0x4d
#define DEL   0x53
#define INS   0x52
#define HOME  0x47
#define END   0x4f
#define PGUP  0x49
#define PGDN  0x51
```

Typy wyliczeniowe, wartości uporządkowane

```
enum ctrl_key_codes
{
  UP    =  0x48,
  DOWN  =  0x50,
  LEFT  =  0x4b,
  RIGHT =  0x4d,
  DEL   =  0x53,
  INS   =  0x52,
  HOME  =  0x47,
  END   =  0x4f,
  PGUP  =  0x49,
  PGDN  =  0x51
};
```



```
enum ctrl_key_codes
{
  HOME  =  0x47,
  UP,
  PGUP,
  LEFT  =  0x4b,
  RIGHT =  0x4d,
  END   =  0x4f,
  DOWN,
  PGDN,
  INS,
  DEL
};
```

Typ zmiennopozycyjne

- Standard nie określa wewnętrznej reprezentacji danych zmiennopozycyjnych, zwykle implementacje są zgodne z formatem IEEE dotyczącym takich liczb.
- *float* to typ przeznaczony do reprezentowania liczb rzeczywistych pojedynczej precyzji.
- *double* to typ przeznaczony jest do reprezentowania liczb rzeczywistych w podwójnej precyzji.

Literały zmiennopozycyjne

Stałe zmiennopozycyjne składają się z:

- części całkowitej (ciąg cyfr),
- kropki dziesiętnej,
- części ułamkowej (ciąg cyfr),
- litery *e* lub *E* oraz opcjonalnego wykładnika potęgi ze znakiem,
- opcjonalnego przyrostka *f* lub *F* lub *l* lub *L*.

Literały zmiennopozycyjne, cd. ...

- Można pominąć część całkowitą lub część ułamkową (lecz nie obie jednocześnie).
- Ogólnie mówiąc, notacja stałych zmiennopozycyjnych odpowiada regułom naukowego zapisu liczb w postaci zwykłej i wykładniczej.
- W przypadku *braku przyrostków* stałe zmiennopozycyjne są *typu double*.
- Dodając przyrostek *f* lub *F* można wymusić aby stała była typu *float*, podobnie, dodając przyrostek *l* lub *L* wymusza się aby stała była typu *long double*.

Zapis	Znaczenie
23.45e6	$23.45 \cdot 10^{6}$
.0	0
0.	0
1.	1
-1.23	-1.23
2e-5	$2.0 \cdot 10^{-5}$
3E+10	$3.0 \cdot 10^{10}$
.09E34	$0.09 \cdot 10^{34}$

Typ void

- Wystąpienie typu *void* (*próżny*, *pusty*) w deklaracji oznacza *brak wartości*.
- W zależności od kontekstu interpretacja zapisu *void* może się nieznacznie zmieniać, zawsze jednak jest to sygnał, że w danym miejscu nie przewiduje się wystąpienia żadnej *konkretnej wartości* lub *konkretnego typu*.

Funkcja bezparametrowa:

```
int func( void ) { ... }
```

Funkcja nie udostępniająca rezultatu:

```
void fun( int i ) { ... }
```

Bezparametrowa funkcja, nie udostępniająca rezultatu:

```
void fun( void ) { ... }
```

Rzutowanie rezultatu funkcji na typ **`void`** — rezultat funkcji jest nieistotny

```
( void )sin( 0 );   // Mało sensowne ale to przykład
```

Operatory arytmetyczne

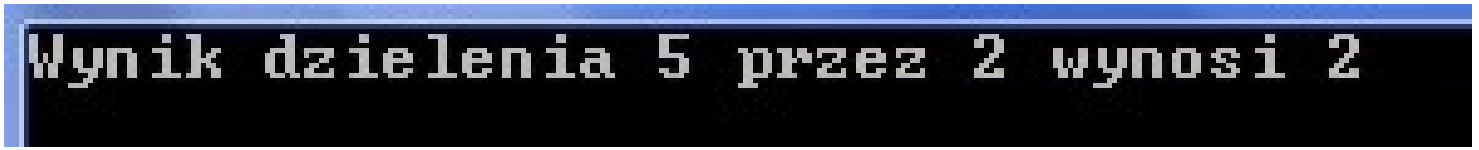
- Operatory arytmetyczne to
  - `+` : dodawanie (dwuargumentowy) i zachowanie znaku (jednoargumentowy),
  - `-` : odejmowanie (dwuargumentowy) i zmiana znaku (jednoargumentowy),
  - `*` : mnożenie (dwuargumentowy),
  - `/` : dzielenie (dwuargumentowy),
  - `%` : reszta z dzielenia *modulo*, tylko liczby całkowite (dwuargumentowy).

Uwaga — dzielenie na operandach całkowitych daje wynik całkowity

Ten kod:

```
int x = 5, y = 2;

float f = x / y;

cout << ”Wynik dzielenia ” < < x << ”przez ” << y << ” wynosi ” <<  f ;
```

wyprodukuje:



Aby otrzymać wynik rzeczywisty jeden z operandów musi być rzeczywisty:

```
float x = 5;
int   y = 2;

float f = x / y;
```

lub

```
int x = 5, y = 2;

float f = ( float ) x / y;
```

Rzutowanie (konwersja) typów w wyrażeniu

Reszta z dzielenia — operator modulo

```
int x = 5, y = 2;
float f;

f = x / y;
cout << "\nWynik dzielenia " << x << " przez " << y << " wynosi " <<  f ;

f = x % y;
cout << "\nReszta dzielenia " << x << " przez " << y << " wynosi " <<  f ;
```

```
Wynik dzielenia 5 przez 2 wynosi 2
Reszta dzielenia 5 przez 2 wynosi 1_
```

Przelicz czas wyrażony w sekundach na godziny, minuty, sekundy:

```
int czas_w_sek, zostalo;
int godziny, minuty, sekundy;

cout << "Podaj czas w sekundach: ";
cin >> czas_w_sek;

godziny = czas_w_sek / 3600;
zostalo = czas_w_sek % 3600;
minuty  = zostalo / 60;
sekundy = zostalo % 60;
. . .
```

```
Podaj czas w sekundach: 7322

Czas w sekundach: 7322
02:02:02
```

Reszta z dzielenia — operator modulo

```
int liczba_linii = 0;

do
{
  liczba_linii++;
  cout << "Wyprowadzam linie nr: " << liczba_linii << endl;

  if( liczba_linii % 24 == 0 )
  {
    cout << "Nacisnij Enter by kontynuowac...";
    if( cin.get() == 'q' ) // "Tajne" wyjscie po nacisnieciu 'q'
      break;
  }
}
while( liczba_linii < 1000 );
```

Zatrzymaj co 24 linie.

Przy okazji — przerwij gdy użytkownik nacisnął klawisz *q* (i zatwierdził to klawiszem Enter)

Operatory relacji i operatory logiczne

- Operatory relacji to : >, >=, <, <=.
- Operatory porównania ==, !=.
- Operatory logiczne to && (and) oraz || (or).

Uwaga:

- Wyrażenia połączone tymi operatorami oblicza się od *lewej do prawej*, koniec obliczeń następuje *natychmiast po określeniu wartości* logicznej wyrażenia.
- Operatory relacji mają wyższy priorytet niż operatory porównania.
- Priorytet operatora && jest wyższy niż || a oba są niższe niż operatorów relacji i porównania. Dlatego poniższy warunek może być zapisany bez nawiasów:

```
if( c >= 'A' && c <= 'Z' )
{
  . . .
}
```

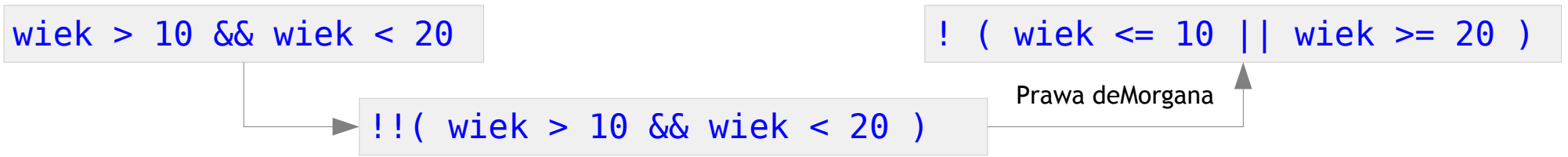
zamiast

```
if( ( c >= 'A' ) && ( c <= 'Z' ) )
{
  . . .
}
```

Operatory logiczne i relacyjne – przykłady, komplementarność

```
int wiek;

cout << "Tylko dla nastolatków\nPodaj wiek: ";
cin >> wiek;

if( wiek > 10 && wiek < 20 )
  cout << "Dostep potwierdzony";
else
  cout << "Brak dostepu!";
```

```
int wiek;

cout << "Tylko dla nastolatków\nPodaj wiek: ";
cin >> wiek;

if( wiek <= 10 || wiek >= 20 )
  cout << "Brak dostepu!";
else
  cout << "Dostep potwierdzony";
```



Operatory logiczne i relacyjne, przykład wykorzystania

Czy rok w zmiennej całkowitej *year* jest przestępny?

```
if( ( year % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 )
  cout << ”Rok ” << year << ” jest rokiem przestępnym”;
else
  cout << ”Rok ” << year << ” nie jest rokiem przestępnym”;
```

Czy zmienna znakowa *c* jest cyfrą lub małą literą?

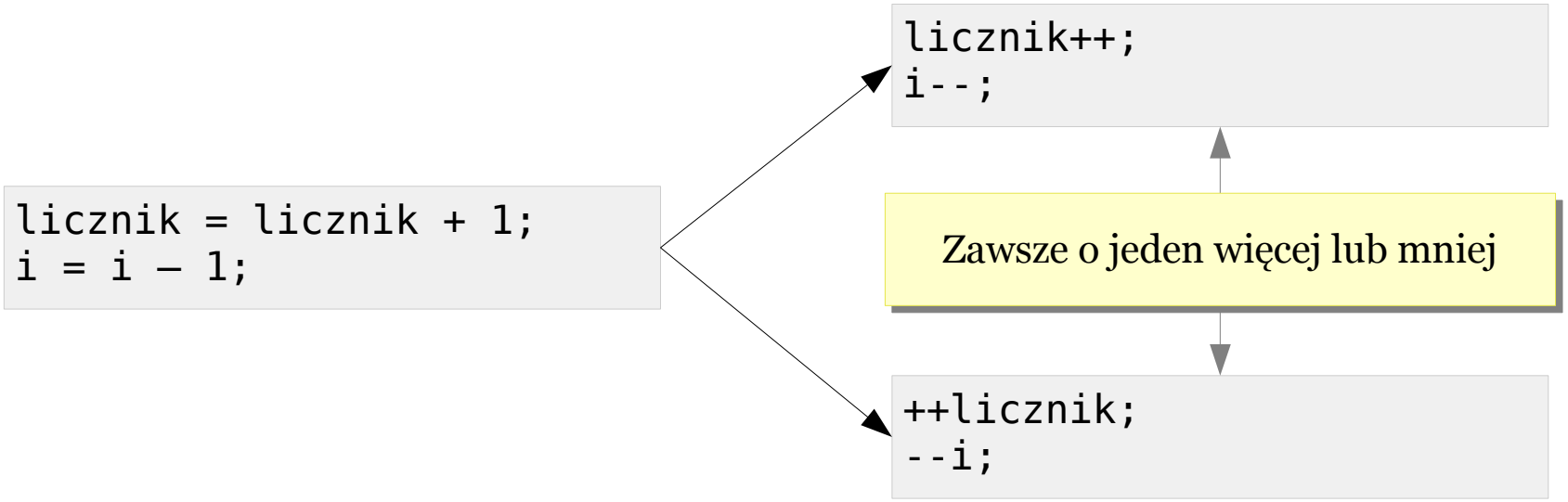
```
if( ( c >= '0' && c <= '9' ) || ( c >= 'a' && c <= 'z' ) )
  cout << ”Cyfra lub mała litera”;
else
  cout << ”To nie jest ani cyfra, ani mała litera”;
```

Czy zmienna znakowa *c* jest cyfrą szesnastkową?

```
if( ( c>='0' && c<='9' ) || ( c>='a' && c<='f' ) || ( c>='A' && c<='F' ) )
  cout << ”Cyfra szesnastkowa”;
else
  cout << ”To nie jest cyfra szesnastkowa”;
```

Operatory zwiększania ++ i zmniejszania ––

- Operatory ++ i –– zawsze powodują odpowiednio *zwiększenie* lub *zmniejszenie* wartości argumentu.
- Mogą one jednak występować jako *przedrostki* (ang. *prefix*) lub *przyrostki* (ang. *postfix*).
- To, czy występują jako przedrostek czy przyrostek ma wpływ na *otoczenie* — zwykle jest nim *wyrażenie* zawierające argument z operatorem ++ lub ––.
- Operatory ++ i –– pozwalają kompilatorowi na wygenerowanie efektywnego kody wynikowego.



Operatory zwiększania ++ i zmniejszania ––

- Wersja *przedrostkowa zwiększa* (zmniejsza) wartość argumentu *przed użyciem jego wartości* w wyrażeniu.
- Wersja *przyrostkowa zwiększa* (zmniejsza) wartość argumentu *po użyciem jego wartości* w wyrażeniu.

```
int a = 5, b;

b = ++a;
```

```
a == 6, b == 6
```

```
int a = 5, b;

b = a++;
```

```
a == 6, b == 5
```

```
int licznik = 0;

cout << "Licznik: " << licznik;
licznik++;
...
licznik++;
cout << "Licznik: " << licznik;
```

```
int licznik = 0;

cout << "Licznik: " << licznik++;

...

cout << "Licznik: " << ++licznik;
```

Operatory zwiększania ++ i zmniejszania ——

Wykorzystując operatory ++ i —— iterację:

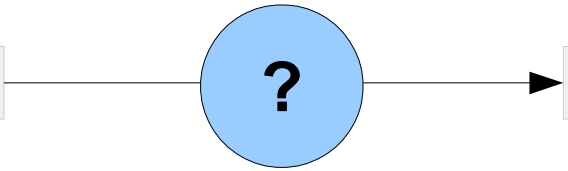
```
for( int licznik = 10; licznik > 0; licznik-- )
  cout << endl << licznik << "...";
```

Można zapisać krócej:

```
for( int licznik = 10; licznik > 0; cout << endl << licznik-- << "..." )
  ;
```

Na operatory ++ i —— trzeba uważać:

```
i = ++i---i+++i+i--;
```




```
i = ++i - --i + ++i + i--;
```

Operatory bitowe — wprowadzenie

Operatory bitowe mogą być stosowane do argumentów typu całkowitego, są to:

Operator	Znaczenie
`&`	bitowa koniunkcja (*and*),
`|`	bitowa alternatywa (*or*),
`^`	bitowa różnica symetryczna (*xor*),
`<<`	przesunięcie w lewo,
`>>`	przesunięcie w prawo,
`~`	dopełnienie jedynkowe.

- Operatory bitowe służą zwykle do realizacji operacji niskiego poziomu (wywołanie funkcji systemowych, obsługa przerwań, komunikacja na poziomie sprzętu i podstawowych protokołów sieciowych).
- Operatory bitowe wykorzystuje się również przy optymalizacji wykonywania operacji czasochłonnych bądź pamięciożernych, również przy przetwarzaniu danych multimedialnych.

Operatory bitowe — wykonywane operacje

Operatory bitowe realizują operacje na poszczególnych bitach liczb

```
int x = 5, y = 7, z;
```

Koniunkcja bitowa

```
z = x & y;

  x 0...0101
  y 0...0111
    0...0101 z == 5
```

Alternatywa bitowa

```
z = x | y;

  x 0...0101
  y 0...0111
    0...0111 z == 7
```

Bitowa różnica symetryczna

```
z = x ^ y;

  y 0...0101
  y 0...0111
    0...0010 z == 2
```

Przesunięcie w lewo

```
y = 1;
z = x << y;

  x 0...0101
  y       1
    0..01010 z == 10
```

Przesunięcie w prawo

```
y = 1;
z = x >> y;

  x 0...0101
  y       1
    00..0010 z == 2
```

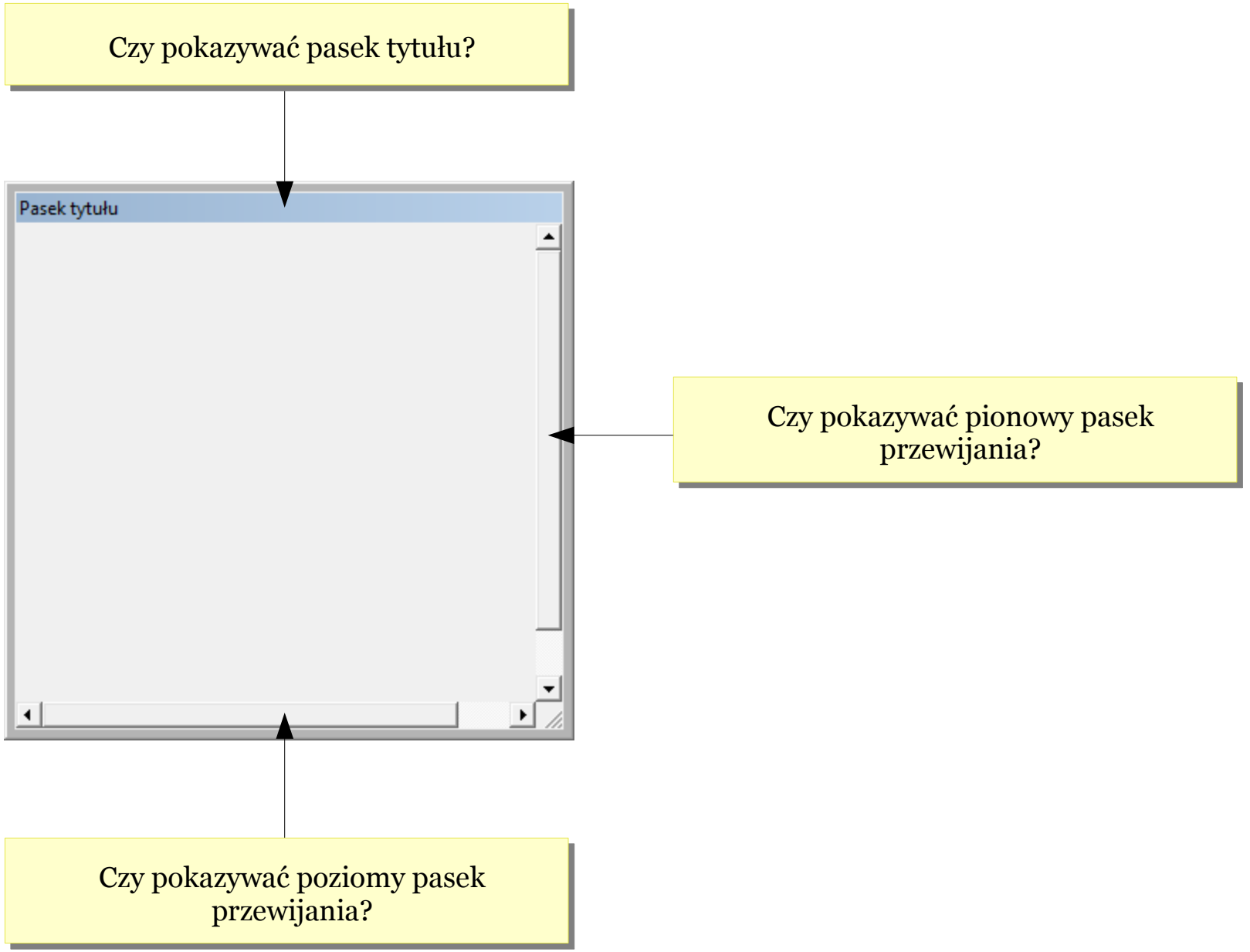
Negacja bitowa (U1)

```
z = ~x;


  x 0...0101
 ~x 1...1010
    1...1010 z == -6
```

Operatory bitowe — przykłady zastosowania

- Często poszczególne bity liczby całkowitej wykorzystuje się jako *znaczniki występowania* lub *nie występowania* pewnej informacji.
- Na pojedynczej liczbie o rozmiarze bajta można wtedy „upakować” *osiem* znaczników.
- Alternatywą jest wykorzystanie osobnych *zmiennych boolowskich*.
- Załóżmy, że dla każdego okna GUI chcemy pamiętać pewien zestaw informacji o wyglądzie — np. czy ma być pokazywany pasek tytułu okna, czy ma być pokazywany poziomy i pionowy pasek przewijania.

Operatory bitowe — przykłady zastosowania



Operatory bitowe — przykłady zastosowania

Informacje o właściwościach okna w postaci osobnych zmiennych logicznych:

```
bool showCaption;
bool showHScrollBar;
bool showVScrollBar;
```

Ustalanie opcji pokazywania okna:

```
showCaption = true;
showHScrollBar = true;
showVScrollBar = true;
```

Sprawdzanie opcji pokazywania okna podczas jego wyświetlania:

```
. . .
displayWindowFrame();
if( showCaption )
  displayWindowCaption();
if( showHScrollBar )
  displayHScrollBar();
if( showVScrollBar )
  displayVScrollBar();
. . .
```

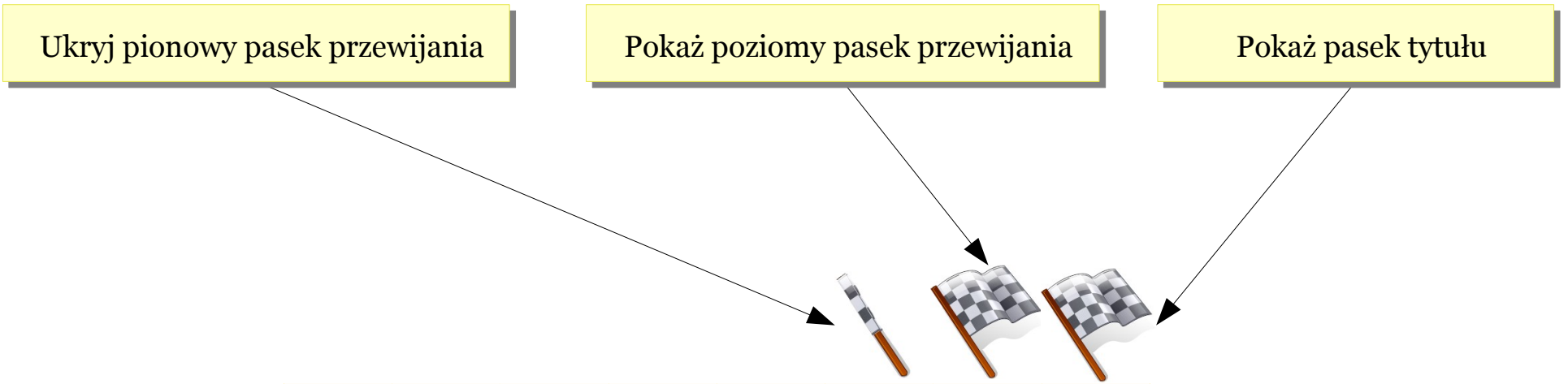
Operatory bitowe — przykłady zastosowania

Informacje o właściwościach okna w postaci jednej zmiennej znacznikowej:

```
unsigned char windowStyle;
```

Wystąpienie 1-ki na określonym bicie oznacza *wystąpienie* określonej opcji okna, wystąpienie 0-ra oznacza *brak* opcji.

Często mówi się o ustawieniu („zapaleniu”) *znacznika* lub *flagi*.



x – dowolna wartość bitu

Operatory bitowe — przykłady zastosowania

Jak ustawiać, czyścić i sprawdzać stan poszczególnych opcji okna?

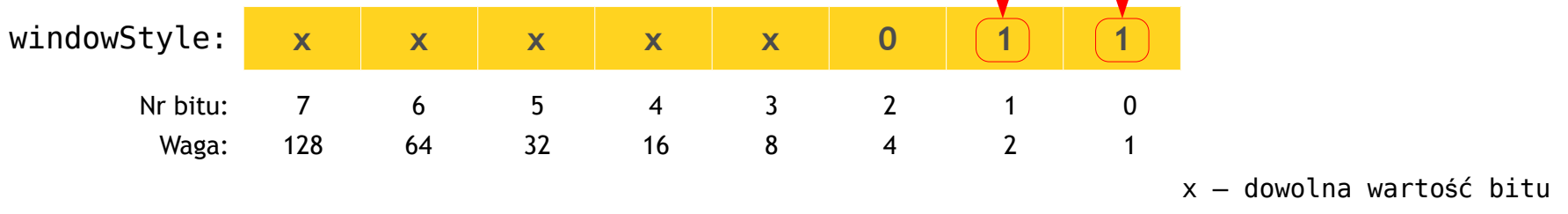
```
enum WINDOW_STYLE
{
  wsShowCaption    = 1; // 0...00000001
  wsShowHScrollBar = 2; // 0...00000010
  wsShowVScrollBar = 4; // 0...00000100
};
```

Wyzerowanie wszystkich znaczników:

```
windowStyle = 0;
```

Ustalenie bitów odpowiedzialnych za tytuł i poziomy pasek przewijania:

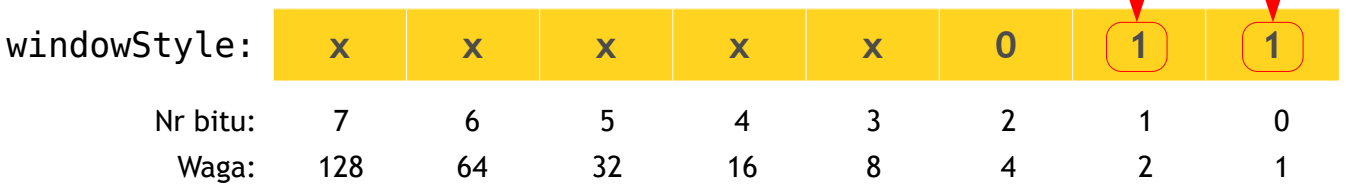
```
windowStyle = windowStyle | wsShowCaption ;
windowStyle = windowStyle | wsShowHScrollBar ;
```



Operatory bitowe — przykłady zastosowania

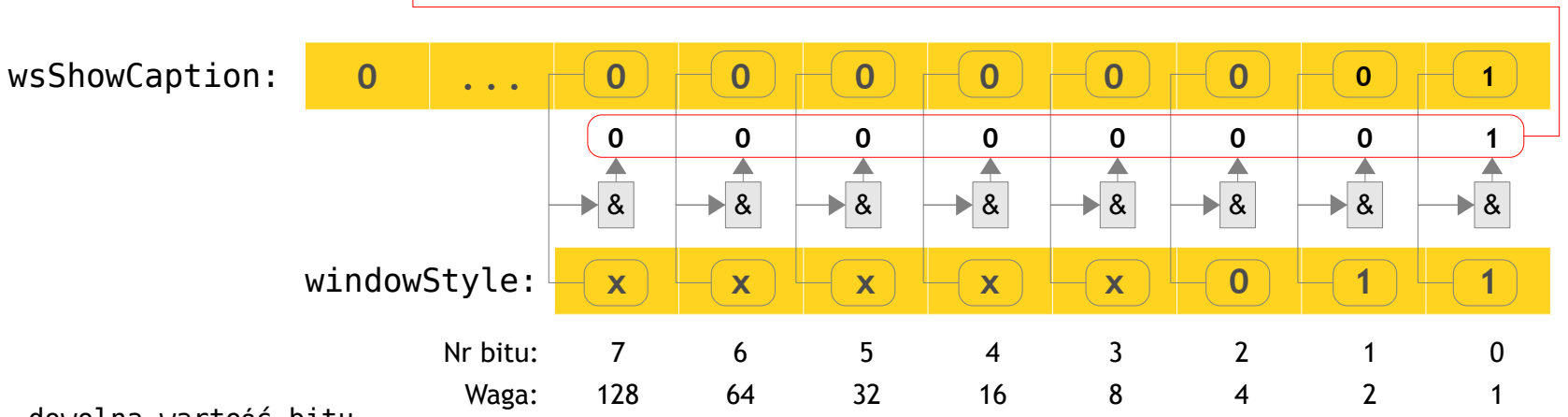
Jednoczesne ustalenie wartości bitów:

```
windowStyle = windowStyle | ( wsShowHScrollBar | wsShowCaption );
```



Testowanie stanu określonego bitu:

```
if( windowStyle & wsShowCaption )
  displayWindowCaption();
```



x – dowolna wartość bitu

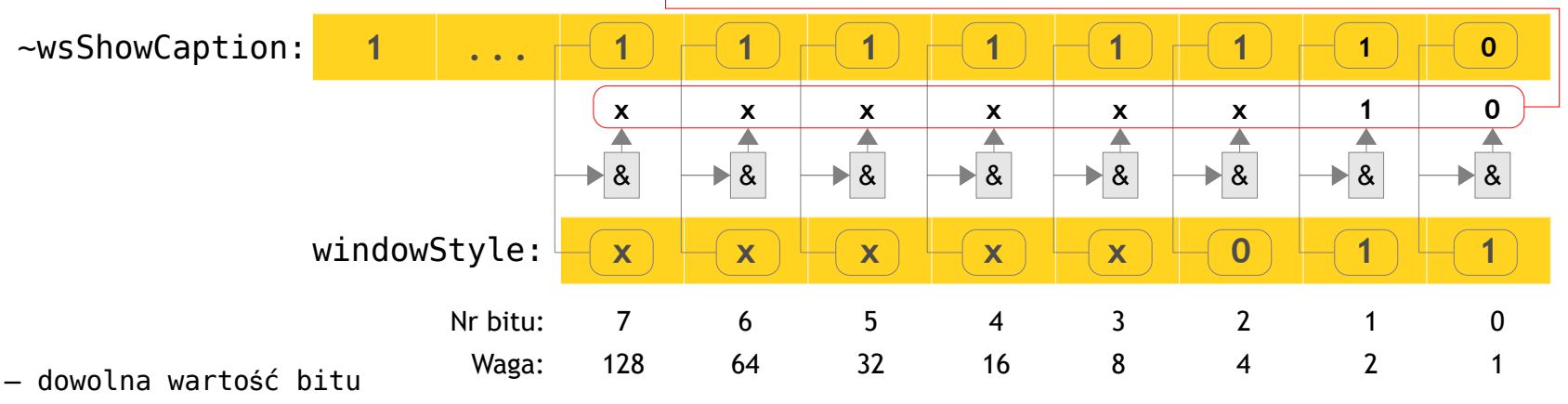
Operatory bitowe — przykłady zastosowania

Sprawdzanie opcji pokazywania okna podczas jego wyświetlania:

```
. . .
displayWindowFrame();
if( windowStyle & wsShowCaption )
  displayWindowCaption();
if( windowStyle & wsShowHScrollBar  )
  displayHScrollBar();
if( windowStyle & wsShowVScrollBar  )
  displayVScrollBar();
. . .
```

Czyszczenie wybranego bitu opcji okna:

```
windowStyle = windowStyle & ~wsShowCaption;
```



x – dowolna wartość bitu

Operatory bitowe — przykłady zastosowania

- Wykorzystywanie poszczególnych bitów liczby całkowitej jako znaczników jest popularnym i powszechnie wykorzystywanym sposobem oszczędnego „upakowywania” informacji.
- Minimalizuje się w ten sposób wykorzystanie pamięci i optymalizuje transfery pamięć operacyjna ↔ procesor.
- Wykorzystując daną 16-bitową (*unsigned short int*) lub 32-bitową (*unsigned int*) otrzymujemy możliwość zapisania sporej liczby informacji.
- Operacje bitowe są szybkie.

Operatory bitowe — przykłady zastosowania

Znaczniki stylów okien na poziomie *WinAPI* (plik nagłówkowy *winuser.h*)

```
/*
 * Window Styles
 */
#define WS_OVERLAPPED       0x00000000L
#define WS_POPUP            0x80000000L
#define WS_CHILD            0x40000000L
#define WS_MINIMIZE         0x20000000L
#define WS_VISIBLE          0x10000000L
#define WS_DISABLED         0x08000000L
#define WS_CLIPSIBLINGS     0x04000000L
#define WS_CLIPCHILDREN     0x02000000L
#define WS_MAXIMIZE         0x01000000L
#define WS_CAPTION          0x00C00000L     /* WS_BORDER | WS_DLGFRAME  */
#define WS_BORDER           0x00800000L
#define WS_DLGFRAME         0x00400000L
#define WS_VSCROLL          0x00200000L
#define WS_HSCROLL          0x00100000L
#define WS_SYSMENU          0x00080000L
#define WS_THICKFRAME       0x00040000L
#define WS_GROUP            0x00020000L
#define WS_TABSTOP          0x00010000L

. . .
```

Operatory bitowe — przesunięcia bitowe

Przesunięcie o jeden bit w lewo:

```
unsigned char flag = 1;       // 00000001
flag = flag << 1;             // 00000010
flag = flag << 1;             // 00000100
```

Przesunięcie o jeden bit w prawo:

```
unsigned char flag = 128;     // 10000000
flag = flag >> 1;             // 01000000
flag = flag >> 1;             // 00100000
```

Uwaga na bit znaku:

```
signed char flag = 128;       // 10000000
flag = flag >> 1;             // 11000000
flag = flag >> 1;             // 11100000
```

Operatory bitowe — przesunięcia bitowe, przykład wykorzystania

Przesunięcia zamiast mnożenia i dzielenia przez krotności 2-ki:

```
int kwota, podwojona, polowa;

cout << "Podaj kwote: ";
cin >> kwota;

podwojona = 2 * kwota;
polowa = kwota / 2;

cout << "Podwojona kwota: " << podwojona << endl;
cout << "   Polowa kwoty: " << polowa << endl;
```

```
Podaj kwote: 100
Podwojona kwota: 200
   Polowa kwoty: 50
```

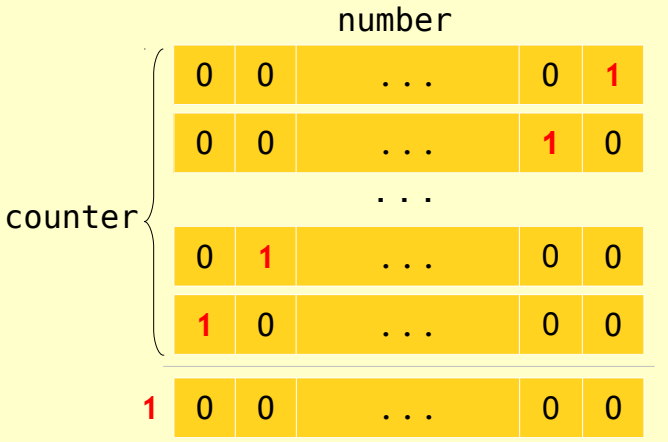
```
int kwota, podwojona, polowa;

cout << "Podaj kwote: ";
cin >> kwota;

podwojona = kwota << 1;
polowa = kwota >> 1;

cout << "Podwojona kwota: " << podwojona << endl;
cout << "   Polowa kwoty: " << polowa << endl;
```

Operatory bitowe — przesunięcia bitowe, przykład wykorzystania

Ile bitów ma liczba typu *int* w danej implementacji (iteracja *while*)?

```
int number = 1, counter = 0;

while( number != 0 )
{
  number = number << 1;
  counter ++;
}

cout << "Liczba bitów typu int: " << counter;
```

Przesuwaj 1-kę w lewo dopóki nie wyleci poza najstarszy bit liczby, licz ile razy udało się przesunąć.



Ile bitów ma liczba typu *int* w danej implementacji (iteracja *for*)?

```
int number = 1, counter = 0;

for( ; number != 0; counter++ )
  number = number << 1;

cout << "Liczba bitów typu int: " << counter;
```

Operatory przypisania

Przypisanie wartości jest w języku C/C++ *wyrażeniem* a nie *instrukcją*. Operator przypisania = jest *lewostronnie* łączny, co umożliwia łączenie przypisań:

```
int i = 5, j, k, l;
j = k = l = i;
```

Zamiast:

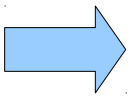
```
l = i + 5;
k = l  + 10;
j = k * 2;
```

można napisać:

```
j = ( k = ( l = ( i + 5 ) ) + 10 ) * 2;    // j == 40, k == 20, l == 10
```

Często:

```
x = sin( alfa );
if( x == 0 )
{
  ...
}
```



```
if( ( x = sin( alfa ) ) == 0 )
{
  ...
}
```

Operatory przypisania a operator porównania

Łatwo się pomylić — przypisanie to = a porównanie to ==. Zamiast:

```
if( delta == 0 )
{
  . . .
}
```

bardzo często pomyłkowo piszę się:

```
if( delta = 0 )
{
  . . .
}
```

Aby do tego nie dopuścić:

```
if( 0 == delta )
{
  . . .
}
```

W przypadku pomyłki, kompilator nie przyjmie konstrukcji: **0 = delta**

Operatory przypisania, cd. ...

Dla większości operatorów dwuargumentowych:

```
+   -   *   /   %   <<   >>   *   ^   &   |
```

można wykorzystywać specjalne operatory przypisania, pozwalające skrócić zapis często wykorzystywanych konstrukcji, takich jak:

Wersja z normalna

```
i = i + 2
y = y * 2
x = x << 1
j = j * ( k + 1 )
flag = flag >> k;
```

Wersja skrócona

```
i += 2
y *= 2
x <<= 1
j *= k + 1
flag >>= k;
```

Operatory przypisania, cd. ...

Ogólnie, jeśli *expr1* i *expr2* to wyrażenia, a *op* to operator dwuargumentowy, zapis:

```
expr1 = expr1 op expr2
```

można uprościć do postaci:

```
expr1 op= expr2
```

```
for( ; number != 0; counter++ )
  number = number << 1;
```

→

```
for( ; number != 0; counter++ )
  number <<= 1;
```

W C++ dopuszcza się użycie *aliasów* dla wybranych operatorów

Operator	Alias
&&	and
&	bitand
&=	and_eq
||	or
|	bitor
|=	or_eq
^	xor
^=	xor_eq
!	not
!=	not_eq
~	compl

- Opisowe nazwy operatorów (aliasy) wprowadzono aby umożliwić edycję programów z wykorzystaniem nietypowych klawatur.
- Opisowych nazw operatorów można używać w języku C++ bez żadnych dodatkowych zabiegów.
- Począwszy od 1995 roku do standardowej biblioteki języka C dodano plik nagłówkowy `iso646.h` definiujący odpowiednia makra, pozwalające na używanie w C tych samych aliasów.

Operator warunkowy

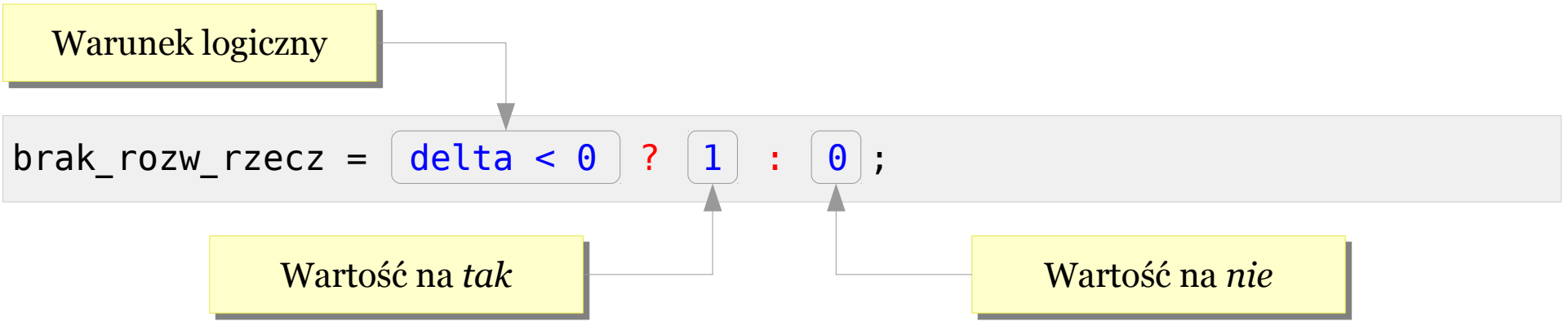
Bardzo często spotyka się „symetryczne” instrukcje warunkowe, np.:

```
if( delta < 0 )
  brak_rozw_rzecz = 1;
else
  brak_rozw_rzecz = 0;
```

Można je zapisać krócej, z wykorzystaniem ***trójargumentowego*** *operatora warunkowego*:

arg1 ? arg2 : arg3

Wartością takiego wyrażenia jest: arg2 gdy arg1 ma wartość niezerową, arg2 gdy arg1 ma wartość równą zero.



Operator warunkowy

Wyznaczanie wartości maksymalnej — instrukcja alternatywy:

```
if( a > b )
  max = a;
else
  max = b;
```

Wyznaczanie wartości maksymalnej — operator warunkowy:

```
max = a > b ? a : b;
```

Czasem bezpieczniej i czytelniej jest zapisywać warunek w nawiasach:

```
max = ( a > b ) ? a : b;
```

Operator warunkowy

Wartość wyrażenia z operatorem warunkowym nie musi być liczbowa:

```
char c;
cin >> c;
...
if( c >= 'A' && c <= 'Z' )
  cout << ”Duza litera”;
else
  cout << ”Nie duza litera”;
```

Zamiast instrukcji warunkowej operator warunkowy:

```
cout << ( c >= 'A' && c <= 'Z' ? ”Duza litera” : ”Nie duza litera” );
```

Uwaga! Bez nawiasów powyższe wyrażenie nie będzie działać prawidłowo.

Wyrażenia z operatorem warunkowym mogą być zagnieżdżane:

Przykład — gdy opóźnienie oddania książki do biblioteki nie przekracza tygodnia, to opłata karna wynosi 5zł, gdy jest większe ale nie przekracza dwóch tygodni oplata wynosi 10zł, powyżej dwóch tygodni opłata karna wynosi 20zł.

```
kara = ( opoznienie > 14 ) ? 20 : ( opoznienie > 7 ) ? 10 : 5;
```

Operator warunkowy

Operator warunkowy pozwala na radykalne skrócenie kodu:

```
. . .
if( ( kara = ( opoznienie > 14 ) ? 20 : ( opoznienie > 7 ) ? 10 : 5 ) > 0 )
  cout << ”Naliczono oplate karna w wysokosci: ” << kara;
. . .
```

Zamiast:

```
. . .
if( opoznienie > 14 )
  kara = 20;
else
  if( opoznienie > 7 )
    kara = 10;
  else
    kara = 5;

if( kara > 0 )
  cout << ”Naliczono oplate karna w wysokosci: ” << kara;
. . .
```

Operator warunkowy

Jeszcze jeden przykład — jeżeli zmienna **po_angielsku** jest niezerowa, program ma posługiwać się komunikatami w języku angielskiem, jeżeli jest zerowa, to komunikaty mają być po polsku.

```
bool po_angielsku;
. . .
po_angielsku = true;
. . .
cout << ( po_angielsku ? ”Input value: ” : ”Podaj wartość: ” );
. . .
po_angielsku = false;
. . .
if( logowanie_nieudane )
  cout << ( po_angielsku ? ”Login incorrect!” : ”Blad logowania!” );
```