

C++

Seweryn Kowalski

III Informatyka Stosowana

Literatura

- B. Stroustrup, Język C++,
- Bruce Eckel, Thinking in C++
- J. Grębosz, Symfonia C++ tom I, II, III,
- J. Grębosz, Pasja, tom I, II,
- Wiele różnych
 - J. Bielecki, ANSI C++,
 - H. M. Deitel, P. J. Deitel, C++ Programowanie
 - Majczak, Praktyczne programowanie w C
 - W. Borowik, B. Borowik, Meandry języka C++

Klasyfikacja języków programowania

- Niskiego poziomu (maszynowe, asemblery), związane z danym typem procesora
- Wysokiego poziomu, niezależne od rodzaju procesora
- Ogólnego przeznaczenia, do tworzenia bardzo różnorodnego oprogramowania
- Specjalizowane, np. do obliczeń inżynierskich i naukowych (Fortran), ekonomicznych (Cobol)
- Języki wysokiego poziomu i ogólnego przeznaczenia to dla przykłady:
 - Pascal/Delphi, C, C++, Java, Python, Perl

Kompilator

Język maszynowy

00000	10011110
00001	11110100
00010	10011110
00011	11010100
00100	10111111
00101	00000000

Suma dwóch liczb

```
1 int a, b, sum;  
2  
3 cin >> a;  
4 cin >> b;  
5  
6 sum = a + b;  
7 cout << sum << endl;
```

To samo w C++

Integrated Development Environment (IDE) dla C++

IDE	Platform	Console programs
Code::blocks	Windows/Linux/MacOS	Compile console programs using Code::blocks
Visual Studio Express	Windows	Compile console programs using VS Express 2013
Dev-C++	Windows	Compile console programs using Dev-C++

Standard C++ 2011

Compiler	Platform	Command
GCC	Linux, among others...	<code>g++ -std=c++0x example.cpp -o example_program</code>
Clang	OS X, among others...	<code>clang++ -std=c++11 -stdlib=libc++ example.cpp -o example_program</code>

Struktura programu

```
1 // my first program in C++
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << "Hello World!";
7 }
```

Linia 1 // rozpoczyna on komentarz, który rozciąga się do końca linii

```
1 // line comment
2 /* block comment */
```

Inne komentarze wieloliniowe to /* i */

Linia 2 **#include <iostream>** - dyrektywa preprocesora, dołącz plik nagłówkowy (ang. header) o nazwie **iostream.h** do tekstu programu i następnie wykonaj kompilację

Linia 3 pusta linia:

Linia 4 deklaracja funkcji głównej o **nazwie main()** – funkcja ta jest specjalna funkcja w C++ (tzw funkcja główna), wykonywanie każdego programu rozpoczyna się od funkcji **main()**, skompilowany program zaczyna się od wykonywania instrukcji zawartych w funkcji głównej

Linia 5 i 7 – otwarcie i zamknięcie bloku { }

Linia 6 – polecenie, zakończone jest znakiem ;

Struktura programu

- Format zapisu jest swobodny co oznacza, że tak zwane białe znaki (ang. white space characters) są pomijane przez kompilator, służą one jedynie zwiększeniu czytelności programu. Białe znaki to:
 - Spacja, tab, line feed (LF) - nowy wiersz, carriage return – powrót karetki, vertical-tab, newline, from feed (FF) – nowa strona

```
1 int main ()  
2 {  
3     std::cout << " Hello World!";  
4 }
```

```
int main () { std::cout << "Hello World!"; }
```

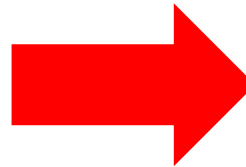
```
1 // my second program in C++  
2 #include <iostream>  
3  
4 int main ()  
5 {  
6     std::cout << "Hello World! ";  
7     std::cout << "I'm a C++ program";  
8 }
```

```
int main () { std::cout << " Hello World! "; std::cout << " I'm a C++ program "; }
```

```
1 int main ()  
2 {  
3     std::cout <<  
4         "Hello World!";  
5     std::cout  
6         << "I'm a C++ program";  
7 }
```

using namespace std

```
1 // my second program in C++
2 #include <iostream>
3
4 int main ()
5 {
6     std::cout << "Hello World! ";
7     std::cout << "I'm a C++ program";
8 }
```



```
1 // my second program in C++
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     cout << "Hello World! ";
8     cout << "I'm a C++ program";
9 }
```


Zmienne i ich typy

```
1 a = 5;  
2 b = 2;  
3 a = a + 1;  
4 result = a - b;
```

- Zmienną można rozumieć jak element pamięci z przypisaną wartością
- Zmienne muszą być odróżnialne jedna od drugiej, czyli muszą posiadać nazwę

Identyfikatory (nazwy)

- Jest to dowolnie długi ciąg liter lub cyfr.
- Pierwszy znak musi być literą (_ - uważa się za literę)
- Rozróżniamy duże i małe liter np.: Litera i litera ("case sensitive,,")
- Następujące identyfikatory są zastrzeżone:

```
alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t,
class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast,
else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable,
namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register,
reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template,
this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void,
volatile, wchar_t, while, xor, xor_eq
```

Identyfikatory (nazwy)

- Identyfikatory zawierające __ nie powinny być stosowane
- operatory i znaki przestankowe
 - ! % ^ & * () - + = { } | ~ [] \ ; ' : " < > ? , . /
- następujące kombinacje znaków to operatory
 - -> ++ -- .* ->* << >> <= >= == != && || *= /= %= += -= <<= >>= &=
^= |= ::
- Preprocesor używa następujących symboli
 - # ##

Podstawowe typy danych

Group	Type names*	Notes on size / precision
Character types	<code>char</code>	Exactly one byte in size. At least 8 bits.
	<code>char16_t</code>	Not smaller than <code>char</code> . At least 16 bits.
	<code>char32_t</code>	Not smaller than <code>char16_t</code> . At least 32 bits.
	<code>wchar_t</code>	Can represent the largest supported character set.
Integer types (signed)	<code>signed char</code>	Same size as <code>char</code> . At least 8 bits.
	<code>signed short int</code>	Not smaller than <code>char</code> . At least 16 bits.
	<code>signed int</code>	Not smaller than <code>short</code> . At least 16 bits.
	<code>signed long int</code>	Not smaller than <code>int</code> . At least 32 bits.
	<code>signed long long int</code>	Not smaller than <code>long</code> . At least 64 bits.
Integer types (unsigned)	<code>unsigned char</code>	(same size as their signed counterparts)
	<code>unsigned short int</code>	
	<code>unsigned int</code>	
	<code>unsigned long int</code>	
	<code>unsigned long long int</code>	
Floating-point types	<code>float</code>	
	<code>double</code>	Precision not less than <code>float</code>
	<code>long double</code>	Precision not less than <code>double</code>
Boolean type	<code>bool</code>	
Void type	<code>void</code>	no storage
Null pointer	<code>decltype(nullptr)</code>	

Deklaracje zmiennych

- Każda nazwa (identyfikator) zanim zostanie użyta w programie C++ musi być najpierw zadeklarowana, tzn. trzeba wyspecyfikować jej typ, żeby poinformować kompilator do jakiego rodzaju elementu odnosi się ta nazwa

```
1 int a;  
2 float mynumber;
```

```
int a, b, c;
```

```
1 // operating with variables
2
3 #include <iostream>
4 using namespace std;
5
6 int main ()
7 {
8     // declaring variables:
9     int a, b;
10    int result;
11
12    // process:
13    a = 5;
14    b = 2;
15    a = a + 1;
16    result = a - b;
17
18    // print out the result:
19    cout << result;
20
21    // terminate the program:
22    return 0;
23 }
```

Inicjalizacja zmiennych

- Zadeklarowana zmienna ma niezdeterminowana wartość, oczywiście można nadać wartość zmiennej podczas jej deklaracji
- Trzy możliwe sposoby inicjalizacji
- Pochodząca z języka C
 - typ zmienna = wartość
- Inicjalizacja jak dla konstruktora
 - typ zmienna (wartość)
- „*uniform initialization*” (C++ standard 2011)
 - typ zmienna {wartość}

```
int x = 0;
```

```
int x (0);
```

```
int x {0};
```

Inicjalizacja zmiennych

```
1 // initialization of variables
2
3 #include <iostream>
4 using namespace std;
5
6 int main ()
7 {
8     int a=5;           // initial value: 5
9     int b(3);          // initial value: 3
10    int c{2};           // initial value: 2
11    int result;         // initial value undetermined
12
13    a = a + b;
14    result = a - c;
15    cout << result;
16
17    return 0;
18 }
```


Automatyczny typ zmiennej (C++ standard 2011)

- Kompilator orientuje się o typie zmiennej podczas jej inicjalizacji

```
1 int foo = 0;  
2 auto bar = foo; // the same as: int bar = foo;
```

```
1 int foo = 0;  
2 decltype(foo) bar; // the same as: int bar;
```

Trudność w czytaniu kodu

Zmienne znakowe (class string)

```
1 // my first string
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string mystring;
9     mystring = "This is a string";
10    cout << mystring;
11    return 0;
12 }
```

Możliwe sposoby deklaracji

```
1 string mystring = "This is a string";
2 string mystring ("This is a string");
3 string mystring {"This is a string"};
```

```
1 // my first string
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string mystring;
9     mystring = "This is the initial string content";
10    cout << mystring << endl;
11    mystring = "This is a different string content";
12    cout << mystring << endl;
13    return 0;
14 }
```

Literały – stałe całkowite

```
1 1776
2 707
3 -273
```

- Przykład
- Trzy rodzaje stałych całkowitych
 - Dziesiętne
 - Ósemkowe – poprzedzone zerem
 - Szesnastkowe – poprzedzone **0x**
- Typy stałych całkowitych

```
1 75           // decimal
2 0113        // octal
3 0x4b        // hexadecimal
```

Suffix	Type modifier
u or U	unsigned
l or L	long
ll or LL	long long

```
1 75           // int
2 75u          // unsigned int
3 75l          // long
4 75ul         // unsigned long
5 75lu         // unsigned long
```

Stałe zmiennopozycyjne

- Domyślnym typem jest **double**

Suffix	Type
<i>f or F</i>	float
<i>l or L</i>	long double

```
1 3.14159L    // long double
2 6.02e23f    // float
```

```
1 3.14159      // 3.14159
2 6.02e23      // 6.02 x 10^23
3 1.6e-19      // 1.6 x 10^-19
4 3.0          // 3.0
```

```
1 'z'  
2 'p'  
3 "Hello world"  
4 "How do you do?"
```

Literały napisowe (strings)

- Literał napisowy jest ciągiem znaków ujętym w cudzysłów
 - Podwójny cudzysłów: **"to jest łańcuch"**
 - Pojedynczy cudzysłów: **'a'**
- Każdy łańcuch zawiera o jedno znak więcej, niż się to wydaje, a jest nim znak **'\0'**, który ma wartość 0
 - **sizeof("abcd")==5**
 - napis **"abcd"** jest typem **char[5]**
 - Napis pusty definiuje się jako **" "** i jest on typu **char[1]**

Literały napisowe (strings)

- Specjalne stałe znakowe

Escape code	Description
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

Inne literały

- Prawda i fałsz – dwie możliwe wartości typu **bool**
- nullptr – to tzw. Zerowa (pusta) wartość wskaźnika

```
1 bool foo = true;  
2 bool bar = false;  
3 int* p = nullptr;
```

Nadanie nazwy stałym

```
1 const double pi = 3.1415926;  
2 const char tab = '\t';
```

```
1 #include <iostream>  
2 using namespace std;  
3  
4 const double pi = 3.14159;  
5 const char newline = '\n';  
6  
7 int main ()  
8 {  
9     double r=5.0;           // radius  
10    double circle;  
11  
12    circle = 2 * pi * r;  
13    cout << circle;  
14    cout << newline;  
15 }
```


Definicje preprocesora

- Stałe można wprowadzać także do preprocesora
- Następuje zastąpienie stałej poprzez jej nazwę
- Zamiana następuje przed kompilacją programu
- Nie następuje sprawdzanie składni
- Jest to tzw. instrukcja pojedynczej linii, brak znaku ; na końcu

#define identyfikator wartość

```
1 #include <iostream>
2 using namespace std;
3
4 #define PI 3.14159
5 #define NEWLINE '\n'
6
7 int main ()
8 {
9     double r=5.0;           // radius
10    double circle;
11
12    circle = 2 * PI * r;
13    cout << circle;
14    cout << NEWLINE;
15
16 }
```

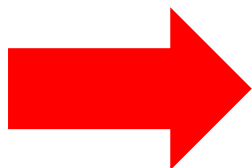
Operatory

`x = 5;`

`x = y;`

- Operator podstawienia (przypisania)
 - Przypisuje wartość zmiennej
 - Przypisanie zawsze działa od prawej do lewej i nigdy odwrotnie

`y = 2 + (x = 5);`



```
1 x = 5;  
2 y = 2 + x;
```

`x = y = z = 5;`

```
1 // assignment operator  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main ()  
6 {  
7     int a, b;           // a:?, b:?  
8     a = 10;             // a:10, b:?  
9     b = 4;              // a:10, b:4  
10    a = b;              // a:4, b:4  
11    b = 7;              // a:4, b:7  
12  
13    cout << "a:";  
14    cout << a;  
15    cout << " b:";  
16    cout << b;  
17 }
```

Operatory arytmetyczne

operator	description
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operatory związane

`+=, -=, *=, /=, %=, >>=, <<=, &=, ^=`

expression	equivalent to...
<code>y += x;</code>	<code>y = y + x;</code>
<code>x -= 5;</code>	<code>x = x - 5;</code>
<code>x /= y;</code>	<code>x = x / y;</code>
<code>price *= units + 1;</code>	<code>price = price * (units+1);</code>

Operatory inkrementacji i dekrementacji

++, --

- Zmiana o jeden
- Są to operatory jedno argumentowe i mogą znajdować się po obu stronach argumentu. Czyli mogą przyjmować formę:
 - przedrostkową (ang. prefix) **++a, --a**
 - końcówkową (ang. postfix) **a++, a--**

Example 1	Example 2
<pre>x = 3; y = ++x; // x contains 4, y contains 4</pre>	<pre>x = 3; y = x++; // x contains 4, y contains 3</pre>

Na ćwiczenia – przykład

```
int a=5, b=5, c=5, d=5;

cout << "oto wartosci" << endl;

cout << "a++" << a++ << endl
<< "++b" << ++b << endl
<< "c--" << c-- << endl
<< "--d" << --d << endl;

cout << "a teraz jakie wartosci maja zmienne"

cout << "a = " << a << endl
<< "b = " << b << endl
<< "c = " << c << endl
<< "d = " << d << endl;
}
```

Operatory relacji

- Wynikiem działania tych operatorów jest prawda lub fałsz
- Bardzo częstym błędem jest postawienie tylko jednego znaku = (czyli operatora przypisania).

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

```
1 (7 == 5)    // evaluates to false
2 (5 > 4)     // evaluates to true
3 (3 != 2)    // evaluates to true
4 (6 >= 6)    // evaluates to true
5 (5 < 5)     // evaluates to false
```

a=1 b=3 c=6

```
1 (a == 5)    // evaluates to false, since a is not equal to 5
2 (a*b >= c)  // evaluates to true, since (2*3 >= 6) is true
3 (b+4 > a*c) // evaluates to false, since (3+4 > 2*6) is false
4 ((b=2) == a) // evaluates to true
```

Operatory logiczne

!, &&, ||

```
1 !(5 == 5)    // evaluates to false because the expression at its right (5 == 5) is true
2 !(6 <= 4)    // evaluates to true because (6 <= 4) would be false
3 !true        // evaluates to false
4 !false       // evaluates to true
```

&& OPERATOR (and)		
a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

OPERATOR (or)		
a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

Operatory logiczne

- *short-circuit evaluation*
- W C++ ewaluacja zawsze następuje od lewej do prawej, stąd np. w linii 2 nigdy nie nastąpi ewaluacja (3>6)

```
1 ( (5 == 5) && (3 > 6) ) // evaluates to false ( true && false )
2 ( (5 == 5) || (3 > 6) ) // evaluates to true ( true || false )
```

operator	short-circuit
&&	if the left-hand side expression is false, the combined result is false (the right-hand side expression is never evaluated).
	if the left-hand side expression is true, the combined result is true (the right-hand side expression is never evaluated).

```
if ( (i<10) && (++i<n) ) { /*...*/ } // note that the condition increments i
```

Powiększenie i tylko gdy wyrażenie po prawej stronie jest prawdą

Operator warunkowy ?

warunek ? rezultat1 : rezultat2

```
1 7==5 ? 4 : 3    // evaluates to 3, since 7 is not equal to 5.
2 7==5+2 ? 4 : 3  // evaluates to 4, since 7 is equal to 5+2.
3 5>3 ? a : b     // evaluates to the value of a, since 5 is greater than 3.
4 a>b ? a : b     // evaluates to whichever is greater, a or b.
```

```
1 // conditional operator
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int a,b,c;
8
9     a=2;
10    b=7;
11    c = (a>b) ? a : b;
12
13    cout << c << '\n';
14 }
```

Operator bitwise

operator	asm equivalent	description
&	AND	Bitwise AND
	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

Operator zmiany typu (rzutowania)

```
1 int i;  
2 float f = 3.14;  
3 i = (int) f;
```

Operator sizeof

- Sprawdza rozmiar typu danych

```
x = sizeof (char);
```

Hierarchia operatorów

From greatest to smallest priority, C++ operators are evaluated in the following order.

Level	Precedence group	Operator	Description	Grouping
1	Scope	::	scope qualifier	Left-to-right
2	Postfix (unary)	++ --	postfix increment / decrement	Left-to-right
		()	functional forms	
		[]	subscript	
		. ->	member access	
3	Prefix (unary)	++ --	prefix increment / decrement	Right-to-left
		~ !	bitwise NOT / logical NOT	
		+ -	unary prefix	
		& *	reference / dereference	
		new delete	allocation / deallocation	
		sizeof	parameter pack	
		(type)	C-style type-casting	
4	Pointer-to-member	.* ->*	access pointer	Left-to-right
5	Arithmetic: scaling	* / %	multiply, divide, modulo	Left-to-right
6	Arithmetic: addition	+ -	addition, subtraction	Left-to-right
7	Bitwise shift	<< >>	shift left, shift right	Left-to-right
8	Relational	< > <= >=	comparison operators	Left-to-right
9	Equality	== !=	equality / inequality	Left-to-right
10	And	&	bitwise AND	Left-to-right
11	Exclusive or	^	bitwise XOR	Left-to-right
12	Inclusive or		bitwise OR	Left-to-right
13	Conjunction	&&	logical AND	Left-to-right
14	Disjunction		logical OR	Left-to-right
15	Assignment-level expressions	= *= /= %= += -=>>= <<= &= ^= =	assignment / compound assignment	Right-to-left
		?:	conditional operator	
16	Sequencing	,	comma separator	Left-to-right

Podstawowe wejście/wyjście

stream	description
<code>cin</code>	standard input stream
<code>cout</code>	standard output stream
<code>cerr</code>	standard error (output) stream
<code>clog</code>	standard logging (output) stream

Standardowe wyjście

- Najczęściej na ekran
- Używane z operatorem <<

```
1 cout << "Output sentence"; // prints Output sentence on screen
2 cout << 120;                // prints number 120 on screen
3 cout << x;                  // prints the value of x on screen
```

```
1 cout << "Hello"; // prints Hello
2 cout << Hello;   // prints the content of variable Hello
```

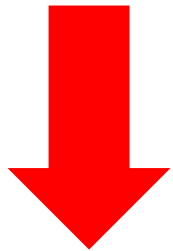
- Jest możliwe użycie wielokrotnego <<

```
cout << "This " << " is a " << "single C++ statement";
```


Standardowe wejście

- Najczęściej jest to klawiatura
- Używane z operatorem >>
- Dozwolone jest:

```
cin >> a >> b;
```



```
1 cin >> a;  
2 cin >> b;
```

```
1 // i/o example  
2  
3 #include <iostream>  
4 using namespace std;  
5  
6 int main ()  
7 {  
8     int i;  
9     cout << "Please enter an integer value: ";  
10    cin >> i;  
11    cout << "The value you entered is " << i;  
12    cout << " and its double is " << i*2 << ".\n";  
13    return 0;  
14 }
```

cin i stringi

```
1 string mystring;  
2 cin >> mystring;
```

- Można używać do ze stringami ale wszystkie białe znaki są traktowane jako zakończenie stringu

```
1 // cin with strings  
2 #include <iostream>  
3 #include <string>  
4 using namespace std;  
5  
6 int main ()  
7 {  
8     string mystr;  
9     cout << "What's your name? ";  
10    getline (cin, mystr);  
11    cout << "Hello " << mystr << ".\n";  
12    cout << "What is your favorite team? ";  
13    getline (cin, mystr);  
14    cout << "I like " << mystr << " too!\n";  
15    return 0;  
16 }
```

Instrukcje sterujące – instrukcja if

if (warunek) instrukcja

```
1 if (x == 100)
2     cout << "x is 100";
```

Blok instrukcji

Gdy chcemy wykonać więcej niż jedną instrukcję to ograniczamy je nawiasami {...}

```
{
instr1;
instr2;
instr3;
}
```

```
1 if (x == 100)
2 {
3     cout << "x is ";
4     cout << x;
5 }
```

Instrukcje sterujące – instrukcja if

if (warunek) instrukcja1 else instrukcja2

```
1 if (x == 100)
2     cout << "x is 100";
3 else
4     cout << "x is not 100";
```

```
1 if (x > 0)
2     cout << "x is positive";
3 else if (x < 0)
4     cout << "x is negative";
5 else
6     cout << "x is 0";
```

Pętla - while

while (wyrażenie) instrukcja

najpierw jest obliczana wartość **wyrażenia** – gdy jest niezerowa (prawda) to wykonana jest **instrukcja**, a potem znowu obliczana jest wartość wyrażenia i tak dalej aż wyrażenie nie przyjmie wartości zerowej (fałsz).

```
1 // custom countdown using while
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int n = 10;
8
9     while (n>0) {
10         cout << n << ", ";
11         --n;
12     }
13
14     cout << "liftoff!\n";
15 }
```

Pętla - do-while

do instrukcja while (wyrażenie);

Wartość wyrażenia jest badana po wykonaniu instrukcji, a nie przed, jak w petli while. Instrukcja zostanie wykonana **co najmniej jeden raz**, nawet gdy wyrażenie nie będzie prawdziwe od początku.

```
1 // echo machine
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string str;
9     do {
10         cout << "Enter text: ";
11         getline (cin,str);
12         cout << "You entered: " << str << '\n';
13     } while (str != "goodbye");
14 }
```

Pętla - for

for (inicjalizacja; warunek; zwiększenie) instrukcja;

najpierw wykonywana jest inicjalizacja, następnie sprawdzany jest warunek gdy jest prawda (różny od zera) to wykonywana jest treść pętli, a na końcu wykonywana jest zwiększenie (krok) i wracamy do początku. Pętla wykonywana jest tak długo aż warunek nie przybierze wartości zero (fałsz).

```
1 // countdown using a for loop
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     for (int n=10; n>0; n--) {
8         cout << n << ", ";
9     }
10    cout << "liftoff!\n";
11 }
```

Pętla for z operatorem zakresu

for (deklaracja : zakres) instrukcja;

```
1 // range-based for loop
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     string str {"Hello!"};
9     for (char c : str)
10    {
11        cout << "[" << c << "];"
12    }
13    cout << '\n';
14 }
```


Pętle nieskończone

for(; ;)

while(1)

Instrukcja break

- Przerywa instrukcję switch, powoduje także natychmiastowe zakończenie/przerwanie wykonywania pętli; for, while, do ... while.
- Jeżeli mamy do czynienia z kilkoma pętlami, zagnieżdżonymi jedna wewnątrz drugiej, to instrukcja break powoduje przerwanie tylko tej najbardziej wewnętrznej pętli, w której występuje, tkwi. Jest to przerwanie z wyjściem o jeden poziom wyżej.

```
1 // break loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     for (int n=10; n>0; n--)
8     {
9         cout << n << ", ";
10        if (n==3)
11        {
12            cout << "countdown aborted!";
13            break;
14        }
15    }
16 }
```

Instrukcja continue

- Powoduje zaniechanie instrukcji będących treścią pętli, lecz w przeciwieństwie do instrukcji break nie pętla nie zostaje przzerwana.

```
1 // continue loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     for (int n=10; n>0; n--) {
8         if (n==5) continue;
9         cout << n << ", ";
10    }
11    cout << "liftoff!\n";
12 }
```

Instrukcja goto

- instrukcja skoku do miejsca oznaczonego etykietą

```
1 // goto loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int n=10;
8 mylabel:
9     cout << n << ", ";
10    n--;
11    if (n>0) goto mylabel;
12    cout << "liftoff!\n";
13 }
```

Instrukcja switch

switch (wyrażanie)

```
{  
  case stała1:  
    grupa_instrukcji-1;  
    break;  
  case stała2:  
    grupa_instrukcji -2;  
    break;  
  ...  
  default:  
    domyslna_grupa_instrukcji  
}
```

Instrukcja switch

- Obliczane jest wyrażenie w nawiasie przy słowie **switch**, jeżeli jego wartość odpowiada którejś z wartości podanej w jednej z etykiet **case**, wówczas wykonywane są instrukcje począwszy od tej etykiety, wykonywanie ich kończy się po napotkaniu instrukcji **break**. Powoduje to wyskok z instrukcji **switch**, poza jej końcową klamrę. Jeżeli wartość wyrażenia nie zgadza się z żadną z wartości podanych przy etykietach **case**, to wykonywane są instrukcje po etykiecie **default**.
- Etykieta **default** może być w dowolnym miejscu instrukcji **switch**, nawet na jej samym początku, lub może jej w ogóle nie być.
- Jeżeli wartość wyrażenie nie zgadza się z żadną z wartości przy etykietach **case**, a etykiety **default** nie ma wcale, to opuszcza się instrukcję **switch** nie wykonując niczego. Instrukcji występujących po etykiecie **case** nie musi kończyć instrukcja **break**, gdy jej nie ma, to wykonywane są instrukcje po następnej etykiecie **case**.
- C++ po napotkaniu przypadku zgodnego z badanym wyrażeniem zakłada, że wszystkie umieszczone po tym przypadku wartości wyrażen przy kolejnych słowach **case** w klamrach instrukcji **switch** są także równe wyrażeniu, a więc następujące po nich instrukcje są wykonywane.

Instrukcja switch

```
1 switch (x) {  
2     case 1:  
3     case 2:  
4     case 3:  
5         cout << "x is 1, 2 or 3";  
6         break;  
7     default:  
8         cout << "x is not 1, 2 nor 3";  
9 }
```

switch example	if-else equivalent
<pre>switch (x) { case 1: cout << "x is 1"; break; case 2: cout << "x is 2"; break; default: cout << "value of x unknown"; }</pre>	<pre>if (x == 1) { cout << "x is 1"; } else if (x == 2) { cout << "x is 2"; } else { cout << "value of x unknown"; }</pre>

Funkcje

- Funkcja jest grupą powiązanych deklaracji i instrukcji realizujących określone zadania. Treść funkcji ujęta jest w nawiasy klamrowe
- Deklaracja funkcji:

typ nazwa_funkcji(lista parametrów); //średnik występuje gdy deklarujemy funkcje

- Definicja funkcji:

typ nazwa_funkcji(lista parametrów) {
treść funkcji

...

...

...

}

Funkcje

```
1 // function example
2 #include <iostream>
3 using namespace std;
4
5 int addition (int a, int b)
6 {
7     int r;
8     r=a+b;
9     return r;
10 }
11
12 int main ()
13 {
14     int z;
15     z = addition (5,3);
16     cout << "The result is " << z;
17 }
```

Funkcje

- Wartość zwracana przez funkcje znajduje się przy instrukcji:
return
- Instrukcja ta kończy działanie funkcji i jeżeli po tej instrukcji znajdują się jakieś inne instrukcje nie będą one wykonywane.
- Typ wartości zwracanej przez funkcji zależy od deklaracji funkcji.

Przykłady deklaracji funkcji

float kwadrat(int bok); \\ zwraca wartość rzeczywistą, parametr bok

void fun(int st, char znak, int nic); \\ nie zwraca wartość,

int f2(void); \\ zwraca wartość całkowitą, brak parametrów

char znak(); \\ zwraca wartość znakową, brak parametrów

void pin(...); \\ nie zwraca wartości, dowolna liczba parametrów

Int odd(int, int) \\ zwraca wartość całkowitą, dwa parametry – **brak nazw parametrów**

- Jeżeli funkcja nie zwraca żadnej wartości należy ją poprzedzić słowem **void**
- **return** na końcu funkcji (**void**) nie jest konieczne gdyż obecność **return** na końcu funkcji typu **void** jest domniemana
- Jeżeli do funkcji nie przekazujemy parametrów to należy wpisywać słowo **void**
- Deklaracja **f();** oznacza **f(void);**
- Wywołanie funkcji polega na podaniu jej nazwy z argumentami w nawiasach

Przesyłanie parametrów do funkcji przez wartość

- Przesyłanie parametrów do funkcji przez wartość powoduje, że przesyłamy tylko wartość liczbową służy ona inicjalizacji parametru formalnego czyli zmiennej lokalnej tworzonej w obrębie funkcji.
- Funkcja pracuje na tej kopii i nie modyfikuje go po wyjściu.

```
1 int x=5, y=3, z;  
2 z = addition ( x, y );
```

```
int addition (int a, int b)  
              ↑      ↑  
z = addition ( 5 , 3 );  
              . . . . .
```

Przesyłanie argumentów przez referencje

- Referencja jest to typ pochodny. Jest to jakby przezwisko jakiegoś obiektu, dzięki jej na tę sam obiekt możemy nałożyć drugą nazwę.
- Deklarujemy go w następujący sposób:

int &k;

float &x;

- Przesyłanie parametrów funkcji przez referencję pozwalają tej funkcji na modyfikowanie zmiennych (nawet lokalnych) znajdujących się poza funkcją.

```
void duplicate (int& a,int& b,int& c)
               ↑x   ↑y   ↑z
               ↓x   ↓y   ↓z
duplicate (  x   ,  y   ,  z   );
```

Przesyłanie argumentów przez referencje

```
1 // passing parameters by reference
2 #include <iostream>
3 using namespace std;
4
5 void duplicate (int& a, int& b, int& c)
6 {
7     a*=2;
8     b*=2;
9     c*=2;
10 }
11
12 int main ()
13 {
14     int x=1, y=3, z=7;
15     duplicate (x, y, z);
16     cout << "x=" << x << ", y=" << y << ", z=" << z;
17     return 0;
18 }
```

Wydajność – zapobieganie zmiany argumentów w przesyłaniu przez referencje

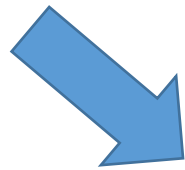
```
1 string concatenate (string a, string b)
2 {
3     return a+b;
4 }
```

Niepotrzebna kopia – strata na wydajności




```
1 string concatenate (string& a, string& b)
2 {
3     return a+b;
4 }
```

Możliwa modyfikacja argumentów



```
1 string concatenate (const string& a, const string& b)
2 {
3     return a+b;
4 }
```

Parametry domniemane

void fun2(int par1, int par2=3);  **fun2(1,2)**
lub
fun2(1)

- kilka parametrów domniemanych, to argumenty te muszą być na końcu listy

void funx(int a, float b=1.6, int c=12, char ch='a');

W języku C++ nie może wystąpić sytuacja, że obok siebie stoją dwa przecinki

Parametry domniemane

```
1 // default values in functions
2 #include <iostream>
3 using namespace std;
4
5 int divide (int a, int b=2)
6 {
7     int r;
8     r=a/b;
9     return (r);
10 }
11
12 int main ()
13 {
14     cout << divide (12) << '\n';
15     cout << divide (20,4) << '\n';
16     return 0;
17 }
```

Zakres ważności nazw deklarowanych wewnątrz funkcji

- Zakres ważności nazw deklarować wewnątrz funkcji ogranicza się tylko do bloku tej funkcji.
- Etykieta deklarowana wewnątrz funkcji jest też jej nazwa wewnętrzna, więc nie można skoczyć do etykiety (znajdującej się wewnątrz funkcji) wykorzystując instrukcję **goto** zapisaną poza blokiem funkcji
- Ponieważ etykieta jest lokalna to w dwóch różnych funkcjach mogą istnieć te same etykiety

Deklaracja static

```
1.  #include <iostream.h>
2.  void f1(void);
3.  void f2(void);
4.  main()
5.  {
6.      f1();
7.      f2();
8.      f1();
9.      f2();
10. }
11. void f1(void)
12. {
13.     static int ilef1=0;
14.     ilef1++;
15.     cout <<"f1 wywolana po raz"<<ilef1<<endl;
16. }
17. void f2(void)
18. {
19.     static int ilef2=100;
20.     ilef2++;
21.     cout <<"f1 wywolana po raz"<<ilef2<<endl;
22. }
```

Wynik działania programu:

f1 wywalana po raz 1
f1 wywalana po raz 101
f1 wywalana po raz 2
f1 wywalana po raz 102

gdyby nie było instrukcji **static** to otrzymalibyśmy:

f1 wywalana po raz 1
f1 wywalana po raz 101
f1 wywalana po raz 1
f1 wywalana po raz 101

Przeciążenie nazw funkcji (overloading)

- W języku angielskim przeciążenie/przeładowanie (overloading) danego słowa oznacza, że ma ono więcej niż jedno znaczenie. Słowo jest przeciążone znaczeniami.
- Przeciążenie nazwy funkcji polega na tym, że w danym zakresie ważności jest więcej niż jedna funkcja o takiej samej nazwie. To, która z nich zostaje w danym momencie uaktywniona zależy do typu argumentów jej wywołania. Funkcje takie mają tę samą nazwę, ale muszą się różnić typem poszczególnych argumentów.

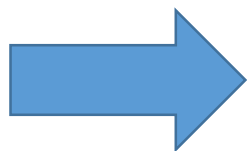
```
void write (int i);  
void write (char a, float x, char b);
```

```
write ('A', 7.13, 'c');
```

Przestrzenie nazw

- Uważne raczej dla zmiennych, funkcji globalnych (gdzie konflikt jest bardziej prawdopodobny)

```
namespace identifier
{
    named_entities
}
```



```
1 namespace myNamespace
2 {
3     int a, b;
4 }
```



```
1 myNamespace::a
2 myNamespace::b
```

```
1 namespace foo { int a; }
2 namespace bar { int b; }
3 namespace foo { int c; }
```

Możliwe rozdzielenie deklaracji

- Aliasy

namespace new_name = current_name;

Przestrzenie nazw

```
1 // namespaces
2 #include <iostream>
3 using namespace std;
4
5 namespace foo
6 {
7     int value() { return 5; }
8 }
9
10 namespace bar
11 {
12     const double pi = 3.1416;
13     double value() { return 2*pi; }
14 }
15
16 int main () {
17     cout << foo::value() << '\n';
18     cout << bar::value() << '\n';
19     cout << bar::pi << '\n';
20     return 0;
21 }
```

Przestrzenie nazw - using

- Wiąże nazwę z przestrzenią

```
1 // using
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8     int y = 10;
9 }
10
11 namespace second
12 {
13     double x = 3.1416;
14     double y = 2.7183;
15 }
16
17 int main () {
18     using first::x;
19     using second::y;
20     cout << x << '\n';
21     cout << y << '\n';
22     cout << first::y << '\n';
23     cout << second::x << '\n';
24     return 0;
25 }
```

Przestrzenie nazw - using

- Można także powiązać całą przestrzeń

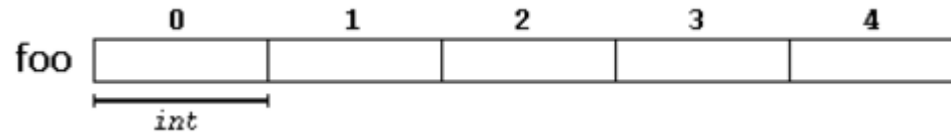
```
1 // using
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8     int y = 10;
9 }
10
11 namespace second
12 {
13     double x = 3.1416;
14     double y = 2.7183;
15 }
16
17 int main () {
18     using namespace first;
19     cout << x << '\n';
20     cout << y << '\n';
21     cout << second::x << '\n';
22     cout << second::y << '\n';
23     return 0;
24 }
```

- Bloki ograniczają powiązanie

```
1 // using namespace example
2 #include <iostream>
3 using namespace std;
4
5 namespace first
6 {
7     int x = 5;
8 }
9
10 namespace second
11 {
12     double x = 3.1416;
13 }
14
15 int main () {
16     {
17         using namespace first;
18         cout << x << '\n';
19     }
20     {
21         using namespace second;
22         cout << x << '\n';
23     }
24     return 0;
25 }
```


Tablice

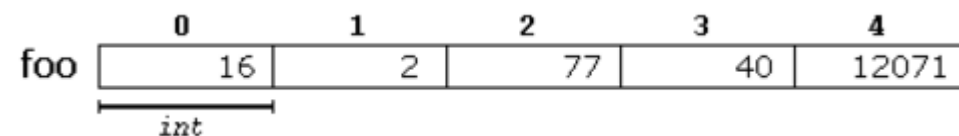
- Typ złożony - to szereg elementów tego samego typu, umieszczonych w sąsiednich miejscach pamięci, do których można się indywidualnie odwoływać, dodając indeks jako unikatowego identyfikatora.



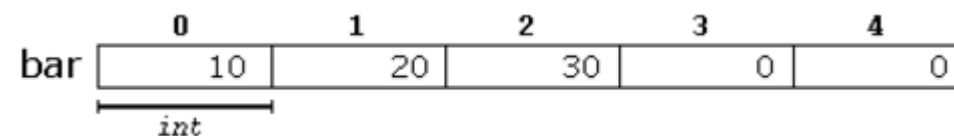
- Deklaracja
 - typ nazwa [liczba elementów];**
 - char znak[20];**
 - float numer[2];**
 - int liczba[2][2];**
 - int wiele[2][2][2];**
- Numeracja tablicy zaczyna się od zera

Tablice - inicjalizacja

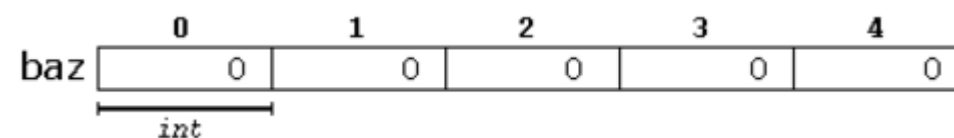
```
int foo [5] = { 16, 2, 77, 40, 12071 };
```



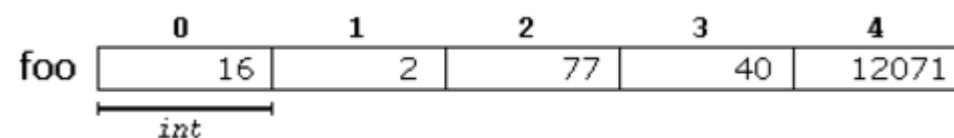
```
int bar [5] = { 10, 20, 30 };
```



```
int baz [5] = { };
```

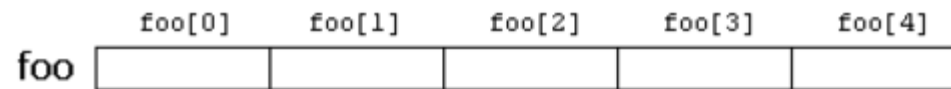


```
int foo [] = { 16, 2, 77, 40, 12071 };
```



Tablice – dostęp do elementów tablicy

- Dostęp jest poprzez indeks

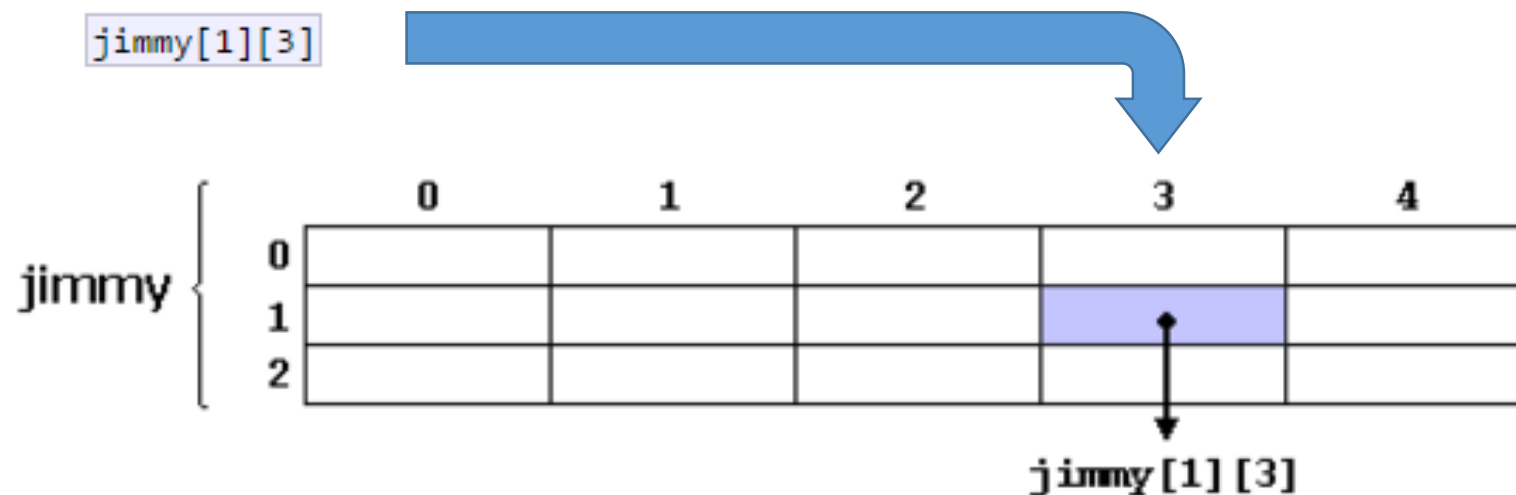
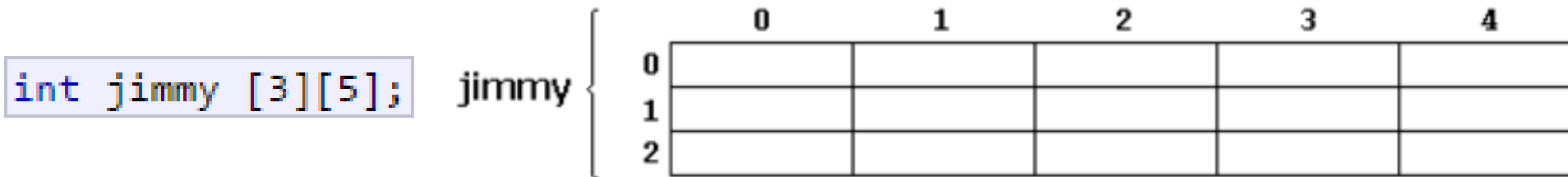


```
foo [2] = 75;
```

```
1 foo[0] = a;  
2 foo[a] = 75;  
3 b = foo [a+2];  
4 foo[foo[a]] = foo[2] + 5;
```

Tablice wielowymiarowe

- Można je nazwać tablicami tablic



Ciekawostka

multidimensional array	pseudo-multidimensional array
<pre>#define WIDTH 5 #define HEIGHT 3 int jimmy [HEIGHT][WIDTH]; int n,m; int main () { for (n=0; n<HEIGHT; n++) for (m=0; m<WIDTH; m++) { jimmy[n][m]=(n+1)*(m+1); } }</pre>	<pre>#define WIDTH 5 #define HEIGHT 3 int jimmy [HEIGHT * WIDTH]; int n,m; int main () { for (n=0; n<HEIGHT; n++) for (m=0; m<WIDTH; m++) { jimmy[n*WIDTH+m]=(n+1)*(m+1); } }</pre>

Przekazywanie tablicy do funkcji

- Deklaracje funkcji pracującej na tablicy można zapisać w następujący sposób:

void funkcja(float t[]); **//deklaracja funkcji**

...

float tablica[]={7, 8.1, 12.2}; **// definicja tablicy**

...

funkcja(tablica); **//wywołanie funkcji**

- Przekazanie tablicy do funkcji odbywa się poprzez przekazanie jej nazwy
- W języku C++ nazwa tablicy jest równocześnie adresem jej zerowego elementu
- Działanie funkcji modyfikuje tablice

Przekazywanie tablicy do funkcji

```
1 // arrays as parameters
2 #include <iostream>
3 using namespace std;
4
5 void printarray (int arg[], int length) {
6     for (int n=0; n<length; ++n)
7         cout << arg[n] << ' ';
8     cout << '\n';
9 }
10
11 int main ()
12 {
13     int firstarray[] = {5, 10, 15};
14     int secondarray[] = {2, 4, 6, 8, 10};
15     printarray (firstarray,3);
16     printarray (secondarray,5);
17 }
```

Przekazywanie tablicy do funkcji

- Można także do funkcji przesyłać jeden element tablicy:

```
int f1(int x);           //deklaracja funkcji
int tab[4]={1,2,3,4};    //definicja tablicy
int y;                   //definicja zmiennej całkowitej y
y=3+f1(tab[2]);          //wywołanie funkcji f1
```

- Przesyłanie tablic wielowymiarowych do funkcji

```
long widmo[4][8192];     //deklaracja tablicy
void fun(long t[][8192]); //deklaracja funkcji pracującej na tablicy wielowymiarowej
fun(widmo);
```

- oczywiście poprawna byłaby także deklaracja tablicy

```
void fun(long t[4][8192]); //deklaracja funkcji pracującej na tablicy wielowymiarowej
void fun(long t[][]);      //deklaracja BŁĘDNA
```


Wskaźniki

- Wskaźnik - jest to typem pochodnym i jest obiektem w którym można umieścić adres jakiegoś innego obiektu w pamięci.
- Przykładowe definicje wskaźników:

```
int *w ;  
char *wsk_do_znakow ;  
float *wsk_do_rzeczywistych ;
```

- Aby wskaźnikowi nadać wartość (czyli sprawić aby na coś pokazywał) należy użyć jednoargumentowego operatora **&** :

```
int *w ;           //def. wskaźnika  
int k=3 ;          //def. zwykłego obiektu  
w= &k;             // ustawienie wskaźnika na obiekt  
cout << "W obiekcie wskazywanym przez wskaźnik w jest wartość" << (*w);
```

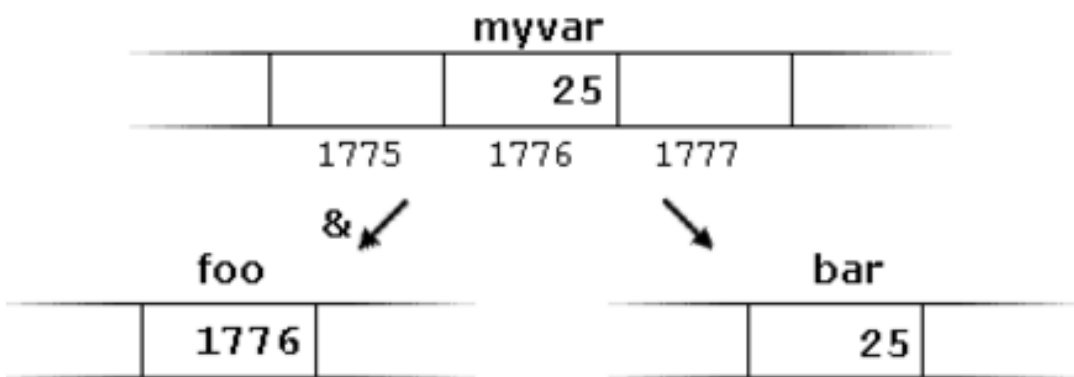
- Operator ***** kieruje nas do obiektu jaki jest wskazywany przez wskaźnik.

```
cout << k;  
cout << *w;
```

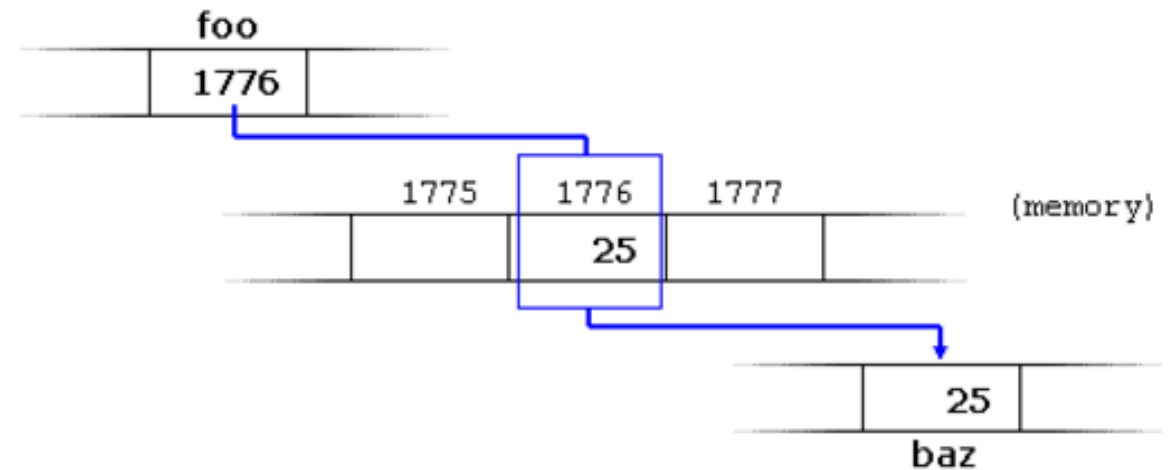
jest jednoznaczne

Operator adresu & i Operator *

```
1 myvar = 25;  
2 foo = &myvar;  
3 bar = myvar;
```



```
baz = *foo;
```



Wskaźniki

```
1 // my first pointer
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int firstvalue, secondvalue;
8     int * mypointer;
9
10    mypointer = &firstvalue;
11    *mypointer = 10;
12    mypointer = &secondvalue;
13    *mypointer = 20;
14    cout << "firstvalue is " << firstvalue << '\n';
15    cout << "secondvalue is " << secondvalue << '\n';
16    return 0;
17 }
```

```
firstvalue is 10
secondvalue is 20
```

Wskaźniki

- Najczęściej wskaźniki stosuje się gdy chodzi nam o:
 - ulepszanie pracy z tablicami,
 - funkcje mogące zmieniać wartość przesłanych do nich obiektów,
 - dostęp do specjalnych komórek pamięci,
 - rezerwacje obszarów pamięci.

Wskaźniki i tablice

- Przykład ustawienia wskaźnika:

```
int *wsk;           // definicja wskaźnika
int tab[10];        // definicja tablicy
wsk = &tab[n];      // ustawienie wskaźnika na elemencie tablicy o indeksie n

wsk = &tab[0];      // jest ustawieniem wskaźnika na element zerowy tablicy

wsk = tab;          // jest także ustawieniem wskaźnika na element zerowy tablicy gdyż
                    // nazwa tablicy jest równocześnie adresem jej początku

wsk = &tab[4];      // ustawienie wskaźnika na 4 element tablicy

wsk = wsk + 1;      // przesunięcie wskaźnika z 4 na 5 element tablicy

wsk = wsk++;        // to sam o tylko krócej

wsk += n;           //przesunięcie wskaźnika o n elementów
```

- Wskaźnik i obiekt na jaki on wskazuje muszą być tego samego typu
- Dodanie do wskaźnika jakiegokolwiek liczby całkowitej powoduje, że pokazuje on tyleż elementów tablicy dalej. Niezależnie od tego jakie są te elementy

Wskaźniki i tablice

```
1 // more pointers
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     int numbers[5];
8     int * p;
9     p = numbers; *p = 10;
10    p++; *p = 20;
11    p = &numbers[2]; *p = 30;
12    p = numbers + 3; *p = 40;
13    p = numbers; *(p+4) = 50;
14    for (int n=0; n<5; n++)
15        cout << numbers[n] << ", ";
16    return 0;
17 }
```

10, 20, 30, 40, 50,

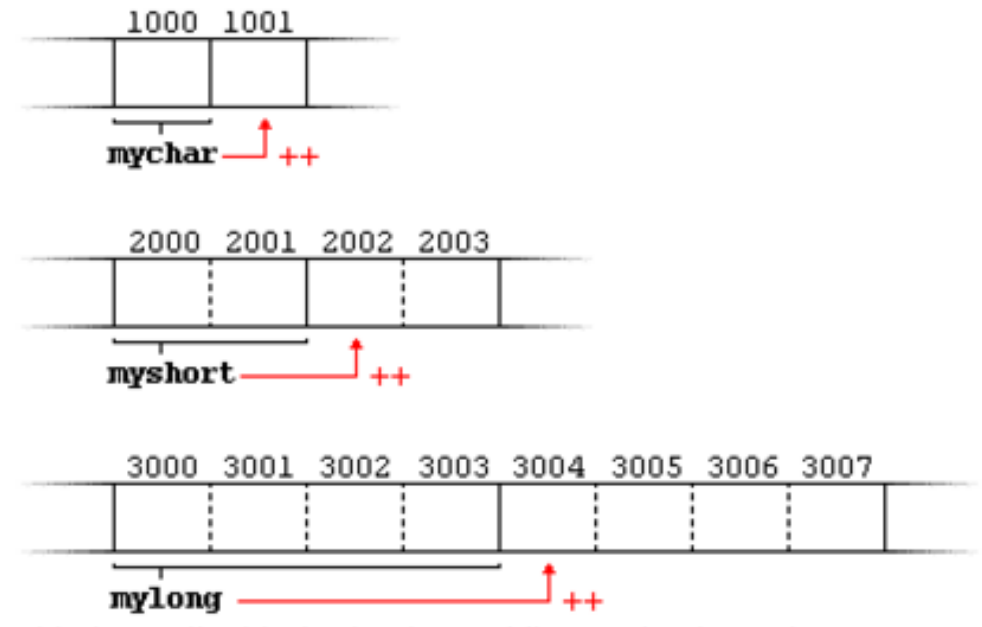
Arytmetyka wskaźników

- Możliwe jest tylko dodawanie i odejmowanie. Jak deko „skakamy” w pamięci zależy od typu wskaźnika

```
1 char *mychar;  
2 short *myshort;  
3 long *mylong;
```

```
1 ++mychar;  
2 ++myshort;  
3 ++mylong;
```

```
1 mychar = mychar + 1;  
2 myshort = myshort + 1;  
3 mylong = mylong + 1;
```

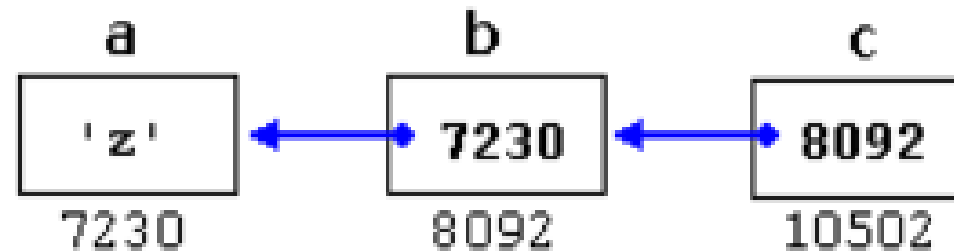


```
1 *p++ // same as *(p++): increment pointer, and dereference unincremented address  
2 *++p // same as *(++p): increment pointer, and dereference incremented address  
3 ++*p // same as ++(*p): dereference pointer, and increment the value it points to  
4 (*p)++ // dereference pointer, and post-increment the value it points to
```

Wskaźniki do wskaźników

- Dokładamy kolejną gwiazdkę

```
1 char a;  
2 char * b;  
3 char ** c;  
4 a = 'z';  
5 b = &a;  
6 c = &b;
```



Wskaźniki – nie wskazujące

```
1 int * p = 0;  
2 int * q = nullptr;
```

```
int * r = NULL;
```

Wskaźniki jako argumenty funkcji

```
#include <iostream.h>
```

```
void f1(int *wsk_do_arg);  
/*****/  
main()  
{  
    int a=-1;  
    cout<<"Przed wywołaniem funkcji f1 "<<a<<endl;  
    f1(&a);  
    cout<<"Po wywołaniu funkcji f1 "<<a<<endl;  
}  
/*****/  
void f1(int *wsk_do_arg)  
{  
    *wsk_do_arg=100;  
}
```

Wskaźniki przy przesyłaniu tablic do funkcji

```
#include <iostream.h>

void f_wsk1(int *wsk, int rozm);
void f_tab1(int tab[], int rozm);
void f_wsk2(int *wsk, int rozm);
/*****/
main()
{
    int tafla[4]={5,10,15,20};
    f_wsk1(tafla,4);
    f_tab1(tafla,4);
    f_wsk2(tafla,4);
}
/*****/
void f_tab1(int tab[], int rozm)
{
    cout << "\n Wewnatrz f1_tab \n";
    for (int i=0; i<rozm; i++)
    {
        cout<<tab[i]<<"\t";
    }
}
```

```
/*****/
void f_wsk1(int *wsk, int rozm)
{
    cout << "\n Wewnatrz f1_wsk1 \n";
    for (int i=0; i<rozm; i++)
    {
        cout<<*(wsk++)<<"\t";
    }
}
/*****/
void f_wsk2(int *wsk, int rozm)
{
    cout << "\n Wewnatrz f1_tab \n";
    for (int i=0; i<rozm; i++)
    {
        cout<<wsk[i]<<"\t";
    }
}
```

Wskaźniki do funkcji

```
1 // pointer to functions
2 #include <iostream>
3 using namespace std;
4
5 int addition (int a, int b)
6 { return (a+b); }
7
8 int subtraction (int a, int b)
9 { return (a-b); }
10
11 int operation (int x, int y, int (*functocall)(int,int))
12 {
13     int g;
14     g = (*functocall)(x,y);
15     return (g);
16 }
17
18 int main ()
19 {
20     int m,n;
21     int (*minus)(int,int) = subtraction;
22
23     m = operation (7, 5, addition);
24     n = operation (20, m, minus);
25     cout <<n;
26     return 0;
27 }
```

Rezerwacja obszarów pamięci – operatory **new** i **delete**, dynamiczna alokacja tablicy

- Operator **new** powoduje tworzenie, a operator **delete** unicestwianie obiektów
- Cechy obiektów utworzonych operatorem **new**
 - obiekty te istnieją aż do ich usunięcia operatorem **delete**,
 - obiekty te nie mają nazwy, ale można nimi operować przy pomocy wskaźników,
 - obiekty te nie są inicjalizowane zerami.

```
char *wsk;
```

```
wsk = new char;           //utworzenie obiektu typu  
//char bez nazwy, ale jego adres przekazywany  
//jest do wsk
```

```
delete wsk;              //powoduje    likwidację  
tego obiektu
```

```
float *w;
```

```
w = new float[15];       //utworzenie tablicy typu  
//float, tablica niema nazwy ale wskaźnik zna jej  
//adres
```

```
delete [] w;             //kasowanie  wyżej    utworzonej  
tablicy
```

Struktura

- Często występuje sytuacja, gdy chcemy dla wygody zgromadzić pod jedną, wspólną nazwą dane różnych typów. Gdy mamy konglomerat danych różnych typów, wtedy C++ oferuje pojęcie struktury.

struct nazwa_struktury

{

...

składniki, elementy, zmienne, pola struktury

...

} lista zmiennych typu strukturalnego (opcjonalnie) ;

Struktura

```
1 struct product {  
2     int weight;  
3     double price;  
4 } ;  
5  
6 product apple;  
7 product banana, melon;
```

```
1 struct product {  
2     int weight;  
3     double price;  
4 } apple, banana, melon;
```

Struktura

- Odwołanie do pola (składowej, elementu) struktury - kropka

```
1 apple.weight  
2 apple.price  
3 banana.weight  
4 banana.price  
5 melon.weight  
6 melon.price
```



```

1 // example about structures
2 #include <iostream>
3 #include <string>
4 #include <sstream>
5 using namespace std;
6
7 struct movies_t {
8     string title;
9     int year;
10 } mine, yours;
11
12 void printmovie (movies_t movie);
13
14 int main ()
15 {
16     string mystr;
17
18     mine.title = "2001 A Space Odyssey";
19     mine.year = 1968;
20
21     cout << "Enter title: ";
22     getline (cin,yours.title);
23     cout << "Enter year: ";
24     getline (cin,mystr);
25     stringstream(mystr) >> yours.year;
26
27     cout << "My favorite movie is:\n ";
28     printmovie (mine);
29     cout << "And yours is:\n ";
30     printmovie (yours);
31     return 0;
32 }
33
34 void printmovie (movies_t movie)
35 {
36     cout << movie.title;
37     cout << " (" << movie.year << ")\n";
38 }

```

Enter title: Alien

Enter year: 1979

My favorite movie is:

2001 A Space Odyssey (1968)

And yours is:

Alien (1979)

```

1 // array of structures
2 #include <iostream>
3 #include <string>
4 #include <sstream>
5 using namespace std;
6
7 struct movies_t {
8     string title;
9     int year;
10 } films [3];
11
12 void printmovie (movies_t movie);
13
14 int main ()
15 {
16     string mystr;
17     int n;
18
19     for (n=0; n<3; n++)
20     {
21         cout << "Enter title: ";
22         getline (cin,films[n].title);
23         cout << "Enter year: ";
24         getline (cin,mystr);
25         stringstream(mystr) >> films[n].year;
26     }
27
28     cout << "\nYou have entered these movies:\n";
29     for (n=0; n<3; n++)
30         printmovie (films[n]);
31     return 0;
32 }
33
34 void printmovie (movies_t movie)
35 {
36     cout << movie.title;
37     cout << " (" << movie.year << ")\n";
38 }

```

```

Enter title: Blade Runner
Enter year: 1982
Enter title: The Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

```

```

You have entered these movies:
Blade Runner (1982)
The Matrix (1999)
Taxi Driver (1976)

```

Wskaźniki do struktury

- Nie ma różnic w stosunku do wskaźników dla innych typów

```
1 struct movies_t {  
2     string title;  
3     int year;  
4 };  
5  
6 movies_t amovie;  
7 movies_t * pmovie;
```

```
pmovie = &amovie;
```

```

1 // pointers to structures
2 #include <iostream>
3 #include <string>
4 #include <sstream>
5 using namespace std;
6
7 struct movies_t {
8     string title;
9     int year;
10 };
11
12 int main ()
13 {
14     string mystr;
15
16     movies_t amovie;
17     movies_t * pmovie;
18     pmovie = &amovie;
19
20     cout << "Enter title: ";
21     getline (cin, pmovie->title);
22     cout << "Enter year: ";
23     getline (cin, mystr);
24     (stringstream) mystr >> pmovie->year;
25
26     cout << "\nYou have entered:\n";
27     cout << pmovie->title;
28     cout << " (" << pmovie->year << ")\n";
29
30     return 0;
31 }

```

```

Enter title: Invasion of the body snatchers
Enter year: 1978

```

```

You have entered:
Invasion of the body snatchers (1978)

```

Operator strzałka ->

Zagnieżdżanie struktur

- Struktury mogą być zagnieżdżane

```
1 struct movies_t {  
2     string title;  
3     int year;  
4 };  
5  
6 struct friends_t {  
7     string name;  
8     string email;  
9     movies_t favorite_movie;  
10 } charlie, maria;  
11  
12 friends_t * pfriends = &charlie;
```

```
1 charlie.name  
2 maria.favorite_movie.title  
3 charlie.favorite_movie.year  
4 pfriends->favorite_movie.year
```

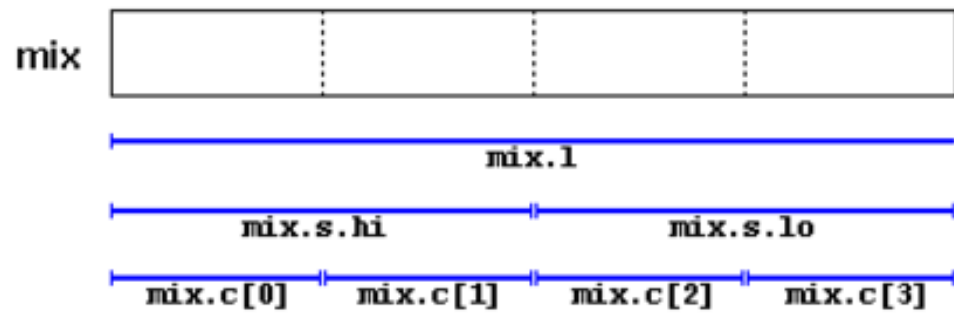
Unie

- Unia jest to tak jakby struktura zawierająca jedno miejsce w pamięci przeznaczone dla paru obiektów różnego typu.
- Wielkość unii zawsze jest równa miejscu zajmowanemu przez największy obiekt znajdujący się w unii
- Deklarując unię zawierającą 3 zmienne **int** przydzielamy jej miejsce o wielkości jednej zmiennej **int**

```
union type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

```
1 union mytypes_t {  
2     char c;  
3     int i;  
4     float f;  
5 } mytypes;
```

```
1 union mix_t {  
2     int l;  
3     struct {  
4         short hi;  
5         short lo;  
6     } s;  
7     char c[4];  
8 } mix;
```



Int -> 4 bajty
short -> 2 bajty
char -> 1 bajt

Typ wyliczeniowy **enum**

- To rodzaj danych, który zawiera już listę wartości, jaką można nadać zmiennej własnego typu **enum**

enum nazwa_typu

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

{

lista_wyliczeniowa

};

```
1 colors_t mycolor;  
2  
3 mycolor = blue;  
4 if (mycolor == green) mycolor = red;
```

Klasa

- Rozwinięcie struktury – w swoim składzie mogą zawierać także funkcje
- Funkcje klasy zwane są metodami tej klasy
- Składowe różnych typów to pola klasy
- Klasa grupuje składowe (dane, cechy) obiektu i funkcje działające na składowych klasy.
- Rzeczywisty obiekt można w komputerze opisać zespołem zmiennych (liczb) reprezentujących cechy obiektu i zespołem funkcji, reprezentujących zachowanie się obiektu
- Klasa to inaczej mówiąc typ

Klasa

```
class nasza_klasa {
```

```
...
```

```
...
```

// ciało klasy

```
...
```

```
};
```

- Po definicji klasy zawsze występuje średnik.

```
class class_name {  
    access_specifier_1:  
        member1;  
    access_specifier_2:  
        member2;  
    ...  
} object_names;
```

Klasy – dostęp do składowych

- Składowe klasy mogą być publiczne (**public**) lub prywatne (**private**). Jeżeli **public** to program ma do nich dostęp z zewnątrz, spoza zakresu klasy przy użyciu operatora wyboru z kropką
- **private** oznacza, że deklarowane zmienne i funkcje klasy są dostępne tylko z wnętrza klasy. Funkcje klasy mogą być wtedy wywołane tylko przez inne funkcje zawarte w tej samej klasie
- **protected** - jest dostępny tak jak składnik **private** ponadto związany jest z dziedziczeniem i dostęp mają tylko klasy zaprzyjaźnione i te, które dziedziczą po danej klasie
- **public, private, protected** można umieszczać w dowolnej kolejności, mogą się też powtarzać. Zawsze oznaczają, że te składniki klasy, które następują bezpośrednio po etykiecie - mają tak określony dostęp
- Domniemywa się, że - dopóki w definicji klasy nie wystąpi żadna z etykiet - składniki klasy mają dostęp **private**.
- Enkapsulacja - ukrywanie informacji. Sterowanie dostępem do składników klasy (składowych, funkcji) jest swego rodzaju dobrodziejstwem, które chroni dobrze napisaną klasę przed przypadkowym "zepsuciem". Wskazane jest ukrywanie jak największej ilości zmiennych, pozwala oderwać to kod klasy od bieżącego kontekstu i wykorzystać go w przyszłości.

Klasy

- Do składników klasy odnosimy się jak do składników struktury
- . (kropka) operator odniesienia do składników obiektu znanego z nazwy lub referencji
- -> (strzałka) operator odniesienia do wskaźnika
- Definicja klasy nie definiuje żadnych obiektów
- Klasa to typ obiektu, a nie sam obiekt

```
1 class Rectangle {  
2     int width, height;  
3     public:  
4     void set_values (int,int);  
5     int area (void);  
6 } rect;
```

```
1 rect.set_values (3,4);  
2 myarea = rect.area();
```

```
1 // classes example
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7     public:
8     void set_values (int,int);
9     int area() {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13     width = x;
14     height = y;
15 }
16
17 int main () {
18     Rectangle rect;
19     rect.set_values (3,4);
20     cout << "area: " << rect.area();
21     return 0;
22 }
```

Funkcje składowe klasy

- Funkcje składowe klasy to funkcje za pomocą, których dokonujemy operacji na danych składowych klasy w szczególności dla tych, które są z opcją **private**.

- Odwołanie:

obiekt.funkcja(parametry);

student1.zapisz("Jan Nowak", 26);

student2.zapisz("Zyzio Dyzma", 30);

- Za pomocą wskaźnika:

osoba *wsk; //definicja wskaźnika

wsk = &profesor //ustawianie wskaznika o obiekcie profesor

wsk -> zapisz("Jan Nikt", 55);

Funkcje składowe klasy

- Funkcja składowa może być definiowana w dwóch miejscach:
 - wewnątrz definicji klasy
 - poza klasą
- Zasada:
 - Jeżeli ciało funkcji ma nie więcej niż dwie linijki to definiujemy ją wewnątrz definicji klasy, natomiast, jeżeli funkcja jest dłuższa to definiujemy ją poza klasą.

```
1 // example: one class, two objects
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7     public:
8     void set_values (int,int);
9     int area () {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13     width = x;
14     height = y;
15 }
16
17 int main () {
18     Rectangle rect, rectb;
19     rect.set_values (3,4);
20     rectb.set_values (5,6);
21     cout << "rect area: " << rect.area() << endl;
22     cout << "rectb area: " << rectb.area() << endl;
23     return 0;
24 }
```

Konstruktor

- Służy do inicjowania zmiennych, nadawania im wartości
- Zawsze jest wołany gdy tworzony jest nowy obiekt klasy
- Jej nazwa musi być taka sama, jak nazwa klasy
- nazwa konstruktora nie może być poprzedzona typem zwracanej wartości, nawet typ **void** jest niedopuszczalny, przed nazwą konstruktora nic nie stoi
- Nazwa konstruktor jest nieco myląca, konstruktor nie definiuje obiektu, tylko nadaje wartości początkowe jego zmiennym, konstruktor nie jest obowiązkowy w klasie
- Konstruktor może być przeładowany

Konstruktor

```
1 // example: class constructor
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7     public:
8     Rectangle (int,int);
9     int area () {return (width*height);}
10 };
11
12 Rectangle::Rectangle (int a, int b) {
13     width = a;
14     height = b;
15 }
16
17 int main () {
18     Rectangle rect (3,4);
19     Rectangle rectb (5,6);
20     cout << "rect area: " << rect.area() << endl;
21     cout << "rectb area: " << rectb.area() << endl;
22     return 0;
23 }
```

Przeładowanie konstruktora

```
1 // overloading class constructors
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7     public:
8     Rectangle ();
9     Rectangle (int,int);
10    int area (void) {return (width*height);}
11 };
12
13 Rectangle::Rectangle () {
14     width = 5;
15     height = 5;
16 }
17
18 Rectangle::Rectangle (int a, int b) {
19     width = a;
20     height = b;
21 }
22
23 int main () {
24     Rectangle rect (3,4);
25     Rectangle rectb;
26     cout << "rect area: " << rect.area() << endl;
27     cout << "rectb area: " << rectb.area() << endl;
28     return 0;
29 }
```

- Konstruktor domyślny to taki, który nie ma parametrów
 - Jest wołany gdy tworzony jest obiekt
 - Nie może być wywoływany z nawiasami

```
1 Rectangle rectb;    // ok, default constructor called
2 Rectangle rectc();  // oops, default constructor NOT called
```

Konstruktor – sposoby wywołania

```
1 // classes and uniform initialization
2 #include <iostream>
3 using namespace std;
4
5 class Circle {
6     double radius;
7 public:
8     Circle(double r) { radius = r; }
9     double circum() {return 2*radius*3.14159265;}
10 };
11
12 int main () {
13     Circle foo (10.0);    // functional form
14     Circle bar = 20.0;    // assignment init.
15     Circle baz {30.0};    // uniform init.
16     Circle qux = {40.0}; // POD-like
17
18     cout << "foo's circumference: " << foo.circum() << '\n';
19     return 0;
20 }
```

- Nawiasy {} pozwalają wywołać konstruktor domyślny

```
1 Rectangle rectb; // default constructor called
2 Rectangle rectc(); // function declaration (default constructor NOT called)
3 Rectangle rectd{}; // default constructor called
```

Konstruktor

- Bezpośrednia inicjalizacja zmiennych – operator :

```
1 class Rectangle {  
2     int width,height;  
3     public:  
4         Rectangle(int,int);  
5         int area() {return width*height;}  
6 };
```

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

```
Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

Wskaźniki do obiektów klasy

```
1 // pointer to classes example
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7 public:
8     Rectangle(int x, int y) : width(x), height(y) {}
9     int area(void) { return width * height; }
10 };
11
12
13 int main() {
14     Rectangle obj (3, 4);
15     Rectangle * foo, * bar, * baz;
16     foo = &obj;
17     bar = new Rectangle (5, 6);
18     baz = new Rectangle[2] { {2,5}, {3,6} };
19     cout << "obj's area: " << obj.area() << '\n';
20     cout << "*foo's area: " << foo->area() << '\n';
21     cout << "*bar's area: " << bar->area() << '\n';
22     cout << "baz[0]'s area:" << baz[0].area() << '\n';
23     cout << "baz[1]'s area:" << baz[1].area() << '\n';
24     delete bar;
25     delete[] baz;
26     return 0;
27 }
```

Słowo **this**

- Wskaźnik **this** wskazuje na obiekt, dla którego została wywołana metoda

```
1 // example on this
2 #include <iostream>
3 using namespace std;
4
5 class Dummy {
6     public:
7         bool isitme (Dummy& param);
8 };
9
10 bool Dummy::isitme (Dummy& param)
11 {
12     if (&param == this) return true;
13     else return false;
14 }
15
16 int main () {
17     Dummy a;
18     Dummy* b = &a;
19     if ( b->isitme(a) )
20         cout << "yes, &a is b\n";
21     return 0;
22 }
```