



---

## **Zaawansowane Techniki Programowania**

*Podręcznik zawiera podstawowy materiał do przedmiotu pod tą  
samą nazwą*

WERSJA: 0.1

DATA: LISTOPAD, 2011

---



**KAPITAŁ LUDZKI**  
NARODOWA STRATEGIA SPÓJNOŚCI

**UNIA EUROPEJSKA**  
EUROPEJSKI  
FUNDUSZ SPOŁECZNY



## Informacje o Autorach:

Imię i Nazwisko:	<i>Paweł Wnuk</i>
Email:	<i>p.wnuk@mchtr.pw.edu.pl</i>
Stanowisko:	<i>Pracownik naukowo-dydaktyczny na stanowisku adiunkta</i>
Miejsce pracy:	

## Wstęp

Zacznijmy wstęp od końcówki wstępu do programowania obiektowego, czyli słów Bijarne Stroumtroupa:

„... Pisanie dobrych programów wymaga inteligencji, smaku i cierpliwości ...”

Już kilka projektów z programowania macie za sobą – teraz przyszła pora na typowe rozwiązania nietypowych problemów. Ten podręcznik zaczniemy od przypomnienia – przypomnienia kilku podstawowych zasad dotyczących programowania w języku C++. Jednak jego istotą będzie opowieść o różnych, luźno ze sobą powiązanych, lecz często występujących w praktyce problemach. Problemach, które rzadko opisywane są w książkach, natomiast często spotyka się je w praktyce.

Tak więc przedstawię Wam zasady tworzenia szablonów w języku C++, wraz z informacjami o ich podstawowym zastosowaniu – bibliotekach STL i BOOST. Potem przejdziemy do elementów inżynierii oprogramowania – pewnie słyszeliście już o wzorcach projektowych – w ramach tego podręcznika dowiecie się, jak implementować wybrane z nich w języku C++. Pokażę Wam także, w jaki sposób można dynamicznie ładować fragmenty kodu w trakcie działania programu, co w rezultacie prowadzi do uzyskania mechanizmu powszechnie znanego jako wtyczki. Zamieszcze przykład programów sieciowych – serwerów i klientów, pisanych w różny sposób, zarówno przy wykorzystaniu czystego API systemu operacyjnego, jak i korzystając z dodatkowych bibliotek. Pokażę Wam jak buduje się aplikacje wielowątkowe, i jak można zapewnić synchronizację pomiędzy wątkami. Opowiem o mechanizmie sygnałów / slotów, jego implementacji w Qt oraz możliwych wykorzystaniach.

Zapraszam więc do lektury.

# Spis treści

<b>Składnia i funkcjonalność języka C++</b>	<b>6</b>
<b>1 Podstawowa składnia języka</b>	<b>8</b>
1.1 Struktura programu w C++	8
1.1.1 Komentarze	10
1.1.2 Składnia języka raz jeszcze	12
1.1.3 Deklaracje i definicje	15
<b>2 Typy danych</b>	<b>17</b>
2.1 Informacje wprowadzające	17
2.1.1 Wartości i wyrażenia	17
2.1.2 Zmienne, stałe i L-wartości	18
2.1.3 Ogólny podział typów danych	19
2.2 Typy proste	20
2.2.1 Typ logiczny (bool)	20
2.2.2 Typ całkowity (stałoprzecinkowy)	20
2.2.3 Typ znakowy	22
2.2.4 Liczby rzeczywiste	23
2.2.5 Typ bez wartości (void)	24
2.2.6 Wyliczenia	24
2.2.7 Rozmiar i ograniczenia wybranych typów podstawowych	26
2.3 Definicje zmiennych i ich zasięg	27
2.3.1 Zasięg widoczności zmiennej	27
2.3.2 Czas życia zmiennej	28
2.3.3 Modyfikatory deklaracji zmiennych i stałych	30
2.4 Typy pochodne	32
2.4.1 Wskaźniki	32
2.4.2 Referencje	35
2.5 Typy złożone	37
2.6 Tablice	38
2.6.1 Napisy	42
2.6.2 Struktury	42
2.6.3 Unie	45
2.7 Jawne rzutowanie typów	46
2.7.1 Rzutowanie w stylu C	47
2.7.2 Różnego rodzaju cast-y	47

<b>3</b>	<b>Instrukcje</b>	<b>49</b>
3.1	Wyrażenia i operatory . . . . .	49
3.1.1	Priorytety operatorów . . . . .	49
3.1.2	Podział operatorów . . . . .	50
3.2	Instrukcje sterujące . . . . .	53
3.2.1	Instrukcje proste i złożone . . . . .	53
3.2.2	Instrukcja warunkowa . . . . .	54
3.2.3	Instrukcja wielokrotnego wyboru . . . . .	55
3.2.4	Przerwanie wykonania pętli . . . . .	56
3.2.5	Pętle o nieznannej liczbie iteracji . . . . .	57
3.2.6	Pętla o zadanej liczbie iteracji . . . . .	59
<b>4</b>	<b>Funkcje</b>	<b>61</b>
4.1	Funkcje . . . . .	61
4.1.1	Wiadomości podstawowe . . . . .	61
4.1.2	Przeciążanie nazw . . . . .	65
4.1.3	Wskaźniki do funkcji . . . . .	66
4.1.4	Wyrażenia lambda . . . . .	67
4.1.5	Konwencja wywołania funkcji . . . . .	69
	<b>Programowanie obiektowe</b>	<b>73</b>
<b>5</b>	<b>Programowanie obiektowe</b>	<b>75</b>
5.1	Czym jest programowanie obiektowe? . . . . .	75
5.1.1	Obiekty i klasy . . . . .	76
5.2	Deklaracje i definicje klas . . . . .	78
5.2.1	Składnia . . . . .	79
5.3	Ochrona danych w klasach . . . . .	82
5.4	Deklaracje pól . . . . .	82
5.5	Metody . . . . .	83
5.5.1	Konstruktor i destruktor . . . . .	84
5.6	Implementacja metod . . . . .	87
5.6.1	Magiczne słówko this . . . . .	88
5.7	Praca z obiektami . . . . .	88
5.7.1	Wskaźniki do obiektów . . . . .	92
<b>6</b>	<b>Programowanie zorientowane obiektowo</b>	<b>94</b>
6.1	Dziedziczenie . . . . .	94
6.1.1	Zasady dostępu do pól i metod w hierarchii . . . . .	97
6.1.2	Dziedziczenie wielokrotne . . . . .	99
6.1.3	Pułapki dziedziczenia . . . . .	99
6.2	Polimorfizm . . . . .	100
6.2.1	Abstrakcje . . . . .	103
6.2.2	Klonowanie obiektów . . . . .	104

<b>Szablony</b>	<b>106</b>
<b>7 Funkcje uogólnione</b>	<b>108</b>
7.1 Funkcje uogólnione . . . . .	108
7.1.1 Funkcje uogólnione bez szablonów . . . . .	109
7.2 Szablony funkcji . . . . .	110
7.2.1 Używanie szablonów . . . . .	113
7.2.2 Pozatypowe parametry szablonów . . . . .	116
7.2.3 Szablony parametrów szablonu . . . . .	116
<b>8 Szablony klas</b>	<b>118</b>
8.1 Typy uogólnione . . . . .	118
8.2 Szablony klas . . . . .	119
8.3 Pozatypowe parametry szablonów klas . . . . .	119
8.4 Szablony parametrów szablonu . . . . .	120
8.5 Konkretyzacja na żądanie . . . . .	121
8.6 Typy stowarzyszone . . . . .	121
<b>9 Szablony klas</b>	<b>123</b>
<b>Przykłady</b>	<b>124</b>
<b>10</b>	<b>126</b>
10.1 Aplikacje wielowątkowe . . . . .	126
10.2 Sekcja krytyczna . . . . .	126
10.3 Mutex . . . . .	127
10.4 Klasa zabezpieczona przed wielodostępem . . . . .	127
10.5 Wielowątkowa aplikacja sortująca . . . . .	129
<b>11 Programowanie gniazd sieciowych</b>	<b>135</b>
11.1 Protokół sieciowy . . . . .	135
11.1.1 Klasa protokołu . . . . .	135
11.1.2 Protokół przykładowego programu . . . . .	137
11.2 Przykładowy program klienta . . . . .	140
11.3 Przykładowy program serwera . . . . .	142
<b>Słownik</b>	<b>153</b>
<b>Bibliografia</b>	<b>153</b>



## Część I Składnia i funkcjonalność języka C++



**KAPITAŁ LUDZKI**  
NARODOWA STRATEGIA SPÓJNOŚCI

**UNIA EUROPEJSKA**  
EUROPEJSKI  
FUNDUSZ SPOŁECZNY



## Wstęp

Na początek chcielibyśmy zaprezentować Wam, w ramach częściowo przypomnienia, częściowo usystematyzowania wiedzy – składnię języka C++. W tej części będziemy zwracali głównie uwagę na formalny sposób zapisu kodu programu w języku C++, nie koncentrując się na algorytmach czy też ogólnych strukturach. Postaramy się pokazać wszystkie niuanse podstawowej składni języka, włączając w to elementy programowania niskopoziomowego, operacje na wskaźnikach, pełne i skrócone wersje poszczególnych instrukcji i operatorów. Zajmiemy się też procesem kompilacji w języku C++ - omówimy rolę i zadania preprocesora, kompilatora i linkera, oraz możliwości programowania ich zachowania.



# Rozdział 1

## Podstawowa składnia języka

Autorzy:

*Paweł Wnuk*

### Wstęp

Materiały zamieszczone w tym podręczniku powstały przy założeniu ukończenia przez Was już pierwszego kursu programowania – wprowadzenia do programowania strukturalnego, oraz częściowo – ukończenia programowania obiektowego C++. Zakładam więc, że informacje przedstawione w tym rozdziale nie są dla Was czymś nowym. Raczej są rozszerzeniem i przypomnieniem wiedzy już nabytej wcześniej. Jednak – by materiały były kompletne – nic nie będziemy pomijali.

Wiedza prezentowana w naszym podręczniku na żadnym etapie nie jest uzależniona od stosowanego przez Was kompilatora / środowiska programistycznego. Wszystkie przykłady i konstrukcje programistyczne powinny dać się skompilować i uruchomić na dowolnym standardowym kompilatorze C++ - my w trakcie naszej pracy wykorzystywaliśmy dwa z nich: CodeGear Developer Studio (wcześniej Borland Developer Studio) – jedno z najbardziej zaawansowanych komercyjnych środowisk programistycznych dostępnych dla języka C++, oraz Netbeans / GCC – połączenie darmowego środowiska z darmowym kompilatorem.

### 1.1 Struktura programu w C++

Język C++, jak już wiecie - jest dość elastyczny. Struktura programu jako takiego jest zupełnie swobodna, np. bloki instrukcji oraz definiowania zmiennych mogą się praktycznie dowolnie przeplatać ze sobą, funkcje mogą być definiowane w różnej kolejności i w różnych miejscach. Podobnie wygląda sytuacja z klasami

Schemat każdego programu w języku C++ można zapisać następująco:

### Kod programu 1.1

```
1  /* Na samym początku zazwyczaj umieszcza się pliki dołączane (nagłówki bibliotek)
2  wraz z dodatkowymi dyrektywami kompilatora. Mówimy o zwyczajowym umieszczaniu -
3  bo z punktu widzenia składni języka załączenie bibliotek może być wszędzie */
4  /** dołączenie biblioteki standardowej języka C / C++ */
5  #include <cstdlib>
6  /** dołączenie strumieniowego wejścia / wyjścia (zalecanego dla języka C++ */
7  #include <iostream>
8  /** biblioteka string zawiera implementację łańcuchów tekstowych (napisów).
9  W C++ nie ma typu prostego w pełni implementującego napisy */
10 #include <string>
11
12 /** Wykorzystanie przestrzeni nazw biblioteki standardowej. Co to jest przestrzeń
13 nazw { dowiedzie się później */
14 using namespace std;
15
16 /** Program zapisuje się w C++ w postaci funkcji. Każda funkcja zaczyna się
17 nagłówkiem, potem występuje treść zamknięta w nawiasy klamrowe. Więcej o funkcjach
18 będzie w dalszej treści podręcznika. Program może składać się z wielu funkcji. Zawsze
19 musi być co najmniej jedna - main (patrz niżej). Od niej zaczyna się tok wykonania
20 programu */
21 typ_zwracanej_wartości nazwa_funkcji(lista_parametrów)
22 {
23     ...
24 };
25
26 /** Przed lub pomiędzy funkcjami zamieszcza się definicje i deklaracje stałych,
27 zmiennych, typów i klas globalnych dla danego pliku, czyli takich, z których można
28 korzystać w każdej funkcji. */
29 const int...{definicja stałych}
30 ...
31 typedef...{definicja typów}
32 ...
33 double... {definicja zmiennych}
34 ...
35
36 /** W każdym programie C++ jest jedna główna funkcja - nazywa się main. */
37 int main(int argc, char *argv[])
38 /** Nawiasy klamrowe służą do oznaczenia początku i końca funkcji */
39 {
40     /** Kolejne instrukcje składające się na nasz algorytm */
41     instrukcja;
42     instrukcja;
43     ...
44     instrukcja;
45     /** W C++ pomiędzy instrukcjami znów mogą się znaleźć definicje
46     zmiennych, typów, stałych - lecz w takim wypadku będą one lokalne.
47     O zasięgu widoczności zmiennych będzie zamieszczona w dalszej części podręcznika.
48     */
49
50     /** Funkcja main powinna zwrócić jakąś wartość. W przypadku prawidłowego zakończenia
51     programu zwrócone powinno zostać 0 lub równoważna stała symboliczna EXIT_SUCCESS */
52     return EXIT_SUCCESS;
53     /** Zamykający nawias klamrowy na koniec funkcji main */
54 }
```

### 1.1.1 Komentarze

Tłumaczenie zaczniemy dość nietypowo – od zamieszczania komentarzy. W języku C++ mamy dwa możliwe tryby komentowania. Pierwszy z nich historycznie wywodzi się jeszcze z języka C. W tym wypadku komentarzem jest każdy fragment tekstu zaczynający się od znaków `/*` i kończący się na `*/`. Taki komentarz może zawierać wiele linii tekstu. Jeśli wewnątrz komentarza wystąpi jeszcze raz para `/*` zostanie ona zignorowana.

Drugi tryb komentarzy jest uzupełnieniem pierwszego: wszystko co zaczyna się od znaku podwójnego ukośnika `//` aż do końca linii jest traktowane jako komentarz.

Samo stosowanie komentarzy również rządzi się pewnymi regułami. My będziemy (i Wam również zalecamy) stosowali następującą konwencję komentowania:

- Komentarz zaczynający się od początku linii (bez instrukcji przed nim) będzie dotyczył tego co jest poniżej - czyli najpierw komentarz, potem kod.
- Komentarz umieszczony za instrukcją dotyczy tej instrukcji

Pamiętajcie, że nadmiar trywialnych komentarzy stanowi mniejszy błąd niż ich zupełny brak. Podkreślimy przy okazji, że komentarze służą wyłącznie osobie czytającej treść programu, i nie wpływają w żaden sposób na jego wykonanie. Kompilator (a dokładniej: preprocesor) usuwa wszelkie komentarze. Autor programu w postaci komentarzy informuje odbiorców (innych programistów, a nie użytkowników) o tym, co umieścił w programie (w tym i siebie samego ...)

W tym miejscu chcielibyśmy, byście rzucili okiem na komentarze nieco bardziej sformalizowane. Istnieje darmowy system generowania dokumentacji nazywający się [doxygen](#). Na dzień dzisiejszy to najpopularniejszy standard komentowania kodu w różnych językach programowania, oraz generowania z takiego kodu gotowej dokumentacji – łatwej do czytania i przeglądania. Sama nazwa doxygen oznacza program parsujący kod źródłowy i generujący dokumentację do niego. Jeśli kod nie jest odpowiednio skomentowany – to jedyną informacją którą możemy z niego uzyskać jest informacja o strukturze kodu – klasach, ich wzajemnych związkach, dołączanych plikach, itp. W przypadku gdy w kodzie umieścicie odpowiednio przygotowane komentarze – możliwości programu rosną w sposób znaczący. Możecie uzyskać dokumentację API (Application Programming Interface) o jakości nie odbiegającej od dokumentacji dostarczanej do komercyjnych środowisk programistycznych. I to wszystko dla Waszego kodu, w dodatku zupełnie za darmo...

Jak więc komentować kod? Zasad jest kilka. Po pierwsze – istnieją specjalne znaki komentarza, które rozpoznaje i interpretuje Doxygen. W przypadku języka C++ jest to komentarz w następującej formie:

#### Kod programu 1.2

Przykład komentarzy doxygen

```
1 /**  
2  * ... tekst (z opcjonalną gwiazdką * na początku) ...
```

```

3 */
4
5 /*!
6 * tekst (z opcjonalną gwiazdką * na początku)
7 */
8
9 ///
10 /// ... tekst ...
11 ///
12
13 ///!
14 ///! ... tekst ...
15 ///!

```

Jak widzicie – możliwości komentowania jest wiele. Wystarczy wybrać jedną. Sam Doxygen wykorzystuje dwa rodzaje opisu fragmentu kodu: krótki i szczegółowy. Zalecam zamieszczanie obu z nich – jest to możliwe bez specjalnej komplikacji:

### Kod programu 1.3

```

1 /*! \brief Tu zamieszczamy opis krótki
2 *      Dalszy ciąg krótkiego opisu
3 *
4 * Opis szczegółowy jest oddzielony od krótkiego pustą linią.
5 */
6 /// Alternatywnie możecie zamieścić opis krótki po trzech ukośnikach.
7 /** A za nim zamieścić opis szczegółowy oznaczony jako blok */

```

To nie są wszystkie możliwości programu w sensie rozróżniania rodzajów opisu. Zainteresowanych odsyłam na strony projektu. Komentarz do fragmentu kodu można umieszczać w dwóch miejscach względem komentowanego kodu: Przed komentowanym kodem, oraz za nim. Przedstawione wyżej sposoby odnoszą się do komentarzy umieszczonych przed komentowanym kodem. Aby Doxygen zrozumiał komentarz umieszczony za komentowanym kodem należy użyć znaku mniejszości przed komentarzem, tak jak w przykładzie poniżej (ale lepiej tej techniki nie stosować):

### Kod programu 1.4

```

1 int zmienna; /*!< To jest krótki opis zmiennej */

```

W przypadku funkcji, oprócz komentarzy krótkich (ogólnych) i szczegółowych, warto również poświęcić kilka minut na udokumentowanie argumentów wejściowych i wyjściowych oraz zwracanych wartości. Poniżej przykład poprawnie udokumentowanej funkcji z wykorzystaniem komend specjalnych (`\param` oraz `\return`):

### Kod programu 1.5

```

1 /**
2 * Funkcja sprawdza, czy z trzech odcinków da się zbudować trójkąt.
3 * Pobiera trzy wartości typu int i zwraca wartość typu bool.
4 *
5 * \param[in] x długość pierwszego odcinka.
6 * \param[in] y długość drugiego odcinka.

```

```

7  * \param[in] z długość trzeciego odcinka.
8  * \return true jeśli da się zbudować trójkąt, false w przeciwnym wypadku
9  */
10 bool triangle(int x, int y, int z) {
11     return (x < y+z) && (y < x+z) && (z < x+y);
12 }

```

Atrybut `[in]` umieszczony po komendzie `\param` jest atrybutem opcjonalnym, wskazującym że komentowany argument jest argumentem wejściowym funkcji (dostarcza danych do funkcji). Jeśli na liście argumentów znajduje się argument przekazywany przez referencję lub za pomocą wskaźników (jest pobierany i zmieniany w trakcie działania funkcji), należy użyć atrybutu `[in,out]`. Jeśli argument nie wprowadza żadnych danych do funkcji, a jedynie funkcja zwraca wartość za pomocą argumentu, należy użyć atrybutu `[out]`.

Doxygen posiada zdefiniowanych jeszcze wiele komend specjalnych (takich jak `\param`), można także oznaczać je na różne sposoby (np. `@param` też jest dopuszczalne). Więcej informacji znajdziecie na stronie projektu.



W tym podręczniku także będę wykorzystywał składnię doxygena do komentarzy. Podobnie powinno być w Waszym przypadku – wymogiem zaliczenia projektu powiązanego z tym przedmiotem jest prawidłowa dokumentacja kodu który powstaje.

### 1.1.2 Składnia języka raz jeszcze

W języku C++ formalizm zapisu jest stosunkowo prosty i ograniczony. Jednakże elegancja obowiązuje zawsze – tym bardziej, że w większości współczesnych środowisk i edytorów programistycznych możecie swobodnie korzystać z autoformatowania. Przypominam zestaw dobrych rad odnośnie formatowania:

1. Każda instrukcja powinna być zapisana w oddzielnej linii,
2. Wszystko to, co znajduje się pomiędzy nawiasami klamrowymi `{ i }` (blok programu), powinno zostać przesunięte względem nich o 2-3 spacje,
3. Zmienne deklarujemy kolejno, w oddzielnych liniijkach umieszczając oddzielne deklaracje / definicje.
4. Każda zmienna powinna być opisana za pomocą komentarza. Podobnie podstawowe kroki algorytmu.



Wspomniane wyżej 4 zasady formatowania nie są ani standardem, ani kompletnym sposobem opisu. Właściwie każda grupa programistów wypracowuje własny styl formatowania, nazywania zmiennych i funkcji, nazywania plików, itp. Ważniejsze od konkretnych reguł w takim stylu jest jego konsekwentne stosowanie. Poniżej zamieszczamy dla Was przykłady trzech najbardziej popularnych stylów formatowania kodu, przy czym w niniejszym podręczniku będziemy stosować standard Kernighan z modyfikacjami naszymi ;)



Formatowanie kodu – standard ANSI

### Kod programu 1.6

```
1 namespace foospace
2 {
3     class Bar
4     {
5     public:
6         int foo();
7     private:
8         int foo_2();
9     };
10
11     int Bar::foo()
12     {
13         switch (x)
14         {
15             case 1:
16                 a++;
17                 break;
18             default:
19                 break;
20         }
21         if (isBar)
22         {
23             bar();
24             return m_foo+1;
25         }
26         else
27             return 0;
28     }
29 }
```

Formatowanie kodu – standard Kernighan - Ritchie

### Kod programu 1.7

```
1 namespace foospace {
2     class Bar {
3     public:
4         int foo();
5     private:
6         int foo_2();
7     };
8
9     int Bar::foo() {
10         switch (x) {
11             case 1:
12                 a++;
13                 break;
14             default: break;
15         }
16         if (isBar) {
17             bar();
18             return m_foo+1;
19         } else
20             return 0;
21     }
22 }
```



## Formatowanie kodu – standard GNU

### Kod programu 1.8

```
1 namespace foospace
2 {
3     class Bar
4     {
5     public:
6         int foo();
7     private:
8         int foo_2();
9     };
10
11     int Bar::foo()
12     {
13         switch (x)
14         {
15             case 1:
16                 a++;
17                 break;
18             default:
19                 break;
20         }
21         if (isBar)
22         {
23             bar();
24             return m_foo+1;
25         }
26         else
27             return 0;
28     }
29 }
```

W dużym skrócie przypomnę – że w języku C++ mamy do czynienia ze słowami kluczowymi (jest ich określona ilość i nie można ich zmieniać), oraz typami, zmiennymi, stałymi, itp – definiowanymi przez użytkownika. Zobaczcie na trywialny przykład:

### Kod programu 1.9

```
1 x = y + f(2);
```

W C++ by to miało sens – x, y i f muszą być odpowiednio zadeklarowane (by stały się bytami o swoich nazwach). Z każdą nazwą (identyfikatorem) jest związany typ, który określa jakie operacje można wykonać na jego przedstawicielu. Każdy taki byt musi być identyfikowalny – vide posiadać identyfikator. Identyfikator w C++ definiuje się następująco:

### Definicja 1.1

Identyfikator

Przykłady poprawnych i niepoprawnych identyfikatorów:

### Kod programu 1.10





## Identyfikatory

```
1 // poprawne zmienne:
2 int  Aaa, aAa, aaa;
3 double _kot, mi29, Moja1B_;
4
5 // niepoprawnie
6 char ala ma kota;
7 bool 39A;
8 double new;
9 float $inna_zmienna;
10
11 // definicja zmiennej
12 double zmienna;
13
14 // deklaracja
15 extern double inna;
```



Nie będę tutaj powtarzał znanych już Wam z poprzednich zajęć dodatkowych informacji o słowach kluczowych, znakach przestankowych, itp – zainteresowani niech sięgną do odpowiednich materiałów. Tu natomiast zatrzymamy się jeszcze przez chwilę przy deklaracjach i definicjach.

### 1.1.3 Deklaracje i definicje

Z każdą nazwą (identyfikatorem) jest związany typ, który określa jakie operacje można wykonać na jego przedstawicielu. Typy są różne – i nie mówimy tu tylko o typach danych ale o każdym elemencie języka. Przykładowe operacje które można wykonać na:

- stałych – można odczytać ich wartość
- zmiennych – można odczytać i zapisać
- funkcjach – można wykonać.
- klasach, szablonach, przestrzeniach nazw – o tym powiemy w drugiej części podręcznika.

#### Definicja 1.2

Związywanie nazwy (identyfikatora) z jej typem będziemy nazywali **deklaracją**

Deklaracja nie oznacza przyznania pamięci dla zmiennej, czy podania kodu dla funkcji – jest to jedynie informacja składniowa. W ten sposób programista może „obiecać” kompilatorowi, że gdzieś tam znajdzie się definicja zmiennej czy funkcji. Kompilator musi przyjąć deklarację programisty, i wg deklaracji sprawdzana jest poprawność składniowa kodu.

Z pojęciem deklaracji ściśle powiązane jest pojęcie **definicji**:

#### Definicja 1.3



Żądanie przyznania pamięci dla zmiennych, podanie kodu dla funkcji czy podanie opisu dla typów własnych będziemy nazywali definicją.

Innymi słowy – definicją są wszystkie informacje niezbędne do wygenerowania kodu wykonawczego programu. Z tego wynika zależność pomiędzy deklaracją a definicją: każda definicja jest jednocześnie deklaracją (każdy program który można prawidłowo skompilować i uruchomić jest poprawny składniowo), natomiast nie każda deklaracja jest definicją (nie każdy program poprawny składniowo można skompilować i uruchomić).

W języku C++ wszystko co ma nadaną nazwę (stałe, zmienne, funkcje, itp) musi mieć typ, przy czym dla każdej nazwy musi istnieć tylko jedna definicja, natomiast może istnieć wiele takich samych deklaracji. Dlatego też działa mechanizm dołączania plików nagłówkowych (ale o tym za chwilę). Kilka przykładów deklaracji i definicji:

### Kod programu 1.11

```
1 char znak; // definicja zmiennej typu podstawowego
2 string s; // definicja zmiennej typu zdefiniowanego w bibl. standardowych
3 int licznik = 1; // definicja wraz z inicjacją
4 const double pi = 3.14; // definicja stałej (musi być z inicjacją)
5 extern long pid; // deklaracja zmiennej
6
7 char *imie = "Alicja"; // definicja tablicy znakow
8 // definicja tablicy tablic.
9 char *pora[] = {"wiosna",
10               "lato",
11               "jesien",
12               "zima"}
13
14 // wiele deklaracji { jedna definicja
15 extern int licznik;
16 int licznik;
17 extern int licznik;
18
19 // poniższy kod jest błędny
20 // licznik był już zadeklarowany / zdefiniowany jako int
21 extern double licznik;
22 // podobnie tutaj: licznik był już zadeklarowany / zdefiniowany jako int
23 double licznik;
24 // mimo że dwie definicje są identyczne { definicji nie można powtarzać.
25 int licznik;
26
27 // definicja dwóch zmiennych tego samego typu
28 int x, y;
29 // definicja mieszana { z inicjacją i bez niej.
30 int a1 = 2, b1;
31
32 // definicja wskaźnika
33 long int *pole = NULL;
34 // definicja mieszana { jedna zmienna to wskaźnik, druga zmienna jest statyczna.
35 int* p2, p3;
```

# Rozdział 2

## Typy danych

Autorzy:

*Paweł Wnuk*

### 2.1 Informacje wprowadzające

Język C++ należy do grupy języków programowania z silną kontrolą typów, co dla Was w praktyce oznacza, że kontrola typów w C++ odbywa się na etapie kompilacji (tylko pewne elementy tzw. późnego łączenia - sprawdzania typu zmiennej w trakcie wykonania programu - odbywają się w fazie wykonania programu, i tylko dla typów należących do jednej hierarchii klas). Dotyczy to wszystkich znanych technik programowania w C++ - także programowania generycznego (szablonów). Poniżej znajdziecie krótkie przypomnienie dostępnych typów danych:

#### 2.1.1 Wartości i wyrażenia

Zanim przejdziemy do dokładnego opisywania typów pora na jeszcze jedną uwagę, która nie jest oczywista na pierwszy rzut oka, oraz jest cechą charakterystyczną C / C++, lecz już niekoniecznie w przypadku innych języków programowania.



W C++ prawie wszystko jest wyrażeniem które zwraca jakąś wartość.

Nie to końca jest to zawsze zrozumiałe na pierwszy rzut oka. O ile dla prawie każdego jest jasne, że wykonanie operacji dodawania  $X+Y+15$  zwraca wynik, podobnie jak wywołanie funkcji  $\text{SIN}(X)$ , o tyle nie wszyscy od razu wiedzą, że również wykonanie przypisania  $Z = X$  zwraca wartość, podobnie jak przykładowo ... drukowanie tekstu na ekranie `COUT << "WITAJ";` ... i wiele, wiele innych. W C++ nie istnieje możliwość zdefiniowania podprogramu czy operatora który nie zwraca jakiegokolwiek wartości, natomiast możliwe są dwie sytuacje:

- zwracana wartość może być tzw. wartością która oznacza brak wartości ( **void** ),
- zwracaną wartość można pominąć (nie „przechwycić” jej) – co jest niemożliwe w wielu innych językach. Dlatego poprawne jest napisanie po prostu instrukcji `2+2;` (pomijając kompletny brak jej sensu).

Skoro wiadomo, że prawie wszystko w C++ ma wartość, i wartość ta może być odrzucona, to co się dzieje z wartościami które nie mają być są odrzucone?

### 2.1.2 Zmienne, stałe i L-wartości

To, co jest charakterystyczne zarówno dla zmiennych, jak i stałych występujących w C++, to wspomniana na początku silna kontrola typów. W praktyce oznacza to, że zmienna czy stała musi mieć raz przypisany w momencie definicji, i potem do końca jej czasu życia niezmienny – typ danych które może przechowywać. Typ jest niezmienny – lecz od tego czy może się zmieniać wartość, czy nie – zależy czy mamy do czynienia ze zmienną czy ze stałą. W składni C++ to rozróżnienie jest zaznaczone, lecz często nie do końca poważnie brane przez kompilator.

Ogólnie stałe definiuje się i deklaruje tak jak zmienne – jedyną różnicą jest dodanie przedrostka **const** przed wyspecyfikowaniem typu. Podanie **const** w założeniu uniemożliwia zmianę wartości tak oznaczonej zmiennej w zasięgu jej widoczności bez jawnego rzutowania **const\_cast**. No właśnie ... bez jawnego rzutowania ... to co to za stała którą można zmienić w zmienną? Dlatego wspomnieliśmy że **const** nie jest brany poważnie przez kompilator. Jeśli włączona jest optymalizacja, kompilator sprawdza, czy istnieje w programie choćby teoretyczna możliwość zmiany wartości oznaczonej jako stała – i jeśli tak, to ignoruje przyrostek **const** tworząc zamiast tego zmienną, której wartości nie można prosto zmienić. W przeciwnym wypadku często wartości stałych są bezpośrednio rozwijane w kodzie programu, i nie jest im w ogóle przydzielana pamięć z obszaru pamięci danych.

Z pojęciami zmiennej i stałej ściśle związane jest pojęcie L-wartości. Ogólnie można przyjąć, że:

#### Definicja 2.1

L-wartością możemy nazwać wszystko co może stać po lewej stronie przypisania – czemu można przypisać wartość.

W praktyce L-wartościami najczęściej są zmienne bez modyfikatora **const**, ale także – np. parametry funkcji z tym modyfikatorem. Od powyższej definicji jest wyjątek – w przypadku definiowania stałej połączonego z jej inicjacją, stała stoi po lewej stronie równania – lecz nie jest ona L-wartością ... wartość jest wyliczana na etapie kompilacji, a operator przypisania jest wtedy traktowany jako operator inicjacji.

Na razie parametrami funkcji i innymi skomplikowanymi zagadnieniami nie zajmujemy się, natomiast sama definicja L-wartości mówi nam, dlaczego poprawny jest poniższy kod:

### Kod programu 2.1

```
1 double r = 12.3;  
2 double x = 2.0*M_PI*r;
```

x jest zmienną – więc jest L-wartością, natomiast niepoprawny jest ten kod:

### Kod programu 2.2

```
1 double r = 12.3;  
2 double x;  
3 2.0*M_PI*r = x;
```

Wartość wyrażenia  $2 * M\_PI * r$  nie jest L-wartością.

## 2.1.3 Ogólny podział typów danych

W pierwszym przybliżeniu wszelkie dostępne typy danych w języku C++ można podzielić na następujące grupy:

- Typy podstawowe. Wśród nich wyróżniamy typy ściśle powiązane z architekturą sprzętową komputera (logiczne, znakowe, całkowite, zmiennoprzecinkowe), typ wyliczeniowy i typ oznaczający brak wartości
- Typy pochodne dla typów podstawowych, czyli funkcje, wskaźniki i referencje
- Typy złożone, czyli tablice, struktury i klasy.

Wspomniany podział nie jest jedynym możliwym, wśród wspomnianych typów można wydzielić typy przeliczalne:

### Definicja 2.2

Typem przeliczalnym (porządkowym) nazywamy każdy typ, który charakteryzuje się skończonym, uporządkowanym zbiorem wartości, w którym można wyznaczyć wartość największą i najmniejszą oraz dla którego dla każdej wartości z wyjątkiem wartości brzegowych możemy jawnie wskazać wartość następną i poprzednią.

Do typów przeliczalnych zaliczamy typ logiczny, znakowy, całkowitoliczbowy i wyliczeniowy. Wyróżnienie typów przeliczalnych jest istotne z tego względu, że w niektórych instrukcjach (np. `switch`) zmienna sterująca musi być przeliczalna. Istnieje także wydzielona grupa typów arytmetycznych:

### Definicja 2.3

Typem arytmetycznym nazwiemy każdy typ, dla którego da się w sposób naturalny stosować operatory arytmetyczne.

Tutaj zaliczymy wszystkie typy liczbowe (logiczny, całkowitoliczbowy i zmiennoprzecinkowy) oraz ... typ znakowy czy wskaźniki.

## 2.2 Typy proste

### 2.2.1 Typ logiczny (`bool`)

Nazwą typu logicznego jest **bool**. Typ może przechowywać tylko dwie wartości: fałsz (**false**) i prawda (**true**) uporządkowane w tej właśnie kolejności. Jest to typ arytmetyczny przeliczalny.

Typ logiczny został wprowadzony jako nowy w języku C++ - w starym C jako logiczny fałsz przyjmowano wartość zero, jako prawdę wszystkie inne wartości. I tak też dokonywane jest rzutowanie w języku C++. W przypadku rzutowania zmiennej typu **bool** na typ liczbowy fałsz zostanie przedstawiony jako 0, natomiast prawda jako 1.

Fakt przechowywania tylko dwóch wartości sugerowałby wyjątkową oszczędność pamięci przy stosowaniu zmiennych tego typu – tak niestety jednak nie jest. Każda zmienna logiczna zajmuje w pamięci komputera co najmniej 1 bajt (a nie bit) – wynika to ze względów wydajności. W zdecydowanej większości architektur sprzętowych najmniejszą możliwą jednostką pamięci możliwą do przesłania między RAM a procesorem jest właśnie bajt.

### 2.2.2 Typ całkowity (stałoprzecinkowy)

Nazwą typu stałoprzecinkowego jest `int`. Typ **int** może przechowywać liczby całkowite (inaczej: stałoprzecinkowe) z pewnego określonego przedziału zależnego od połączenia procesora, systemu operacyjnego oraz kompilatora który wykorzystujecie.

W przypadku większości kompilatorów dla Windows, jeśli zdefiniujecie zmienną jako całkowitą, kompilator założy, że jest to 32-bitowa liczba ze znakiem. Zatem będziecie w stanie przechowywać w niej wartości z przedziału od  $-2^{31}$  do  $2^{31}-1$ , czyli od -2147483648 do 2147483647. Trochę ciężko zapamiętać, nie? Prościej skorzystać z faktu, że rzeczywista wartość maksymalna danego typu jest praktycznie zawsze dostępna przez odpowiednie makrodefinicje. W przypadku **int** wartość minimalna i maksymalna są dostępne pod nazwami odpowiednio `INT_MIN` i `INT_MAX` (zobacz program ).

Typ **int** występuje w kilku wariacjach, różniących się zakresem i faktem posiadania znaku lub nie. Dwie podstawowe modyfikacje to żądanie zmniejszenia ilości bitów i zakresu, czyli **short**, oraz żądanie zwiększenia liczby bitów i zakresu, czyli **long**. Drugi modyfikator to

oznaczenie sposobu interpretacji najstarszego bitu w liczbie, czyli **signed** - oznacza że najstarszy bit oznacza znak liczby (innymi słowy – można przechowywać zarówno liczby dodatnie jak i ujemne) oraz **unsigned** - oznacza, że najstarszy bit wchodzi w skład liczby, i nie ma możliwości przechowywania wartości ujemnych.

Jeśli nie podacie żadnego modyfikatora, C++ zakłada że typ jest typem zwykłym ze znakiem (czyli **int** jest równoważne **signed int**). Jeśli podacie tylko modyfikator typu, czyli **short**, **long**, **signed** lub **unsigned** – kompilator założy że zmienna będzie typu **int**. Wystarczy pisać np. **long** i będzie to oznaczało **long int**. Podobnie **unsigned** będzie oznaczać **unsigned int**.

Modyfikatory wielkości zmiennej **short** i **long** przez standard C++ są traktowane jako wyrażenie woli programisty, i nigdzie nie jest powiedziane, że za każdym razem w zmiennej typu **long int** da się przechować większą wartość niż w zmiennej typu **int** – zdziwicie się, lecz w większości przypadków te typy są sobie kompletnie równoważne. Jedyne wymóg nakładany przez standard języka można w skrócie zapisać w następujący sposób:

$$1 = \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$$

co oznacza naszymi słowami, że każdy następny wielkościowo typ ma być nie mniejszy niż bezpośrednio go poprzedzający. Jak się można było spodziewać, tworzy to niezły misz-masz. Dlatego też twórcy kompilatorów zwyczajowo zamieszczają makrodefinicje typów jawnie pokazujące jaką długość bitową mają, czyli **\_\_int16** oznacza szesnastobitową liczbę, a **\_\_int64** liczbę 64-bitową.

Do wykonywania obliczeń polecamy Wam stosowanie typów ze znakiem – podejście w stylu „*zmienna x nie powinna przyjmować wartości ujemnej więc zdefiniuję ją jako unsigned int*” może być przyczyną poważnych błędów – po przypisaniu do takiej zmiennej wartości ujemnej otrzymamy ... brak błędu i bardzo dużą liczbę dodatnią. Rozsądnym stosowaniem typów bez znaku jest wykorzystywanie ich do indeksowania, a i to przy założeniu zachowania konsekwencji.

Oprócz samego definiowania zmiennych i stałych, istnieje także wiele sposobów zapisu literałów stałoprzecinkowych (wartości) w kodzie programu. Wartości możemy podawać dziesiętnie, ósemkowo lub szesnastkowo, wymuszając jednocześnie traktowanie liczby jako liczby ze znakiem lub bez. Liczby dziesiętne piszemy „normalnie”, liczby ósemkowe poprzedzamy cyfrą 0, a szesnastkowe parą znaków 0x. O ile z zapisem szesnastkowym nie ma problemów, to powinniście uważać na zapis ósemkowy:

### Kod programu 2.3

```
1 int x = 15;
2 int y = 015;
3 if (x == y)
4     cout << "Tego sie spodziewamy";
5 else
6     cout << "a to jest";
```

15 i 015 to różne liczby!

Zasady zapisu zawiera tabela poniżej:



**Tabela 2.1** – Literały liczbowe stałoprzecinkowe.

Zapis dziesiętny	0	2	83
Zapis ósemkowy	00	02	0123
Zapis szesnastkowy	0x0	0x2	0x53

Dodatkowo, przy zapisie literałów możemy wymusić ich traktowanie jako liczby bez znaku (dodając U na końcu) lub jako liczby długiej (dodając L na końcu).

Przykładowe definicje zmiennych i stałych stałoprzecinkowych:

#### Kod programu 2.4

```

1 // zmienna całkowitoliczbowa ze znakiem
2 int a;
3 // zapis równoważny
4 signed int b;
5 // deklaracja zmiennej powiększonej:
6 extern long c;
7 // krótka liczba bez znaku
8 unsigned short d;
9 // inicjacja zmiennej liczbą w zapisie szesnastkowym
10 int e = 0x0EF;
11 // inicjacja zmiennej liczbą w zapisie ósemkowym z wymuszeniem braku znaku
12 unsigned long f = 0x12U;

```

### 2.2.3 Typ znakowy

W C++ występują dwa typy znakowe: **char** przeznaczony do trzymania znaków w kodowaniu ASCII (jednobajtowych), oraz **wchar\_t** przeznaczony do trzymania znaków w kodowaniu UTF16 (dwubajtowy). Typ znakowy jest typem przeliczalnym i arytmetycznym.

W rzeczywistości C++ nie analizuje znaków w żaden sposób – przechowuje je, i operuje na nich tak jak na liczbach całkowitych. Dlatego też jest to typ arytmetyczny, i dlatego znaki możecie do siebie dodawać. Wartości znaków mogą być podawane jako literały znakowe (w pojedynczych apostrofach), albo bezpośrednio jako kody znaków. W przypadku podawania znaków jako literałów należy pamiętać o tym, że znak odwróconego ukośnika ma znaczenie specjalne, i służy do podawania kodów sterujących:

**Tabela 2.2** – Wybrane literały znakowe dla kodów sterujących

\n	LF - przejście do następnej linii ("wysunięcie linii")
\r	CR - skok na początek linii
\t	HT - tabulacja pozioma
\v	VT - tabulacja pionowa
\a	BL - sygnalizacja (tzw. BELL)
\f	FF - wysunięcie strony w drukarce (w *nix-ach powoduje skasowanie ekranu)
\xNN	znak o kodzie szesnastkowym 0xNN, gdzie N – cyfra szesnastkowa

<code>\DDD</code>	znak o kodzie dziesiętnym DDD, gdzie D – cyfra dziesiętna.
<code>\\</code>	Znak <code>\</code>
<code>\'</code>	Znak <code>'</code>
<code>\"</code>	znak <code>"</code>

Typ znakowy można wykorzystywać również do operacji na małych liczbach całkowitych.

Pamiętajcie również, że istnieją w C++ dwa typy znakowe: **signed char** i **unsigned char**. Jest również typ **char** (bez modyfikatora) i jest on równoważny albo jednemu, albo drugiemu z nich (standard nie precyzuje, któremu) w związku z czym, nie należy zakładać nigdy sposobu, w jaki w danym kompilatorze wartości z zakresu -128 do -1 czy 128 do 255 będą traktowane przez typ **char**. W przypadku, gdy chce się używać zakresów typu **char** poza 0-127 należy jawnie określać **char** jako **signed** lub **unsigned**.

### Kod programu 2.5

```
1 char z1 = 'a'
2 char z2= '\t'
3 char z3 = 48;
4
5 wchar_t wz = L'ab';
```

## 2.2.4 Liczby rzeczywiste

Typ liczb rzeczywistych występuje (podobnie jak **int**) w kilku wersjach różniących się wielkością i zakresem wartości: **float**, **double** i **long double**. Jest typem arytmetycznym, lecz nie jest typem przeliczalnym.

Najmniejszy z typów rzeczywistych, **float**, nie powinien być przez Was traktowany jako podstawowy typ zmiennoprzecinkowy (mimo że wiele podręczników ciągle traktuje go w ten sposób). Najczęściej **float** posiada tylko 6 (sic!) cyfr znaczących - wszelkie obliczenia na takich wartościach obarczone są ogromną niedokładnością wynikłą z konieczności przybliżania. Dla porównania – **double** zazwyczaj ma 15-16 cyfr znaczących.

Literały stałoprzecinkowe można wprowadzać również na kilka sposobów, które pokażemy na przykładzie liczby 123.4567:

**Tabela 2.3** – Literały zmiennoprzecinkowe

Zapis	Typ
123.4567	double
123.4567F lub 123.4567f	float
123.4567L lub 123.4567l	long double
1.234567e2 lub 123.4567E2	double

Wewnętrznie zmienna rzeczywista jest pamiętana w postaci dwu członów: podstawy *a* i wykładnika *b* i jest równa  $a * 10^b$  (*a* razy 10 do potęgi *b*), przy czym zarówno *a* jak i *b* muszą mieścić się w pewnym przedziale. Z tego faktu wynikają dwa ograniczenia - liczba bitów przeznaczona na pamiętanie podstawy *a* określa nam maksymalną możliwą precy-



zję zapamiętania liczby (ilość miejsc po przecinku), liczba bitów, jaka jest przeznaczona na pamiętanie `b` definiuje natomiast zakres zmienności zmiennej. Mówiąc inaczej, możecie zapamiętać dokładnie liczbę 0.0000000000000001 oraz 1000000000000000, natomiast nie można zapamiętać 1000000000000000. 0000000000000001 - część ułamkowa zostanie pominięta w tym przypadku. Co gorsza, jeśli dodacie te dwie liczby do siebie, w wyniku otrzymacie pierwszą z nich, a o spowodowanej poprzez zaokrąglenie niedokładności nie zostaniecie nawet poinformowani.

Przykłady definicji zmiennych rzeczywistych:

### Kod programu 2.6

```
1 double da = 1.23;  
2 double db = .23;  
3 double dc = 1.;  
4 double dd = 1.2e-12;
```

Wartość minimalna i maksymalna liczb zmiennoprzecinkowych zazwyczaj nie jest definiowana tak, jak to miało miejsce w przypadku typów prostych. W zamian za to można uzyskać do niej dostęp poprzez szablon `numeric_limits` w sposób pokazany w przykładzie .

## 2.2.5 Typ bez wartości (void)

Ostatnim z typów prostych które występują w języku C++ jest **void** – typ oznaczający brak wartości. Nie jest to typ ani arytmetyczny, ani przeliczalny, co więcej – nie można stworzyć zmiennej ani stałej typu **void**. Jego główne zastosowania to albo oznaczenie że funkcja wykorzystana jako wyrażenie nie zwraca żadnej wartości, oraz do rzutowania wskaźników. Oba przypadki zostaną dokładnie wyjaśnione w dalszej części podręcznika.

## 2.2.6 Wyliczenia

Wyliczenia w C++ są najczęściej wewnętrznie pamiętane jako jedna z odmian liczb całkowitych. Nie są typem stricte podstawowym, bo wymagają wcześniejszej definicji typu (przed pierwszym użyciem), natomiast też nie są typem użytkownika (nie można definiować w pełni ich zachowania). My zamieszczamy je wśród typów podstawowych.

Wyliczenie definiuje się przy wykorzystaniu słowa kluczowego **enum**, i są typem przeliczalnym, ale uwaga – nie zalicza się ich do typów arytmetycznych. Ogólnie wyliczenia są przewidziane do przechowywania ściśle określonego, zdefiniowanego przez użytkownika zbioru wartości. Przy czym trzeba pamiętać, że każde wyliczenie jest oddzielnym typem, który może – ale nie musi – mieć nazwę.

Domyślnie C++ przypisuje wartości liczbowe nazwom elementów wyliczenia kolejno, poczynając od zera i z krokiem 1, lecz programista może jawnie podać wartości numeryczne przypisywane stałym symbolicznym.



Rysunek 2.1 – Schemat definicji wyliczenia

Istnieje możliwość przekształcenia wyliczenia na liczbę całkowitą i odwrotnie, lecz bez kontroli zakresu - wynik przekształcenia stałej liczbowej spoza zakresu jest niezdefiniowany.

Przykłady definicji i wykorzystania wyliczeń:

#### Kod programu 2.7

```

1 // wyliczenie nienazwane
2 enum {ala, kot, dwa_koty };
3 // wyliczenie nazwane
4 enum asta {la, vista };
5
6 // wyliczenie z określonym zakresem
7 enum eee {aaa = 3, zzz = 9 };
8
9 eee zmienna = eee(3); // ok
10 eee zmienna = eee(101); // źle
    
```

## 2.2.7 Rozmiar i ograniczenia wybranych typów podstawowych

Poniższy program wyświetli Wam informację o wszystkich podstawowych typach arytmetycznych przeliczalnych na Waszej platformie.

### Kod programu 2.8

```

1  #include <iostream>
2  #include <climits>
3
4  using namespace std;
5
6  volatile int char_min = CHAR_MIN;
7
8  int main(void)
9  {
10     cout << "Rozmiar typu bool: " << sizeof(bool) << " bajtow\n";
11     cout << "Liczba bitow do pamietania znaku: " << CHAR_BIT << '\n';
12     cout << "Rozmiar typu char: " << sizeof(char) << " bajtow\n";
13     cout << "Wartosci dla signed char: min: " << SCHAR_MIN << " max: " << SCHAR_MAX << '\n';
14     cout << "Wartosci dla unsigned char min: 0 max: " << UCHAR_MAX << '\n';
15     cout << "Domyslnym typem dla znakow jest ";
16     if (char_min < 0)
17         cout << "signed";
18     else if (char_min == 0)
19         cout << "unsigned";
20     else
21         cout << " ? dziwny jakis";
22     cout << "\n\n";
23
24     cout << "Rozmiar typu short int: " << sizeof(short) << " bajtow\n";
25     cout << "Wartosci dla signed short: min: " << SHRT_MIN << " max: " << SHRT_MAX << '\n';
26     cout << "Wartosci dla unsigned short min: 0 max: " << USHRT_MAX << "\n\n";
27
28     cout << "Rozmiar typu int: " << sizeof(int) << " bajtow\n";
29     cout << "Wartosci dla signed int: min: " << INT_MIN << " max: " << INT_MAX << '\n';
30     cout << "Wartosci dla nsigned int: min: 0 max: " << UINT_MAX << "\n\n";
31
32     cout << "Rozmiar typu long int: " << sizeof(long) << " bajtow\n";
33     cout << "Wartosci dla signed long: min: " << LONG_MIN << " max: " << LONG_MAX << '\n';
34     cout << "Wartosci dla unsigned long: min: 0 max: " << ULONG_MAX << endl;
35
36     cout << "Rozmiar typu float" << sizeof(float) << '\n';
37     cout << "Wartosci dla float: min: " << numeric_limits<float>::min();
38     cout << " max: " << numeric_limits<float>::max() << "\n";
39
40     cout << "Rozmiar typu float" << sizeof(float) << '\n';
41     cout << "Wartosci dla float: min: " << numeric_limits<float>::min();
42     cout << " max: " << numeric_limits<float>::max() << "\n";
43
44     cout << "Rozmiar typu double" << sizeof(double) << '\n';
45     cout << "Wartosci dla double: min: " << numeric_limits<double>::min();
46     cout << " max: " << numeric_limits<double>::max() << "\n";
47
48     cout << "Rozmiar typu long double" << sizeof(long double) << '\n';
49     cout << "Wartosci dla long double: min: " << numeric_limits<long double>::min();
50     cout << " max: " << numeric_limits<long double>::max() << "\n";
51

```

```
52 system("PAUSE");  
53 return EXIT_SUCCESS;  
54 }
```

## 2.3 Definicje zmiennych i ich zasięg

Zanim przejdziemy do opisywania typów pochodnych do podstawowych chcielibyśmy powiedzieć coś więcej o deklaracjach i definicjach zmiennych. Same pojęcia wprowadziliśmy w poprzednim rozdziale (tu link) – teraz pora na rozszerzenie podanych informacji w kontekście zmiennych.

Każda zmienna wymaga przydzielenia obszaru pamięci odpowiedniego dla niej. W C++ cała dostępna pamięć jest dzielona na kilka niezależnych obszarów (klas pamięci, ang. *storage class*). Obszar przeznaczony na zmienne lokalne zwyczajowo nazywa się stosem. Sam stos jest podzielony na dwie klasy pamięci: na zmienne globalne i lokalne. Każdy obiekt lokalny jest dostępny wyłącznie w kontekście, w którym został zadeklarowany, zaś jego czas życia jest od wejścia do kontekstu do wyjścia z niego. Globalny zaś, deklarowany poza wszystkimi funkcjami, jest dostępny dla wszystkich funkcji. Jeszcze jedna różnica dotyczy inicjacji: Zmienne typu prostego z kontekstu lokalnego nie są inicjowane wcale, natomiast pamięć przeznaczona na zmienne globalne powinna być zerowana (nie wszystkie kompilatory to realizują).

Nie zainicjalizowana zmienna posiada wartość taką, jaka się jej trafiła w przeznaczonym dla niej kawałku pamięci. Ważną informacją jest fakt, że wyniki wszystkich operacji na takich wartościach (z wyjątkiem przypisania) są niezdefiniowane. Wartość taką nazywamy wartością osobiłą (ang. *singular*). Wartość osobiła to po prostu taka wartość o której nie tylko nic nie wiemy, ale też nad którą program nie ma żadnej kontroli; innymi słowy, jest to wartość, której nikt nie nadał. Wykonywanie jakichkolwiek operacji poza przypisaniem na niezainicjowanych zmiennych może być wyjątek, i związane z nim awaryjne zakończenie programu.

### 2.3.1 Zasięg widoczności zmiennej

Chwilę wcześniej wspomnieliśmy o „dostępności zmiennej w kontekście” ... hem ... a co to znaczy?

W C++ każda nazwa może być wykorzystywana jedynie w tej części programu, gdzie jest znana. Wszędzie znane są jedynie nazwy (identyfikatory) zdefiniowane w przestrzeni globalnej, przy zastrzeżeniu obowiązywania zasady predeklaracji.

Ogólne zasady widoczności można streścić następująco:

- nazwy globalne są widoczne od miejsca deklaracji do końca pliku. Globalnie widoczne są nazwy deklarowane poza funkcją, klasą i przestrzenią nazw.
- nazwy lokalne są widoczne wewnątrz bloku { }
- nazwy z przestrzeni nazw są widoczne wewnątrz tej przestrzeni

- nazwy należące do klasy są widoczne wewnątrz klasy

Kompilator C++ czyta plik z kodem od góry do dołu. Każdą napotkaną nazwę (identyfikator) próbuje rozszyfrować korzystając kolejno z lokalnej przestrzeni nazw, następnie z klasy, klas podstawowych dla danej klasy, przestrzeni nazw bieżących i podstawowych, oraz przestrzeni globalnej – i przestaje szukać po znalezieniu pierwszego dopasowania.

Z takiego cyklu działania wynika kolejny mechanizm: przysłaniania nazw. Każda nazwa może być wykorzystana tylko raz – ale w jednej przestrzeni nazw. Postawienie nawiasów klamrowych otwiera nową przestrzeń nazw. Popatrzcie na poniższy kod:

### Kod programu 2.9

```
1 int x; // x zasięg nazw globalny
2
3 int main(int argc, char *argv[])
4 {
5     // x w zasięgu nazw funkcji main { inna zmienna niż globalne x
6     int x = 1;
7     {
8         // x w zasięgu nazw wewnętrznym { inne niż dwa dotychczasowe x
9         int x = 2;
10        cout << x << endl;
11
12        // odwołanie do globalnego x
13        ::x = x + 2;
14    }
15    cout << x << endl;
16
17    cout << ::x << endl;
18
19    {
20        // tej linii nie da się wykonać { definicja x przykryła x z main
21        int x = x;
22    }
23 }
24
25 void f()
26 {
27     int y = x; // globalne x
28     int x = 22; // lokalne x
29     y = x; // lokalne x
30 }
```

Zawsze zmienna z najbliższej przestrzeni nazw jest wykorzystywana jako podstawowa. Z pojęciem widoczności związane jest również następne omawiane pojęcie:

## 2.3.2 Czas życia zmiennej

Pojęcie czasu życia zmiennej można zdefiniować następująco:

### Definicja 2.4

Czasem życia zmiennej nazywa się ten okres wykonywania programu, w trakcie którego zmienna istnieje w pamięci.

Czas życia zmiennej jest równy czasowi wykonania programu jedynie w przypadku zmiennych globalnych. Zmienne globalne są inicjowane przed uruchomieniem, kasowane po zakończeniu programu.

Zmienne lokalne są tworzone w momencie ich definicji, natomiast kasowane w momencie opuszczenia zasięgu w którym mogłyby być widoczne (a więc do nawiasu klamrowego zamykającego). Po skasowaniu zmiennej nie ma już możliwości odczytania jej zawartości. Zarządzanie czasem życia zmiennych statycznych jest w pełni automatyczne.

Zainteresowani mogą uruchomić poniższy programik – dzięki własnej klasie i jej konstruktorowi i destruktorowi informacje o tworzeniu i kasowaniu zmiennej będzie wyświetlana na ekranie.

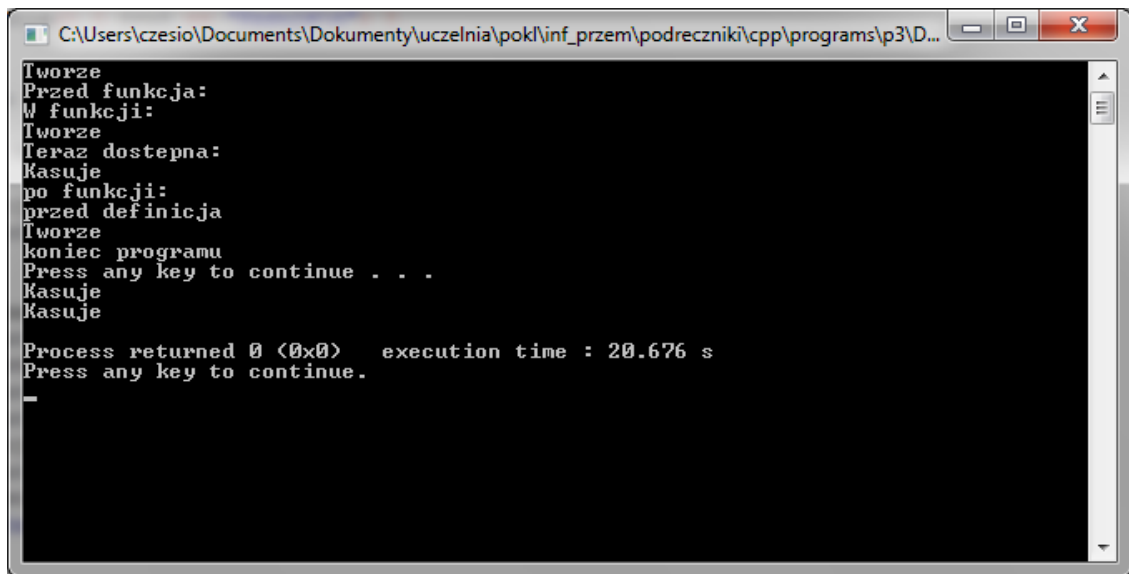
### Kod programu 2.10

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 class CMoja {
7 public:
8     CMoja() { cout << "Tworze\n"; }
9     ~CMoja() { cout << "Kasuje\n"; }
10 };
11
12 CMoja moja;
13
14 void testMoj() {
15     cout << "W funkcji: \n";
16     CMoja a;
17     cout << "Teraz dostepna:\n";
18 }
19
20 int main(int argc, char *argv[]) {
21     cout << "Przed funkcja: \n";
22     testMoj();
23     cout << "po funkcji: \n";
24     cout << "przed definicja\n";
25     CMoja b;
26     cout << "koniec programu\n";
27     system("PAUSE");
28     return EXIT_SUCCESS;
29 }
```



przykładowy wydruk działania programu zamieszczamy na rysunku poniżej:





```

C:\Users\czesio\Documents\Dokumenty\uczelnia\pokl\inf_przem\podreczniki\cpp\programs\p3\D...
Tworze
Przed funkcja:
W funkcji:
Tworze
Teraz dostepna:
Kasuje
po funkcji:
przed definicja
Tworze
koniec programu
Press any key to continue . . .
Kasuje
Kasuje

Process returned 0 (0x0)   execution time : 20.676 s
Press any key to continue.

```

**Rysunek 2.2** – Wynik działania programu prezentującego czas życia zmiennych

### 2.3.3 Modyfikatory deklaracji zmiennych i stałych

Umieszczaniem zmiennych w pamięci oraz ich zachowaniem można w pewien ograniczony sposób sterować. Odpowiednie właściwości uzyskujemy przez modyfikatory podawane w momencie definicji zmiennej. Oto one:

- **register** – oznacza, że zmienna ma być trzymana w rejestrze procesora, a nie w pamięci. Co prawda znów mamy tu do czynienia z wyrażeniem woli programisty – ostateczna decyzja zostanie podjęta przez kompilator, ale przynajmniej przekazujemy mu wskazówki. Przy czym wskazówka zostanie odrzucona, jeśli ... choć raz spróbujemy uzyskać wskaźnik (adres) takiej zmiennej.
- **const** – oznacza, że obiekt jest stały – nie będzie można zmieniać jego wartości. Konsekwencją tego jest obowiązek zainicjowania go (podania wartości początkowej, której nie będzie można zmienić).
- **volatile** – oznacza, że nie ma się wyłączości do podanego obiektu (tzn. może być to rejestr sprzętowy komputera albo zmienna używana przez inny wątek). W praktyce takie zmienne kompilator pomija w procesie optymalizacji dostępu – każdy odczyt wartości zmiennej musi się wiązać z jej ponownym pobraniem z pamięci operacyjnej.
- **static** – w ogólności oznacza, że obiekt taki istnieje przez cały czas, niezależnie od zasięgu, który go używa (zabrania się w ten sposób kasowania zmiennych lokalnych). Każde nowe wywołanie funkcji będzie miało dostęp do wartości zmiennych statycznych z poprzednich wywołań tej funkcji. Dla zmiennych które i tak już istnieją cały czas (zmiennych globalnych) oznacza z kolei zniesienie zewnętrznego symbolu obiektu (tzn. poza bieżącą jednostką kompilacji, czyli plikiem, nic nie może z tego

korzystać). Trochę niemiłe zachowanie – bo mamy rozszerzenie dostępu do zmiennych lokalnych, i zawężenie dla zmiennych globalnych.

- `extern` – było omawiane wcześniej.

Modyfikatory `static` i `extern` wzajemnie się wykluczają, zwłaszcza, że oznaczają dwie całkiem przeciwne właściwości. Mają one też inne znaczenia dla zmiennych i stałych, przykładowo `static`:

1. dla zmiennych globalnych oznacza, że zmienna taka jest wewnętrzna (czyli lokalna dla danego pliku), w przeciwnym razie jest zewnętrzna i może być importowana przez inne pliki. To samo znaczenie ma dla definicji funkcji.
2. dla zmiennych lokalnych oznacza, że zmienna taka nie jest tworzona za każdym wywołaniem funkcji, lecz jest tworzona raz (jak globalna) a lokalny jest tylko jej identyfikator. Można ją zainicjalizować, jednak jest to inicjalizacja podobna do inicjalizacji zmiennej globalnej -: wykonuje się tylko raz. Jeśli tego nie zrobimy, przypisana jej będzie wartość zera. W poniższym przykładzie wykorzystaliśmy zmienną statyczną do zliczania i wyświetlania liczby wywołań funkcji:
- 3.

### Kod programu 2.11

```
1 #include <iostream>
2
3 using namespace std;
4
5 void mojaFunkcja(int a, int b) {
6     // zmienna statyczna zostanie zainicjowana raz i tylko raz
7     // przed uruchomieniem programu, niezależnie od wywołań
8     // funkcji
9     static int liczbaWolan = 0;
10    cout << "Wywołan: " << ++liczbaWolan << " argumenty: " << a << ", " << "b\n";
11 }
12
13 int main() {
14     mojaFunkcja(1, 2);
15     mojaFunkcja(3, 4);
16     mojaFunkcja(5, 6);
17     mojaFunkcja(7, 8);
18
19     system("PAUSE");
20     return 0;
21 }
```

Słowo `static` ma także specjalne znaczenie w odniesieniu do struktur – które omówimy później.

W przypadku słowa kluczowego `extern` również możemy się spodziewać dwóch znaczeń:

1. jeśli poprzedza deklarację zmiennej (globalnej lub lokalnej) lub stałej, ale nie zainicjalizowanej, oznacza to typową deklarację. Można spotkać również funkcje poprzedzone tym modyfikatorem, ale nie ma on wtedy żadnego znaczenia.



2. jeśli poprzedza deklarację stałej zainicjalizowanej, oznacza to, że taka stała ma być również eksportowana do innych plików. Z kolei **extern** przed stałą niezainicjalizowaną oznacza jej import na etapie łączenia (linkowania) programu.

Tu ważna uwaga nt. różnicy traktowania stałych przez C i C++: domyślnie zmienne globalne w C++ są w pamięci globalnej, co oznacza możliwość dostępu do nich z innych plików (jednostek kompilacji). W C taka sama sytuacja dotyczy stałych, natomiast w C++ stałe domyślnie są dostępne tylko w danym pliku (tak jakby były zadeklarowane jednocześnie z modyfikatorem **static**).

Dodatkowe znaczenie **extern** ma w wyrażeniu **extern "C"** lub **extern "C++"**. Wyrażenie takie jest jawnym wymuszeniem języka kompilacji fragmentu kodu. Jest to konstrukcja C++, i zazwyczaj jest stosowana w dwóch przypadkach: eksportowania funkcji do bibliotek łączonych dynamicznie lub statycznie i wykorzystywanych w programach pisanych w innych językach, oraz do importu plików napisanych w C do wykorzystania w programach C++. Przykładowo:

#### Kod programu 2.12

```
1 extern "C" {  
2   #include <clib.h>  
3 }
```



Do tematu **extern** wykorzystywanego dla funkcji jeszcze wrócimy w następnych rozdziałach.

## 2.4 Typy pochodne

Typy pochodne do typów podstawowych pozwalają na operowanie na adresach, łączenie wielu typów podstawowych, czy też finalnie – na tworzenie kompletnych własnych typów danych. Tworzeniem zupełnych typów danych zajmiemy się po zakończeniu omawiania składni.

### 2.4.1 Wskaźniki

Wskaźniki historycznie wywodzą się z niskopoziomowych elementów języka C, ściśle powiązanych ze sprzętem. Wtedy to wskaźnik był fizycznym adresem w pamięci RAM. W C++ do tej pory jest tak często traktowany – choć często nie jest to prawdą. Po pierwsze – większość kompilatorów operuje adresami wirtualnymi w odniesieniu do typów prostych. Po drugie – w przypadku obiektów złożonych nawet ta właściwość (adres wirtualny) nie zawsze musi być zachowana, jeden obiekt może mieć wiele adresów wirtualnych. Tak więc zdecydowanie lepszym przybliżeniem roli wskaźnika będzie następująca definicja:

#### Definicja 2.5

Wskaźnikiem będziemy nazywali unikalną tożsamość danego obiektu, pozwalającą na jego jednoznaczne zidentyfikowanie.

Dzięki temu jesteście w stanie rozróżnić dwie zmienne tego samego typu i mające tę samą wartość.

Mimo że jak wspomnieliśmy, tożsamość między wskaźnikami a fizycznymi adresami nie jest prawdziwa, to w przypadku przechowywania podstawowych typów danych w pierwszym przybliżeniu możemy przyjąć, że zmienna wskaźnikowa przechowuje w pamięci adres elementu danego typu (nie sam element)

Sama zmienna wskaźnikowa w rzeczywistości jest 4-bajtową zmienną statyczną, unikalną dla każdego obiektu (bytu) w programie. Skoro wskaźnik sam w sobie jest zmienną, to oznacza że i on posiada tożsamość, vide - można tworzyć wskaźniki do wskaźników. Podobnie z wszelkimi innymi bytami – możliwe (często nawet wskazane) jest tworzenie wskaźników do obiektów i funkcji.

Wartością zerową, pokazującą „adres donikąd” lub też „byt który nie istnieje” jest NULL lub po prostu 0.

Wskaźniki definiujemy identycznie jak zmienne danego typu – fakt definiowania lub deklarowania wskaźnika a nie zmiennej statycznej oznaczamy podając gwiazdkę przed nazwą zmiennej. Przykłady definicji zmiennych wskaźnikowych:

#### Kod programu 2.13

```
1 char c = 'a';
2 char *p = &c; // wskaźnik na c
3
4 int *pInt; // wskaźnik na int
5 int **ppInt; // wskaźnik na wskaźnik na int
6 int *pInt[10]; // wskaźnik na tablicę int
7 int ***pppInt; // wskaźnik na wskaźnik na wskaźnik na int
8
9
10 int (*fp)(int *p); // wskaźnik na funkcję
11 void (*SetMes)(void *_func); // wskaźnik na wskaźnik na funkcję
```

Pierwsze dwie linie powyższego kodu pokazują nam wzajemną zależność między wskaźnikiem a zmienną – c jest zmienną statyczną, natomiast p – wskaźnikiem do niej. Do wartości zmiennej można się dostać zarówno przez wskaźnik P jak i bezpośrednio korzystając z nazwy C, natomiast jednoznacznie zidentyfikować o jaką zmienną chodzi można tylko poprzez P.

## Operatory pobrania adresu (referencji) oraz wyłuskania

Ze wskaźnikami wiążą się dwa operatory ułatwiające, czy też umożliwiające pracę ze zmiennymi tego typu. Aby uzyskać wskaźnik do zmiennej – stosuje się operator referencji &, aby uzyskać dostęp do wartości wskazywanej, wykorzystuje się operator wyłuskania \*:

#### Kod programu 2.14

```
1  int s1 = 5, s2 = 2;
2  int *p1, *p2;
3
4  p1 = &s1; // p1 wskazuje na zmienną s1
5  p2 = &s2; // p2 wskazuje na zmienną s2
6
7  // wydrukuj kolejno { wskaźnik (adres) zmiennej s2, oraz wartość s2
8  cout << p1 << "\t" << *p1 << endl;
9  // zwiększ wartość wskazywaną (teraz s1)
10 (*p1)++;
11 cout << s1 << endl;
12
13 // kopiowanie wartości wskazywanej (teraz s1) do s2
14 s2 = *p1;
15 s2++;
```

## Arytmetyka wskaźników

W C++ wskaźniki można do siebie dodawać, wykorzystywać jako przełącznik w instrukcji switch, mnożyć, inkrementować itd. - typ wskaźnikowy jest typem przeliczalnym i arytmetycznym. W większości przypadków wyniki takich operacji są zgodne z intuicyjnym wyobrażeniem efektu działania na wskaźniku jak na adresie. W związku z tym możemy traktować operacje na wskaźnikach tak jak bezpośrednie operacje na pamięci RAM, z pominięciem jakiegokolwiek kontroli . . . tak – C++ zakłada, że programista jest człowiekiem inteligentnym – nie zawiedźcie więc tego zaufania, i nie korzystajcie z możliwości takich operacji dopóki na pewno nie wiecie co robicie, i co przez to chcecie osiągnąć!

W przypadku arytmetyki na wskaźnikach działanie operatorów arytmetycznych jest „inteligentne”, tzn – jeśli mamy do czynienia ze zmienną jednobajtową (char) to zwiększenie wskaźnika o 1 przeniesie nas do następnego adresu (zwiększy wartość wskaźnika fizycznie o 1). Jeśli zdefiniowaliśmy wskaźnik na liczbę całkowitą (4 bajty) – to zwiększenie wskaźnika o 1 spowoduje zmianę adresu o 4 (bo skaczymy 4 bajty do przodu). Takie zachowanie wskaźników jest bardzo przydatne przy operacjach wykonywanych na tablicach – ale o tym za chwilę.

Na razie skupmy się na pozostałych aspektach arytmetyki wskaźników. Czasem w trakcie operacji na nich pewna szczątkowa kontrola typów jest zachowana. Przykładowo – nie można przypisać bezpośrednio wartości wskaźnika na char do wskaźnika na int:

### Kod programu 2.15

```
1  char *a;
2  int *b;
3  // źle
4  b = a;
5  // dobrze formalnie { ale może być przyczyną strasznych błędów!
6  b = (int*)a;
```

Można natomiast dokonać jawnego rzutowania – i przypisanie zadziała . . . choć w pamięci nie zostanie zmieniony typ zmiennej A na int, i nie zostanie przydzielona dodatkowa pamięć na nią. Jeśli teraz chcielibyśmy do zmiennej B jakąś wartość, to zniszczymy strukturę pamięci danych naszego programu, i zachowanie się go później jest niezdefiniowane.

Tak więc powtórzymy jeszcze raz – **należy wiedzieć co się robi**. Lecz jak już wiecie co robicie – to zauważcie, do czego może zostać wykorzystany typ `void` (a dokładniej – wskaźnik na ten typ). Skoro wiadomo, że zmienne typu `void` nie istnieją, to powszechnie wykorzystuje się go do porównywania adresów zmiennych dowolnego typu (ich tożsamości), rzutowania zmiennych, i przekazywania dowolnych struktur danych i funkcji między podprogramami. Pojawienie się zmiennej lub argumentu typu ... **`void *cos_tam...`** jest informacją przekazywaną przez jednego programistę innym programistom: „... wyłączam kontrolę typów – musisz wiedzieć jak obsłużyć mój kod, i mechanizmy języka w tym Ci nie pomogą ...”.

## Wskaźniki i stałe

W przypadku pracy ze wskaźnikami musicie pamiętać, że zawsze mamy do czynienia z dwoma różnymi zmiennymi – jedną z nich jest sam wskaźnik, drugą jest zmienna wskazywana. Wykorzystanie **`const`** przy operacjach na wskaźnikach czyni stałym obiekt (zmienną) wskazywany, natomiast w żaden sposób nie wpływa na możliwość czy brak możliwości zmiany wskaźnika (adresu). Dopiero jak podamy **`*const`** - stały staje się wskaźnik. To rozróżnienie jest wykorzystywane głównie przy deklarowaniu parametrów funkcji.

### Kod programu 2.16

```
1 char a, b;
2 // stały wskaźnik do znaku { nie można zmienić wskaźnika,
3 // natomiast można zmienić znak
4 char *const cp = &a;
5 // wskaźnik do stałego znaku { znaku nie można zmienić,
6 // można zmienić wskaźnik
7 char const* cp;
8 // forma alternatywna wskaźnika do stałego znaku
9 const char* cp;
10 // stały wskaźnik do stałego znaku
11 const char *const cp = &a;
12
13 // przykłady zastosowań
14 char n[] = "czem";
15 const char *p = n;
16 // to nie zadziała
17 p[1] = 'a';
18 // natomiast to tak
19 p = &b
```

## 2.4.2 Referencje

Pojęciem związanym z pojęciem wskaźnika jest referencja. W pierwszym przybliżeniu możecie ją rozumieć jako inną nazwę obiektu – czyli faktycznie dostajemy pewną analogię do wskaźników – możliwość odwołania się do jednego obiektu poprzez kilka nazw.

W praktyce możemy tworzyć wiele zmiennych odpowiadających temu samemu obszarowi pamięci, takich „zmiennych wirtualnych” - istniejących w kodzie programu jako nazwa, lecz nie posiadających własnej tożsamości – i to są właśnie referencje. W przeciwieństwie

do wskaźników referencje nie są rzeczywistymi zmiennymi, i obszar ich zastosowań jest zdecydowanie bardziej ograniczony w stosunku do wskaźników. Ich główne zastosowanie to specyfikowanie argumentów funkcji – o tym będzie mowa w dalszej części podręcznika [<link>](#). Fachowo często się mówi o referencjach jako o `<cite>`stałym wskaźniku z niejawnym operatorem adresowania pośredniego`</cite>`. Brzmi skomplikowanie, ale dużo nam mówi o charakterze referencji, między innymi o jej podobieństwie do stałych.

Jeśli chcecie je stosować także w zwykłym kodzie, to pamiętajcie by trzymać się kilku zasad:

- Każda referencja musi zostać zainicjowana w momencie definicji – podobnie jak stałe
- Wskazywanego obiektu dla referencji nie da się zmienić – raz zdefiniowana wskazuje ciągle na tę samą zmienną. S
- Operatory zawsze działają na wartości – nigdy na referencji – inaczej niż we wskaźnikach, ale dokładnie jak w przypadku stałych wskaźników
- Optymalizacja we współczesnych kompilatorach często prowadzi do usunięcia obiektów reprezentujących referencje – podobnie jak w przypadku stałych
- Referencja bez modyfikatora **const** musi zostać zainicjowana L-wartością - stała referencja może być inicjowana wartością stałą (nie l-wartością).

Przykłady definiowania referencji:

#### Kod programu 2.17

```
1  int s1 = 5, s2;  
2  int& r1 = s1;  
3  
4  s2 = r1;  
5  r1 = 6;  
6  r1++;  
7  
8  // źle !  
9  int& r3 = 1;  
10 int& r4;  
11  
12 // dobrze  
13 extern int& r2;  
14 const double& r5=1;
```

## 2.5 Typy złożone

Ogólnie typy złożone w C++ możemy traktować jako zbiorniki na inne elementy (najczęściej typy proste). Zanim przejdziemy do opisów detalicznych poszczególnych typów złożonych - poznajmy ogólne zasady operowania zbiornikami w C++. Co to jest takiego zbiornik? Jest to po prostu taki obiekt, który może w sobie zawierać inne obiekty.

Zasady posługiwania się zbiornikami w C++ (ogólne) są następujące:

- Każdy zbiornik zawiera jakieś elementy (może być również pusty, choć nie każdy - puste tablice nie są dozwolone przez standard).
- Do każdego elementu umieszczonego w zbiorniku możemy się dostać poprzez specjalną wartość, zwaną **iteratorem**. Wartość ta pozwala nam na poruszanie się po zbiorniku.

- Każdy zbiornik ma zdefiniowane odpowiednie wartości iteratorów stanowiące jego początek i koniec.
- Na iteratorze zawsze można wykonać operację pobrania następnego elementu (iterator typu *forward*).
- Dla niektórych typów zbiorników istnieje również możliwość pobrania poprzedniego elementu (iterator *reverse*) lub też możliwości przejścia na dowolny element w jednym kroku (iterator typu *free-access*).

W C++ istnieją różne rodzaje zbiorników o różnych właściwościach, lecz większość z nich jest dostępna jako klasy biblioteki standardowych. Jedynym „wbudowanym” typem zbiornika jest w C++ tablica – gdzie iteratorem jest wskaźnik, i jest to iterator o swobodnym dostępie. Tablice zostaną omówione teraz, pozostałe zbiorniki o dostępie iteracyjnym w dalszej części podręcznika <link>

## 2.6 Tablice

Tablica w C++ to ciąg obiektów jednego typu zajmujący ciągły obszar pamięci. Wielkość tablicy musi być stałą, znaną w czasie kompilacji (nie dotyczy tablic tworzonych dynamicznie za pomocą operatora `new`). Wymiar musi być znany na etapie kompilacji, ale nie musi być koniecznie jawnie podany – jeśli tylko da się go obliczyć na podstawie wartości inicjujących.

Tablica tworzona jest jako typ pochodny pomocą operatora `[]` z:

- typów fundamentalnych (oprócz `void`)
- typów wyliczeniowych
- wskaźników
- tablic (tablice wielowymiarowe)
- klas

Przykłady definicji tablic:

### Kod programu 2.18

```
1 // trójelementowa tablica liczb zmiennoprzecinkowych
2 float v[3];
3 // pięcioelementowa tablica wskaźników na znak
4 char *a[5];
5 // dwuwymiarowa tablica liczb całkowitych
6 int d2[2][10];
7 // trójwymiarowa tablica liczb całkowitych
8 int d3[2][2][2];
9
10 // obliczenie wielkości tablicy z listy inicjującej
11 int v1[] = { 1, 2, 3, 4, 5 };
12 // jeśli lista inicjująca jest zbyt krótka { zostanie automatycznie
```



```

13 // dopełniona zerami
14 int v2[10] = {1, 2};
15
16 // tablica jest równoważna wskaźnikowi na pierwszy element.
17 int *pInt = v1;

```

Jak wspomnieliśmy, iteratorem dla tablicy w C++ jest wskaźnik. Zmienna tablicowa jest tożsama ze wskaźnikiem na jej pierwszy element, i jeśli wrócicie do arytmetyki wskaźników – jasne też okaże się, w jaki sposób można poruszać się po tablicy w alternatywny sposób. Alternatywny do klasycznego indeksowania z wykorzystaniem operatora `[]` – którym zajmiemy się na początek.

Ogólnie dostęp do *i*-tego elementu tablicy można uzyskać poprzez podanie jego indeksu w nawiasach kwadratowych po nazwie zmiennej, przy czym pamiętajcie:



Elementy tablicy są indeksowane (numerowane) zawsze od 0, ostatnim elementem tablicy jest więc element o indeksie *N*-1, jeśli *N* jest liczbą elementów w tablicy.

We wbudowanym typie tablicowym indeksami są zawsze liczby całkowite, przy czym, co również jest bardzo istotne:



C++ nie przeprowadza żadnej kontroli zakresów przy dostępie do elementów tablic.

Brak kontroli zakresu jest charakterystyczny nie tylko dla dostępu indeksowanego, ale także dla dostępu typu iteratorowego.

Oba podejścia do dostępu do elementów są w praktyce równoważne. Czasem możecie spotkać określenia że jeden jest szybszy czy drugi bardziej elegancki – w praktyce dzięki optymalizacjom kompilatorów różnice w czasie dostępu są pomijalne, natomiast co do elegancji – zdecydujcie sami:

### Kod programu 2.19

```

1 int main()
2 {
3     int t[20];
4     // klasycznie (dostęp indeksowany)
5     for (int i = 0; i < 20; ++i)
6         tab[i] = 1;
7     // alternatywnie (dostęp iteratorowy)
8     int *x = t;
9     while (x != tab + 20)
10         *x++ = 2;
11 }

```

Zaprezentowane wyżej tablice były tworzone i kasowane statycznie – na stosie. Istotnym ograniczeniem takiego podejścia jest konieczność obliczenia rozmiaru tablicy na etapie jej kompilacji. Jeśli rozmiar tablicy na tym etapie nie jest znany, koniecznym staje się wykorzystanie dynamicznej alokacji tablic. Do tego celu wykorzystuje się operator



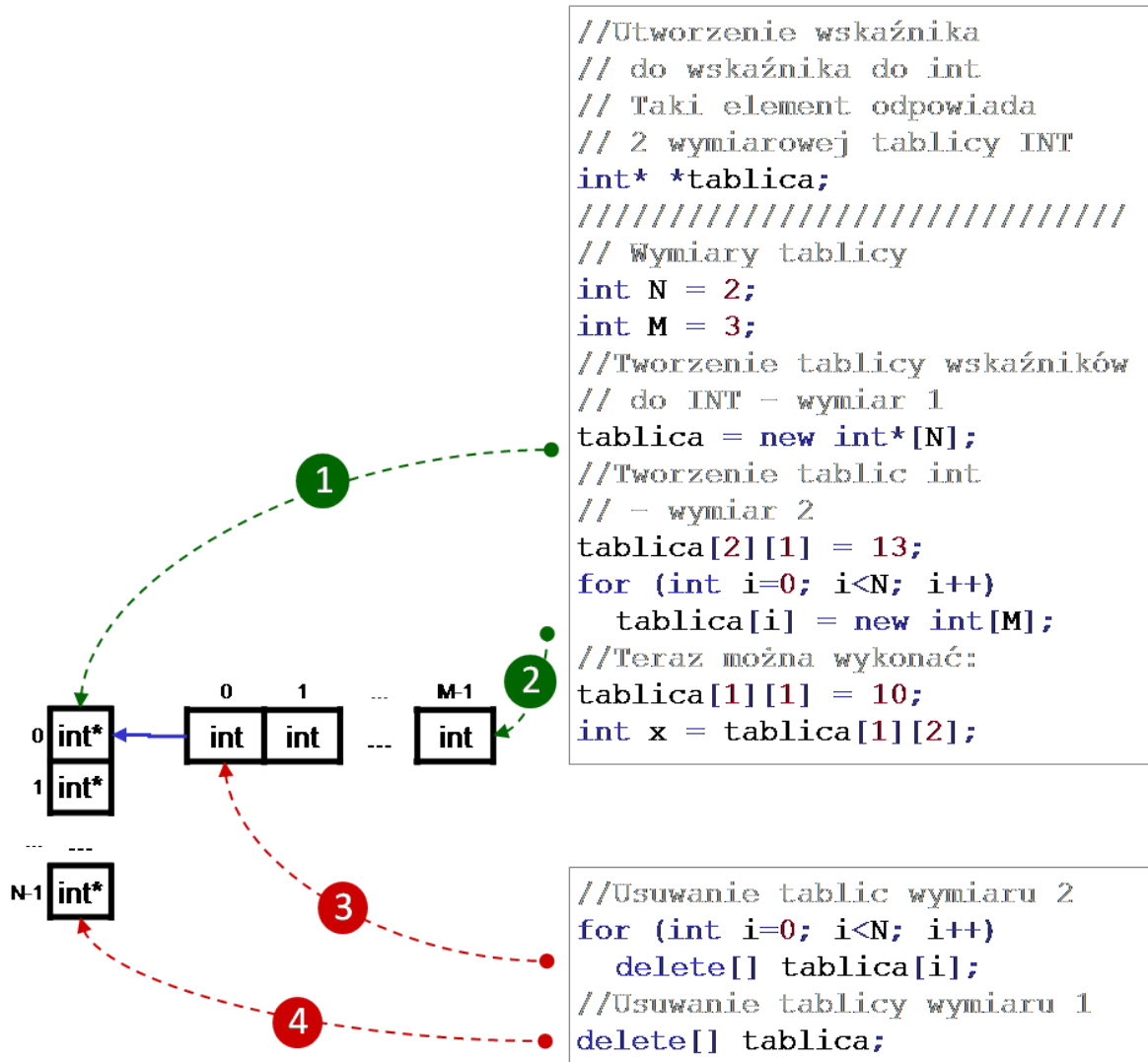
`new[rozmiar]`, z rozmiarem tablicy podawanym wewnątrz nawiasów kwadratowych. Tak utworzone tablice następnie muszą być również ręcznie usunięte za pomocą operatora `delete[]`. Po utworzeniu i przed usunięciem jednowymiarowe tablice statyczne są w pełni równoważne (mogą być stosowane zamiennie) z jednowymiarowymi tablicami statycznymi.

Dynamiczne i statyczne tworzenie tablic jest możliwe również dla tablic wielowymiarowych, przy czym w tym przypadku przestaje obowiązywać równoważność tablic statycznych i dynamicznych. Wielowymiarowa tablica statyczna jest jednolitym obszarem pamięci, a dostęp do odpowiednich elementów tej tablicy jest możliwy dzięki wewnętrznemu przeliczaniu indeksów dwuwymiarowych na indeks jednowymiarowy. W przypadku tablic tworzonych dynamicznie mamy do czynienia z rzeczywistą „tablicą tablic”, co pokazuje choćby kod konieczny do utworzenia i skasowania takiej tablicy:

### Kod programu 2.20

```
1 // dynamiczne tworzenie
2 double **a;
3
4 a = new double[w];
5 for (int i=0; i<w; i++)
6   a[i] = new double[k];
7 ...
8 a[i][j] = 1.5;
9 ...
10 for (int i=0; i<w; i++)
11   delete[] a[i];
12 delete[] a;
```

Dokładniejszy opis działań w przypadku dynamicznego tworzenia i kasowania tablicy zamieściliśmy na poniższym rysunku:



Rysunek 2.3 – Tworzenie i kasowanie wielowymiarowej tablicy dynamicznej.



Pamiętajcie – z punktu widzenia kompilatora, wielowymiarowe tablice statyczne i dynamiczne są zupełnie różnymi typami, nie można ich stosować zamiennie np. jako parametrów przekazywanych do funkcji.

## 2.6.1 Napisy

W C++ podstawowym typem napisowym jest **string**, tyle że nie jest to część języka, lecz jeden z kontenerów z biblioteki STL. W starym C wbudowanego typu napisowego w ogóle nie było - napisy były przechowywane i przetwarzane jako tablica o elementach typu **char**. Dzięki temu tablicę taką możemy inicjalizować stałą napisową:

### Kod programu 2.21

```
1 char napis[20] = "Lolek"; // równoważne { 'L','o','l','e','k','\0' }
```

Zwróćcie uwagę na ostatni znak w tablicy – zawsze powinien być to znak o kodzie 0, podany jawnie lub niejawnie. Takie podejście jest istotą tak zwanych ” *null-terminated strings*” – techniki bardzo niewygodnej, lecz przez tak długi czas będącej standardem, że nie można jej pominąć do dziś.

Wymieńmy niedogodności:

- skoro nie można bezpośrednio kopiować tablic, to nie można także bezpośrednio kopiować napisów,
- nie można ich porównywać (w większości innych języków istnieje domyślny operator dokonujący porównania leksykalnego),
- nie ma także domyślnego działania operatora dodawania, który powinien łączyć napisy,
- do tablicy musimy wpisywać napisy literka po literce (używa się do tego odpowiednich funkcji, pochodzących z biblioteki standardowej C, co nie zmienia braku naturalności takiego podejścia),
- często też powstają problemy z terminatorem napisu – znakiem o kodzie 0. Wstawienie go wewnątrz napisu powodowało skrócenie go do miejsca gdzie było wstawione 0.

Wszystkich wspomnianych wad pozbawiony jest typ **string** – lecz jak wspomnieliśmy – nie jest możliwe zupełne pominięcie go, choćby dlatego, że literały napisowe są zamieniane na stałe „null-terminated strings”

## 2.6.2 Struktury

W przeciwieństwie do tablic struktury to zbiór elementów, które różnią się typem, natomiast powinny pozostawać ze sobą w związku logicznym. Ze względu na wprowadzenie w C++ pojęcia klasy, znaczenie czystych struktur zdecydowanie zmalało. W zasadzie w C++ można traktować strukturę jako zdegenerowaną klasę. Zdegenerowaną – bo pozbawioną kontroli dostępu, ze wszystkimi polami i metodami publicznymi. Napisaliśmy metodami – i to nie jest przeoczenie. W C++ struktura może mieć metody, a nie tylko pola.

Struktury definiujemy przy wykorzystaniu słowa kluczowego **struct**, następnie w nawiasach klamrowych podajemy opis pól i metod. Istnieją dwa ogólne schematy definicji –

struktura nazwana i nienazwana. Zastosowanie struktur nienazwanych ogranicza się do tworzenia okazjonalnych zmiennych strukturalnych – my osobiście rzadko widzimy potrzebę ich stosowania.

Przykłady definicji struktur:

### Kod programu 2.22

```
1 // struktura nazwana
2 struct costam {
3     int a;
4     char b;
5     double c;
6 } ct1;
7
8 costam ct2, ct3 =
9     { 10, 'a', 3.5 };
10
11 // struktura nienazwana
12 struct {
13     int a1, a2;
14     double a3;
15 } ala;
16
17 struct struktura {
18     double a;
19     char nn[5];
20     costam ct;
21     struktura *nast;
22 };
```

Z punktu widzenia składni języka z definicją typów strukturalnych wiążą się pewne nie zawsze jasne aspekty. Po pierwsze – za nawiasem klamrowym zamykającym każdą strukturę musi znajdować się średnik – inaczej niż w przypadku innego stosowania nawiasów klamrowych. Dla każdej struktury zostanie automatycznie wygenerowany operator przypisania, oraz konstruktor kopiujący, lecz wynik działania pozostałych operatorów pozostanie niezdefiniowany. Można mieszać ze sobą (osadzać jedne w drugich) struktury i tablice, lecz – uwaga – w przypadku osadzenia tablicy w strukturze domyślny, wygenerowany operator przypisania i konstruktor kopiujący **przestanią działać prawidłowo!**

Kompilator musi mieć możliwość określenia wielkości każdego pola definiowanej struktury, żeby możliwe stało się obliczenie jej zapotrzebowania na pamięć. Lecz kto pomyśli, że rozmiar struktury jest równy sumie rozmiarów jej pól – może się srodze zawieść.



Rozmiar struktury nie zawsze musi być równy rozmiarowi jej pól i taki sam na różnych systemach operacyjnych.

To niemiłe zachowanie jest spowodowane różnymi ograniczeniami i optymalizacjami dostępu do pamięci w różnych systemach operacyjnych.

Dostęp do pól jest możliwy poprzez operator wyłuskania (kropka) oraz desygnator nazwy pola. W przypadku stosowania wskaźników do struktur, obowiązują w zasadzie te same zasady i ograniczenia co w przypadku wskaźników do pól prostych. Zmienia się jedynie

postać postać operatora wyłuskania na  $\rightarrow$ . Uważajcie na jeszcze jedną nieciekawą cechę - dwie struktury nie są sobie równoważne nawet jeśli mają identyczną definicję. Dla porządku jeszcze wspomnijmy, że nie ma jakiegokolwiek automatycznego rzutowania typów, ani z/na typy proste, ani z / na inne typy złożone, oraz że nazwy pól nie mogą się powtarzać.

Struktura może mieć natomiast pola statyczne – pole takie, podobnie jak w przypadku klas – jest jedno dla wszystkich instancji struktury. Możecie je traktować jako zmienną globalną zdefiniowaną w przestrzeni nazw struktury.

### Kod programu 2.23

```
1 struct struktura {
2     double a;
3     char nn[5];
4     const ct;
5     struktura *nast;
6 };
7
8 struktura ms1, *ms2;
9
10 ms1.a = 10.1;
11 cout << ms1.a;
12 ms2 = &ms1;
13 ms2->a = 10.1;
14 cout << ms2->a;
```

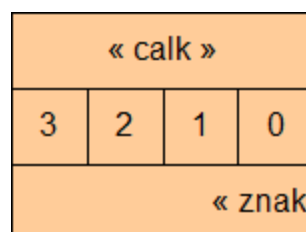
### 2.6.3 Unie

Unie z punktu widzenia kompilatora są bardzo podobne do struktur. Główna różnica polega na tym, że o ile w przypadku struktur każde kolejne pole jest alokowane w innym miejscu pamięci, o tyle w przypadku unii wszystkie pola są alokowane w tym samym obszarze pamięci, i współdzielą swoją wewnętrzną *t* (występuje tzw. zerowe przesunięcie pól). Co to nam daje? Popatrzcie na poniższy przykład:

#### Kod programu 2.24

```
1 union unia
2 {
3     int calk;
4     char znak[];
5 };
6 unia u1;
7 u1.calk = 10 + 0x0f00;
8 for (size_t i = 0; i < sizeof(unia); i++)
9     cout << (int)u1.znak[i] << '\t';
```

Stworzyliśmy w ten sposób jednolity obszar pamięci, który możemy interpretować (w zależności od odwołania) jako liczbę typu `int`, albo jako tablicę znaków (bajtów). W ten oto prosty sposób daliśmy sobie możliwość podejrzenia, co się dzieje w poszczególnych bajtach zmiennej całkowitej, i widzimy, że pamięć jest przydzielana zgodnie z rysunkiem:



**Rysunek 2.4** – Alokacja pamięci w przykładowej unii

Jeśli zdecydujecie się na wykorzystanie unii, pamiętajcie, że:

- Unie nie mogą mieć konstruktorów, destruktorów oraz pól statycznych,
- W tym samym obszarze pamięci mogą być przechowywane różne zmienne,
- Unia zajmuje tyle pamięci co największe z jej pól.

Unie są wyjątkowo mało przenośne, w zasadzie ich zastosowanie ogranicza się do kilku przypadków: przechowywania zmiennych wariantowych, w kodzie niskopoziomowym lub oszczędnych protokołach komunikacyjnych, oraz do dziwnych operacji na pamięci ;)

Dodatkowo, możliwe jest stosowanie unii która ani nie ma zdefiniowanej nazwy, ani nie jest zdefiniowana żadna zmienna jej typu. Oba poniższe przykłady kodu są prawidłowe:

#### Kod programu 2.25

```
1 struct
2 {
3     int a;
4     union {
5         long l
6         char c[4];
7     };
8 } dzial;
```

#### Kod programu 2.26

```
1 union
2 {
3     struct { char c1, c2; short s; } p;
4     long l;
5 };
```

Pusta definicja unii nie oznacza tworzenia jakiegoś nowego typu, jest jedynie informacją, że dane pola współdzielą wspólny obszar pamięci.

## 2.7 Jawne rzutowanie typów

Na koniec tego rozdziału chcielibyśmy wrócić do zagadnienia rzutowania typów w języku C++. Czasami to rzutowanie odbywa się w sposób niejawni i niewidoczny dla programisty, przy czym dzieje się to głównie w obrębie typów użytkownika (klas) znajdujących się w jednej hierarchii, lub też w obrębie typów arytmetycznych. Szczególnie często sytuacja taka ma miejsce w przypadku obliczania wyrażeń arytmetycznych. W tym przypadku C++ kieruje się szlachetną zasadą przyjmowania jako typu wynikowego tego, który ma największą dokładność (żeby nie groziła utrata danych). Tak więc jak dodajemy short do int, to wynik będzie int, jak dzielimy int przez double wynik będzie double .... wszystko pięknie, tylko jak myślicie, jaka wartość będzie w zmiennej x po wykonaniu poniższego kodu?

#### Kod programu 2.27

```
1 double x = 1 / 2;
```

Wydawałoby się że powinno być 0.5 a tu niespodzianka ... w x zostanie zapamiętane 0. Liczby 1 i 2 są liczbami całkowitymi, więc wynik ich podzielenia też będzie liczbą całkowitą. A operacja przypisania jest operacją kolejną w stosunku do dzielenia ... dopiero podczas przypisania nastąpi niejawne rzutowanie int na double.



### 2.7.1 Rzutowanie w stylu C

Aby uniknąć sytuacji analogicznej do opisanej, musimy jawnie wskazać, że choć jeden z argumentów jest typu `double`. Można to zrobić pisząc np. `1.0` zamiast `1`. Można też wymusić rzutowanie w stylu C, podając przed argumentem w nawiasach typ, na jaki zmienna ma zostać przekształcona:

#### Kod programu 2.28

```
1 double x = (double)1/2;
```

Jednak rzutowanie takie nie jest zgodne pod względem składni z filozofią języka C++. Bardziej eleganckie jest rzutowanie w stylu inicjacji:

#### Kod programu 2.29

```
1 double x = double(1)/2;
```

ale i w takim przypadku wymuszamy na kompilatorze naszą decyzję, nie stosując mechanizmów ochronnych C++.

### 2.7.2 Różnego rodzaju cast-y

Oba podejścia wymienione wyżej nie są zalecane w przypadku języka C++. Tutaj standard proponuje zastosowanie jednego z „cast-ów”:

#### `static_cast`

`static_cast` jest typem rzutowania typów pokrewnych niepolimorficznych (nie muszą być w jednej hierarchii dziedziczenia). Jego główne zastosowanie to rzutowanie typów prostych na inne typy proste, przykładowo:

#### Kod programu 2.30

```
1 int x = 1;  
2 double y = static_cast<double>(x)/2;
```

#### `dynamic_cast`

Jest to operator rzutowania typów polimorficznych, muszących pozostawać względem siebie w hierarchii dziedziczenia.

#### `reinterpret_cast`

Niebezpieczny operator rzutowania dowolnych typów na dowolne inne. W praktyce polega na bitowej zmianie znaczenia adresu – w swoim działaniu bardzo przypomina unie.

## **const\_cast**

Kolejny niebezpieczny operator rzutowania – w tym wypadku zmieniana jest zmienna na stałą (co jeszcze nie jest takie złe) jak i stała na zmienną ... to dzięki jego istnieniu pisaliśmy że stałe w C++ nie zawsze konieczne są stałe :)

# Rozdział 3

## Instrukcje

Autorzy:

*Paweł Wnuk*

### 3.1 Wyrażenia i operatory

Na początek chcielibyśmy powrócić do wspomnianej wcześniej kwestii wyrażeń i tego, że prawie wszystko w języku C++ jest wyrażeniem. Skoro prawie wszystko jest wyrażeniem, to i istotną częścią języka są operatory które pozwalają budować owe wyrażenia. Są one najczęściej (ale nie zawsze) określane odpowiednimi symbolami. Każdy z nich ma swoje właściwości. Na początek informacja o wzajemnej zależności operatorów:

#### 3.1.1 Priorytety operatorów

W C++ jest kilkanaście różnych operatorów. By przy interpretacji wyrażeń z nich zbudowanych nie powstał bałagan, są one połączone w dość ścisłą hierarchię, zaprezentowaną na rysunku poniżej:

Nazwa	Symbol	
określenie zakresu	::	L
nazwa globalna	::	P
wybór składnika	->	L
element tablicy	[]	
wywołanie funkcji	()	
nawias w wyrażeniach	()	
rozmiar obiektu	sizeof	P
rozmiar typu	sizeof	
post inkrementacja	++	
pre inkrementacja	++	
post dekrementacja	--	
pre dekrementacja	--	
dopełnienie do 2	~	
negacja	!	
jedno arg. minus	-	
jedno arg. plus	+	
pobranie adresu	&	
wyłuskanie	*	
rezerwowanie	new	
zwalnianie	delete	
zwalnianie wektora	delete	
rzutowanie	[]	
	()	

Nazwa	Symbol	
wybór składnika wsk. i nazwą obiektu	.*	L
wybór składnika wsk. i wsk. do obiektu	->*	
mnożenie	*	L
dzielenie	/	
modulo	%	
dodawanie	+	L
odejmowanie	-	
przesunięcie bit. w lewo	<<	L
przesunięcie bit. w prawo	>>	
mniejszy niż	<	L
mniejszy bądź równy	<=	
większy od	>	
większy bądź równy	>=	
równość	==	L
nierówność	!=	

Nazwa	Symbol	
koniunkcja bitowa	&	L
bitowa różnica symetryczna	^	L
alternatywa bitowa		L
koniunkcja	&&	L
alternatywa		L
arytmetyczne if	? :	L
przypisania	=	P
	*=	
	/=	
	%=	
	+=	
	-=	
	<<=	
	>>=	
	&=	
	=	
	^=	
przecinek	,	L

Rysunek 3.1 – Priorytety operatorów

Od priorytetu operatora zależy, w jakiej kolejności zostaną wykonane operacje z których składa się wyrażenie. W przypadku operatorów o tym samym priorytecie, wyrażenie jest zawsze wykonywane od lewej do prawej. W przypadku, gdybyście mieli wątpliwości odn. kolejności wykonania operacji – celowo jeden z najwyższych priorytetów ma operator nawiasów ().

### 3.1.2 Podział operatorów

Ogólnie operatory można podzielić na kilka grup:

Pierwszy, najbardziej ogólny z podziałów dotyczy liczby argumentów. W tym przypadku mamy do czynienia z operatorami jedno-, dwu-, i wieloargumentowymi. Jednak nie zawsze ten podział jest podziałem zrozumiałym w naturalny sposób. O ile nie budzi on naszego sprzeciwu w przypadku operatorów arytmetycznych (dodawanie dwuargumentowe, jednoargumentowy minus) czy logicznych (logiczna suma która ma dwa argumenty, czy negacja o jednym argumentem), o tyle nie bardzo wiadomo jak rozumieć pojęcie argumentu dla omawianego wcześniej operatora wyłuskania?

Dlatego też naszym zdaniem o wiele ważniejszy jest podział ze względu na rodzaj wykonywanej operacji. Z tego punktu widzenia możemy wydzielić następujące grupy operatorów.

### Operatory arytmetyczne

Do operatorów arytmetycznych zaliczamy oprócz standardowego dodawania, odejmowania, mnożenia i dzielenia także operator modulo, inkrementacji i dekrementacji w dwóch

wersjach, przed- i przyrostkowej, czy też ... operator przypisania `=`. Wynikiem działania operatora przypisania jest wartość przypisana do l-wartości, po wykonaniu ew. konwersji typów. Typ wyniku zastosowania operatora arytmetycznego jest wynikiem niejawniej konwersji typów składowych operacji.

## Operatory relacji

Operatory relacji służą, jak sama nazwa mówi – do ustalenia relacji w jakiej pozostają ze sobą argumenty. W tej grupie mamy dostęp do operatora równoważności (`==`), nierównoważności, większości, mniejszości i pochodnych. Znaczenie tych operatorów może być zdefiniowana przez programistę (i często bywa, w przeciwieństwie do pozostałych operatorów).



Mimo dużej liczby operatorów w C++ nie ma operatora tożsamości, znanego choćby z **php**. Jego brak nie jest dotkliwy ze względu na możliwość porównania wskaźników, o których wspominaliśmy, że jednoznacznie definiują nam tożsamość obiektu. Więc jeśli dwa wskaźniki są równoważne, to wskazywane obiekty są tożsame.

Wynikiem zastosowania operatora relacji jest zawsze wartość logiczna

## Operatory logiczne

Operatory logiczne służą do budowania i wykonywania zdań logicznych. Mamy do dyspozycji operatory logicznej koniunkcji, agregacji i negacji. Wynikiem ich zastosowania jest zawsze wartość logiczna. Uważajcie, by operatorów logicznych nie mylić z następną grupą – operatorami bitowymi.



W ramach umożliwienia lepszej optymalizacji programów i skrócenia czasu wykonania, standard zakłada pewne dodatkowe zasady co do operatorów `&&` i `——` (i tylko dla tych). Wymienione operatory są wyjątkowe w tym sensie, że mają zdefiniowaną kolejność wartościowania wyrażeń (wykonania) i nie zawsze drugie wyrażenie musi zostać obliczone. Zawsze obliczane (wartościowane) są wyrażenia kolejno od lewej do prawej, i w przypadku gdy wartość całego zdania logicznego można ustalić, nie ma potrzeby obliczania pozostałych wyrażeń, więc nie jest to wykonywane.

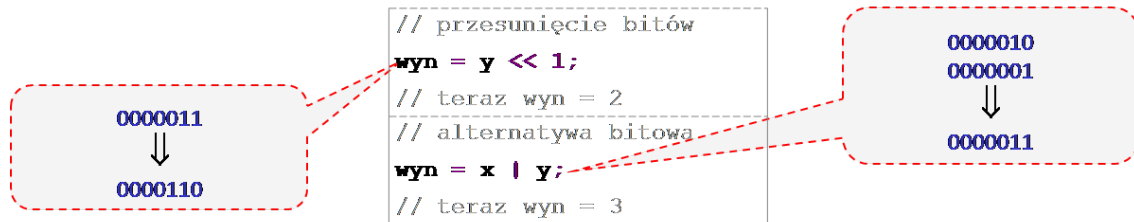
Wyobraźmy sobie że w wyrażeniu `<w1> —— <w2>`, `<w1>` jest prawdą. W takim przypadku niezależnie od tego, jakie byłoby `<w2>`, wynik operacji jest już znany – też jest prawdą. Podobnie dzieje się przy operatorze `&&`.



Pozostałe operatory niczego takiego nie zakładają - w wyrażeniu `<w1> + <w2>` może być najpierw obliczone `<w1>` a potem `<w2>`, ale równie dobrze może być odwrotnie (tu kompilator może sobie zdecydować, jak mu bardziej pasuje).

## Operatory bitowe

Operatory bitowe również wykonują operacje logiczne, lecz nie na całych zmiennych, tylko na pojedynczych bitach wchodzących w skład zmiennych czy stałych. Na pojedynczych bitach – nie znaczy że bezpośrednio można wykonać operacje na wybranych bitach, są one wykonywane na wszystkich bitach danej zmiennej, lecz na każdym z nich niezależnie. Tutaj oprócz typowych koniunkcji, agregacji i negacji mamy dostęp również do różnicy symetrycznej XOR czy przesunięcia bitowego w lewo i w prawo. Typ wyniku po zastosowaniu operatora bitowego jest zależny od typu argumentów.



Rysunek 3.2 – Przykład stosowania operatorów bitowych

## Operatory łączone

W C++ występuje duża grupa operatorów które są połączeniem przypisania wraz z jakąś operacją. Zawsze działają one wg następującego schematu:

### Kod programu 3.1

```
1 // zapis klasyczny
2 x = x+y;
3 // zapis skrócony
4 x += y;
```

W przypadku tych operatorów L-wartość jest jednocześnie jednym z argumentów wyrażenia. Operatory łączone są dla wszystkich operatorów arytmetycznych i bitowych.

## Inne operatory

Trochę dziwna nazwa dla grupy, prawda? Lecz w C++ występuje kilka operatorów, które wyjątkowo ciężko sklasyfikować. Mamy wśród nich wspomniane operatory zasięgu, wyłączenia, ale też i operator pobrania rozmiaru zmiennej czy typu sizeof, operator przecinka zwracający zawsze wartość najbardziej po prawej stronie (ten operator może łączyć wiele argumentów), czy specjalną postać wyrażenia warunkowego (wyrażenie ? wartość\_prawda : wartość\_fałsz).

## 3.2 Instrukcje sterujące

Wszystkie szerzej znane instrukcje wyboru stosowane w innych językach są zaimplementowane także w języku C++, tak więc mamy możliwość korzystania z `if` w wersji prostej i rozszerzonej, `switch` (który działa trochę inaczej niż ma to miejsce w innych językach), oraz trzech rodzajów pętli (tu już mamy bardzo wysoki stopień elastyczności, mocno różniący się od innych języków). W każdej z instrukcji sterujących będziemy mieli do czynienia z koniecznością oceny logicznej wyrażenia w celu podjęcia decyzji. Już o tym wspominaliśmy, lecz nie zaszkodzi powtórzyć:



W C++ wartość 0 (wszystkie bity zmiennej ustawione na 0) jest traktowana jako fałsz. Dowolna inna wartość jest traktowana jako prawda.

Tak więc wyrażeniem, które służy do podjęcia decyzji, nie musi być wyrażenie którego wynikiem jest typ logiczny. Jeśli spojrzymy na poniższy program:

### Kod programu 3.2

```
1 int main() {  
2     int x;  
3     cin << x;  
4     if (x = 0)  
5         cout << "mamy zero";  
6 }
```

Po jego wykonaniu, niezależnie od tego, co zostanie wprowadzone jako `x`, nigdy nie zostanie wyświetlony napis „mamy zero”. Jest to efektem błędu popełnionego przy pisaniu programu – zamiast porównania w `if` mamy przypisanie. Lecz że jest to poprawne składniowo wyrażenie, zwracające wartość przypisywaną, czyli zero, wyrażenie zawsze będzie potraktowane jako fałsz – i napis się nigdy nie wyświetli, mimo że program da się skompilować i uruchomić. Jest to konsekwencją traktowania przez C++ prawie wszystkiego jako wyrażenie.

Mając to na uwadze, przejdźmy do opisu pojęcia instrukcji w języku C++.

### 3.2.1 Instrukcje proste i złożone

Dla każdej instrukcji sterującej dostępnej w języku C++ obowiązuje zasada warunkowego wykonania **jednej i tylko jednej** instrukcji. Może to być instrukcja prosta – najprostszymi przypadkiem. Lecz często chcielibyśmy wykonać ciąg poleceń, co w tym wypadku? W tym celu wprowadzono pojęcie instrukcji złożonej.

#### Definicja 3.1

Blok instrukcji zamknięty w nawiasy klamrowe `{ .. }` tworzy instrukcję złożoną

Taka instrukcja jest nową przestrzenią nazw dla zmiennych, osadzoną w hierarchiczny sposób w nadrzędnych instrukcjach złożonych, aż do przestrzeni nazw wykonywanej funkcji. Dzięki temu wewnątrz bloku możemy definiować nowe zmienne, nawet o takiej samej nazwie jak już zdefiniowane wcześniej w innych, nadrzędnych blokach zmienne <link>.





Ogólna zasada jest taka, że wszędzie tam, gdzie może się znaleźć instrukcja prosta, można też umieścić instrukcję złożoną, przy czym po zamykającym nawiasie klamrowym może – ale nie musi – znaleźć się średnik.

Ściśle rzecz ujmując – po zamykającym nawiasie klamrowym nie może znaleźć się średnik. Natomiast sami możecie sprawdzić, że postawienie średnika nie będzie błędem składniowym ... dlaczego?

Oprócz instrukcji złożonej w C++ występuje pojęcie instrukcji pustej – którą kończy po prostu średnik. Ciąg znaków { ... };; zostanie zinterpretowany jako jedna instrukcja złożona i seria trzech instrukcji pustych. Cecha zarówno miła, jak i niekoniecznie miła ... popatrzcie poniżej:

### Kod programu 3.3

```
1 int main() {
2   int x;
3   cin << x;
4   if (x == 0);
5   cout << "mamy zero";
6 }
```



Po uruchomieniu tego kodu niezależnie od tego co wprowadzimy, zawsze wyświetli się napis „mamy zero” ... bo po if jest instrukcja pusta.

## 3.2.2 Instrukcja warunkowa

Instrukcja warunkowa w języku C++ ma następującą składnię:

### Kod programu 3.4

```
1 if (wyrażenie_warunkowe)
2   // instrukcja wykonywana jeśli wyrażenie jest prawdziwe
3 else // else jest opcjonalne
4   // instrukcja wykonywana jeśli wyrażenie nie jest prawdziwe
```

Nie ma obowiązku podania bloku else – co więcej, jak to widzieliście wcześniej, można też postawić instrukcję pustą w części wykonywanej jeśli wyrażenie jest prawdziwe.

To nie jedyne dopuszczalne udziwnienie w C++. Poniżej prezentujemy inny dziwny, lecz poprawny kod:

### Kod programu 3.5

```
1 int main()
2 {
3   if ( int a = x + 2 > 0 )
4     cout << a << endl;
5   return 0;
6 }
```

Zauważcie, że w wyrażeniu będącym argumentem instrukcji `if` została zdefiniowana zmienna, która jest dostępna w instrukcji podporządkowanej (podobnie byłoby jeśli instrukcją podporządkowaną byłaby instrukcja złożona). W C++ jest to dopuszczalne, ale na tyle dziwaczne, że nie zalecamy nadużywania takiej składni.

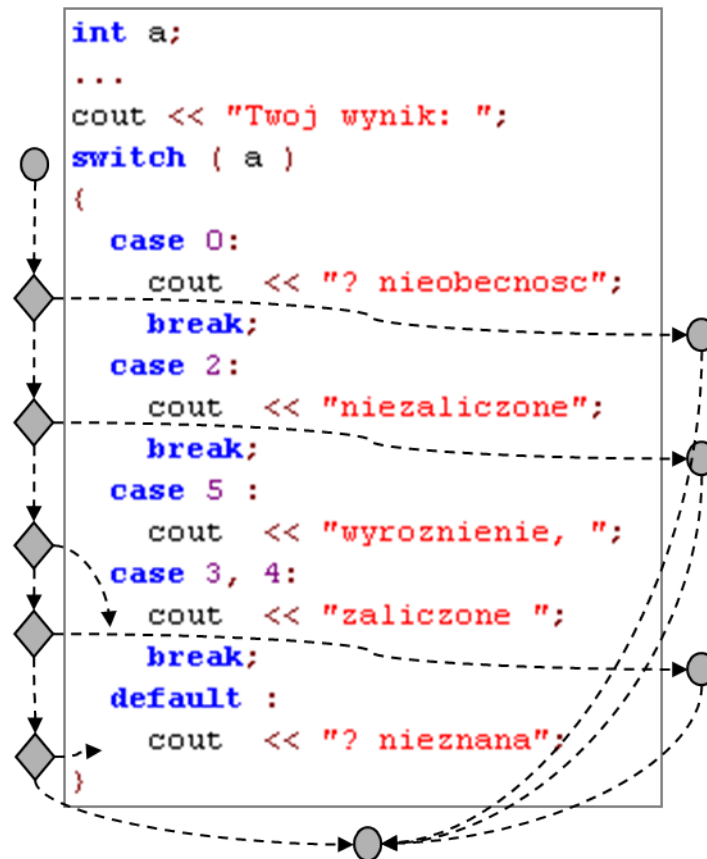
### 3.2.3 Instrukcja wielokrotnego wyboru

Instrukcja wielokrotnego wyboru `switch` ma następującą składnię:

#### Kod programu 3.6

```
1 switch (wyrażenie) {  
2   case 1:  
3     ...  
4   break;  
5     ...  
6   case n:  
7     ...  
8   break;  
9   default:  
10    ...  
11 }
```

Wyrażenie, wg którego następuje różnicowanie wykonania musi być typu (dawać w wyniku) przeliczalnego `<link>`. Zasadę wykonywania instrukcji `switch` zaprezentowaliśmy na rysunku poniżej:



**Rysunek 3.3** – Zasada działania instrukcji switch

Nie jest to działanie do końca zgodne z naszymi przewidywaniami. Po dokonaniu wyboru (wybraniu odpowiedniego case) jest wykonywany kod do wystąpienia break. Jeśli w trakcie wykonania pojawi się następny case – zostanie zignorowany. Ponieważ wewnątrz bloków po case nie jest instrukcją złożoną – nie tworzą własnej przestrzeni zmiennych lokalnych. Z tego względu wewnątrz instrukcji switch **nie można definiować stałych i zmiennych**. Oczywiście jeśli blok instrukcji po case obejmujemy nawiasami klamrowymi – będziemy mogli wewnątrz zdefiniować zmienne, jednak takiego postępowania nie polecamy.

### 3.2.4 Przerwanie wykonania pętli

Zanim przejdziemy do omawiania pętli – zajmijmy się możliwością przerywania ich działania. W przypadku każdej z pętli mamy dwie dodatkowe instrukcje sterujące:

- continue przerywa wykonanie aktualnego przebiegu pętli, i wymusza przeskok do sprawdzenia warunku, i ewentualnego wykonania kolejnego przebiegu.
- break przetrzuwa bezwarunkowo wykonanie pętli i przejście do pierwszej instrukcji poza pętlą.

W przypadku pętli zagnieżdżonych, zawsze continue i break oddziałują tylko na pętlę w której bezpośrednio zostały wywołane ... jeśli mamy pętlę w pętli – nie ma możliwości

przerwania wykonania ich obu jednym poleceniem, chyba że jest to polecenie wyjścia z funkcji.

Nie zalecamy nadużywania, czy też częstego stosowania obu instrukcji `break` i `continue`. Zazwyczaj nie ma potrzeby ich stosowania, a jeśli jest – najczęściej jest przejawem złego zaprojektowania pętli i warunków nią sterujących.



Nie zalecamy stosowania obu poleceń nie tylko dlatego, że są „nieeleganckie”. Dużo ważniejszym powodem jest optymalizacja kompilatorów połączona z konstrukcją współczesnych procesorów. Większość z nich to procesory potokowe, współpracujące z pamięcią cache, dla których niespodziewane przerwanie normalnego (czyt. przewidywanego) toku wykonania programu oznacza ogromną stratę czasu. Trzeba w takim wypadku opróżnić cały potok, oraz najczęściej też uaktualnić zawartość cache.

### 3.2.5 Pętle o nieznanej liczbie iteracji

W C++, podobnie jak w innych językach programowania, mamy do dyspozycji trzy typowe pętle:

- pętlę o nieznanej liczbie iteracji która nie musi być wykonana ani razu - `while`,
- pętlę o nieznanej liczbie iteracji która musi się wykonać co najmniej raz – `do-while`,
- pętlę o zadanej liczbie iteracji `for`.

Jednakże w przeciwieństwie do większości pozostałych języków – każda ze wspomnianych pętli może zostać prosto przekształcona w dowolną inną (są to bardzo elastyczne konstrukcje). Na początek zajmiemy się pętlami typu `while` i `do-while`. Zasada działania obu z nich jest analogiczna, jedyna różnica polega na momencie sprawdzenia warunku – w pierwszym przypadku warunek jest sprawdzany w momencie wejścia do pętli, w drugim – w momencie wyjścia z niej.

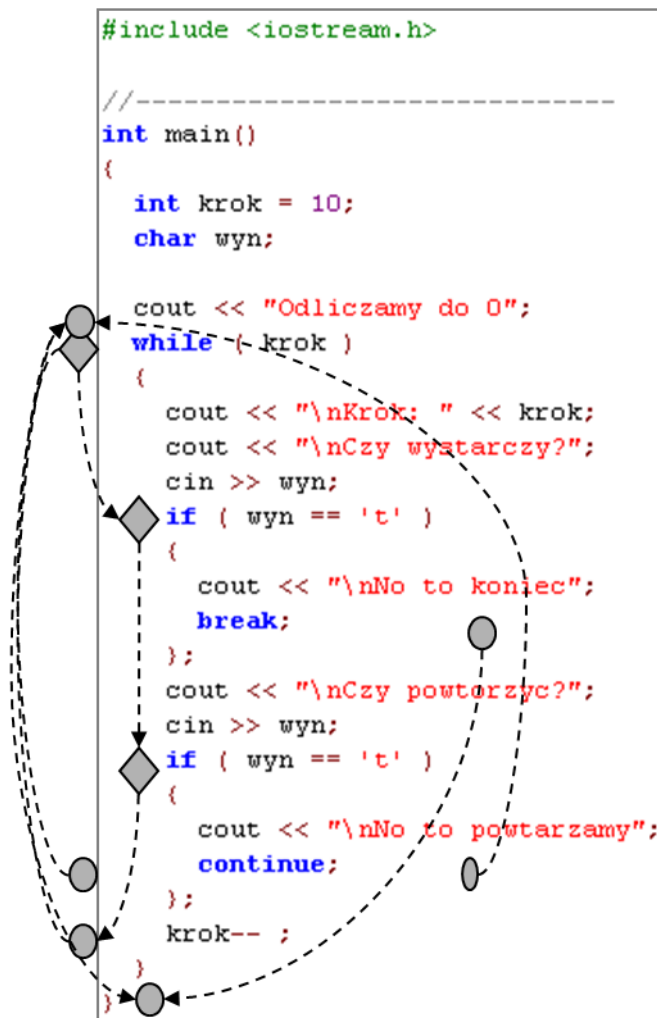
Składnia obu pętli wygląda następująco:

### Kod programu 3.7

```
1 // pętla while
2 while (wyrażenie)
3 // instrukcja wykonywana w pętli
4 instrukcja_w_petli();
5
6 // pętla do { while
7 do
8 // instrukcja wykonywana w pętli
9 instrukcja_w_petli();
10 while (wyrażenie);
```

Podobnie jak w przypadku if, wyrażenie może być dowolnego typu który ma jakąś wartość (czyli każdego poza void). Zasady wartościowania wyrażenia są również identyczne jak w przypadku instrukcji warunkowej.

Pętla jest wykonywana dopóki wyrażenie jest prawdziwe, wg pokazanego schematu:



Rysunek 3.4 – Schemat działania pętli while

<example>

### 3.2.6 Pętla o zadanej liczbie iteracji

Teoretycznie bardziej skomplikowaną niż omawiane wcześniej pętlą jest pętla o zadanej liczbie iteracji. Zazwyczaj taka pętla musi mieć zmienną sterującą, w wielu językach programowania w momencie wejścia do pętli musi być znana i ustalona liczba iteracji. W C++ oba te stwierdzenia są nieprawdziwe. Pętla `for` nie musi mieć zmiennej sterującej, inkrementowanej / dekrementowanej co krok. Liczba iteracji także nie musi być znana przed rozpoczęciem pętli – może się zmieniać w trakcie wykonywania pętli ... Zwyczajowo stosuje się ją jako pętlę o zadanej liczbie iteracji, lecz nic nie stoi na przeszkodzie by tą pętlę wykorzystać analogicznie do `while` czy `do-while`.

Składnia wygląda następująco:

#### Kod programu 3.8

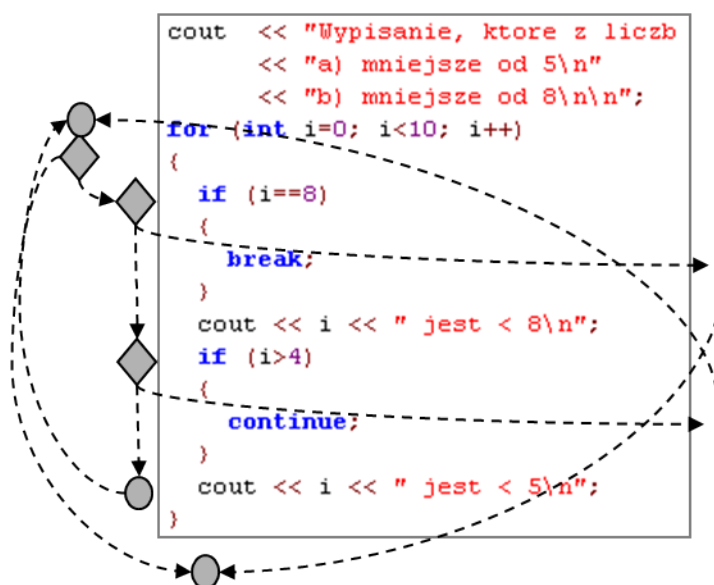
```
1 for (inicjalizacja; wyrażenie_warunkowe; krok)
2   // instrukcja wykonywana w pętli
3   instrukcja_w_petli();
```

Każdy z wymienionych elementów (`inicjalizacja`, `wyrażenie_warunkowe`, `krok`) są niezależnymi wyrażeniami / instrukcjami, każde z nich może zostać pominięte. Moment wykonania każdej z instrukcji wygląda następująco:

- `inicjalizacja` – instrukcja wykonywana przed pierwszym przebiegiem pętli. Zwyczajowo znajduje się tutaj definicja i inicjacja zmiennej sterującej.
- `wyrażenie_warunkowe` – instrukcja / wyrażenie warunkujące wykonanie kolejnego przejścia pętli, przy czym pętla może nie zostać wykonana ani razu. Jeśli pominiemy ten element – pętla nigdy się nie zakończy. Zwyczajowo umieszcza się tutaj wyrażenie testujące zmienną sterującą na okoliczność zakończenia pętli
- `krok` – instrukcja wykonywana po każdym wykonaniu pętli. Zwyczajowo tutaj znajduje się inkrementacja zmiennej sterującej.

W przypadku pominięcia wszystkich wyżej wymienionych elementów uzyskamy pętlę nieskończoną.

Tok wykonania pętli for pokazujemy na rysunku poniżej:



Rysunek 3.5 – Schemat działania pętli for



# Rozdział 4

## Funkcje

Autorzy:

*Paweł Wnuk*

### 4.1 Funkcje

#### 4.1.1 Wiadomości podstawowe

W języku C++ funkcje są podstawowymi jednostkami wykonawczymi, i są podstawowym elementem projektowania strukturalnego – to czego uczyliście się do tej pory opierało się głównie na funkcjach. Często funkcje są określane jako podprogram (ang. *subroutine*), aczkolwiek w C++ mają nieco większe możliwości. Nieco większe – nie oznacza że w C++ możliwe jest programowanie funkcjonalne. Niestety takiej możliwości wciąż nie mamy.

#### Składnia deklaracji i definicji funkcji

Jak już wspominaliśmy, rozróżnia się deklarację i definicję funkcji.

Deklaracja wygląda następująco:

##### Kod programu 4.1

```
1 <modyfikator> <typ_zwracany> <nazwa>( <typ1> <arg1>, <typ2> <arg2> <...> );
```

Definicje funkcji wyglądają bardzo podobnie, z tym że zamiast ; kończącego deklarację mamy nawiasy klamrowe { } zawierające "ciało" (ang. *body*) funkcji.

### Kod programu 4.2

```
1 <modyfikator> <typ_zwracany> <nazwa>( <typ1> <arg1>, <typ2> <arg2> <...> ) {  
2   <instrukcja 1>;  
3   <instrukcja 2>;  
4   ...  
5   <instrukcja n>;  
6   return wartosc_zwracana;  
7 }
```

W nagłówku funkcji występują następujące elementy:

- modyfikator - opcjonalny modyfikator funkcji, omówimy go nieco później <link>.
- typ\_zwracany – typ wartości zwracanej przez funkcję, jeśli jej wywołanie zostało umieszczone w wyrażeniu. Wynik jest zwracany poprzez wywołanie instrukcji return z argumentem będącym zwracaną wartością. Jeśli funkcja nie zwraca żadnej wartości, typ zwracany należy zdefiniować jako void, i konsekwentnie return nie powinien mieć podanego żadnego argumentu lub argument typu void.
- nazwa – identyfikator, przy użyciu którego będzie wywoływana funkcja <link>. Wbrew pozorom, w C++ nazwa nie identyfikuje jednoznacznie funkcji. Poza nazwą do identyfikacji jest również wykorzystana lista typów argumentów, dzięki czemu możliwe jest opisane dalej <link> przeciążanie nazw funkcji.
- lista par: typ\_argumentu nazwa\_argumentu oddzielona przecinkami. Nazwa jest opcjonalna – nie musi występować w deklaracji bez jakichkolwiek konsekwencji, i nie musi też występować w definicji, tyle że wtedy taki argument nazywamy nienazwanym, i nie możemy go wykorzystywać w instrukcjach w ciele funkcji. Nazwa też nie musi się zgadzać w definicji i deklaracji, choć takie zachowanie świadczy o niechlujnym przygotowaniu kodu.
- argumenty funkcji mogą mieć wartości domyślne (domniemane). Wartości te są wpisywane w deklaracji funkcji, przy czym argumenty domniemane muszą być podane na końcu listy. Przy wywołaniu funkcji można pominąć tylko argumenty z końca listy, które mają wartości domyślne.
- Zapis (...) oznacza dowolną liczbę argumentów.

<example>

### Przekazywanie argumentów do funkcji

Domyślnie argumenty przekazujemy do funkcji przez wartość. Ogólnie ten sposób jest dobry, pod warunkiem że wartość argumentu nie będzie zmieniana wewnątrz funkcji (jest to p-wartość), oraz że przekazywany argument ma niewielki rozmiar. Dlaczego akurat te warunki muszą zostać spełnione?

Argumenty przekazywane jako wartość są przekazywane poprzez stos, za każdym razem tworząc kopię przekazywanej zmiennej czy stałej. Stąd ograniczenie na wielkość argumentu przekazywanego przez wartość - wykonywanie po kilkanaście tymczasowych kopii dużego obiektu zakrawa na śmieszność – ze względu na zużycie pamięci, oraz czas potrzebny na kopiowanie.

Inna właściwość, która się co prawda z poprzednią nie wiąże w sposób logiczny, ale której brak jest również mankamentem przekazywania przez wartość, to fakt, że wewnątrz funkcji nie mamy żadnego wpływu na zmienne, jakie jej zostały przekazane. Klasycznym przykładem, który to obrazuje, jest funkcja o nazwie 'swap', której zadaniem jest zamienić miejscami wartości w dwóch zmiennych.



W niektórych językach (np. w Javie) takiej funkcji nie da się jakimkolwiek sposobem zrobić .... widzicie jakiego fajnego języka się uczyć? ;)

Pierwsze podejście do funkcji mogłoby wyglądać następująco:

#### Kod programu 4.3

```
1 void swap(int a, int b) {
2   int temp = a;
3   a = b;
4   b = temp;
5 }
6 ...
7 int main() {
8   int x=10, y=20;
9   swap(x, y);
10  cout << x << " " << y;
11  ...
12 }
```

Lecz po jej wywołaniu okaże się, że zmienne w funkcji nadrzędnej nie zostały zmienione. Funkcja operuje na kopiach wartości argumentów, które jednocześnie są jej zmiennymi lokalnymi.

Poznaliście wskaźniki, więc może ich wykorzystanie będzie jakimś rozwiązaniem? Przekazywana jest co prawda nadal wartość, ale tą wartością jest wskaźnik do zmiennej, wystarczy więc posłużyć się wyłuskiwaniem:

#### Kod programu 4.4

```
1 void swap(int *a, int *b ) {
2   int temp = *a;
3   *a = *b;
4   *b = temp;
5 }
6 ...
7 int main() {
8   int x=10, y=20;
9   swap(&x, &y);
10  cout << x << " " << y;
11  ...
12 }
```



Niestety, jak zauważyliście, konieczna jest zmiana sposobu wywołania funkcji, co nie wygląda najlepiej.

W C++ jest jeszcze jedna możliwość (w starym C niedostępna) – przekazywanie parametrów przez referencję. Wygląda to podobnie do definicji zmiennych typu referencyjnego <link> - wystarczy zmienić nagłówek:

#### Kod programu 4.5

```
1 void swap(int &a, int &b) {  
2     int temp = a;  
3     a = b;  
4     b = temp;  
5 }  
6 ...  
7 int main() {  
8     int x=10, y=20;  
9     swap(x, y);  
10    cout << x << " " << y;  
11    ...  
12 }
```

I voila – otrzymujemy to, o co nam chodziło.

Podsumowując:

- W C++ możemy przekazywać argumenty do funkcji przez wartość i przez referencję (pamiętajcie – przekazanie wskaźnika jest przekazaniem przez wartość),
- Jeśli argumenty są małe (w sensie zajętości pamięci) i nie są zmieniane wewnątrz funkcji (a przynajmniej ta zmiana nie musi być widoczna poza ciałem funkcji) – korzystamy z przekazywania przez wartość
- W pozostałych przypadkach powinien być przekazany wskaźnik, lub wykorzystana referencja.

Dodatkowo, warto pamiętać, że tablice zawsze są przekazywane do funkcji jako wskaźnik.

#### Zwracanie wyników działania funkcji

Funkcja zwraca wyniki zazwyczaj poprzez return <wartość>, przy czym typ zwracanej wartości jest określony w nagłówku funkcji.

Do zwracania wyniku funkcji możemy wykorzystać każdą z metod omawianych wcześniej przy przekazywaniu argumentów. I podobnie jak wtedy - zwracanie dużych obiektów przez wartość, choć nie jest zalecane, nie stanowi problemu (zostanie wykonana kopia).

Lecz gdy zwracamy duże obiekty, i chcemy zrobić to elegancko, to mamy problem ... kto (funkcja? obiekt?) ma zarządzać czasem życia zwróconej wartości?

Jeśli typ zwracany jest typem podstawowym (lub typem niewielkich rozmiarów), to zwróćcie go po prostu przez wartość i nie zwracajcie sobie głowy szczegółami. W pozostałych wypadkach macie do wyboru:

1. Zwracamy obiekt tymczasowy (czyli przez wartość) – a wstyd i narzut czasowy trzeba zaakceptować...
2. Zwracamy referencję do obiektu. Do jakiego? Nie może być to zmienna lokalna w funkcji ... mamy nadzieję że już rozumiecie dlaczego. Ale może do ... zmiennej statycznej w funkcji?. Sprytnie, nie? Jednak unikajcie takich rozwiązań – mało intuicyjne, prowadzi do tworzenia zmiennych statycznych które wykorzystywane są niekoniecznie zgodnie z przeznaczeniem.
3. Zwracamy wskaźnik do obiektu. W ten sposób należy jednak przyjąć prostą zasadę – tak zwracane są jedynie zmienne i obiekty tworzone dynamicznie, i odbiorca wyniku jest zobowiązany do usunięcia tego obiektu, przy czym jeszcze lepiej jest zmusić odbiorcę do skasowania obiektu dzięki odpowiednim zbiornikom, np. `auto_ptr <link>`.

### 4.1.2 Przeciążanie nazw

Wspomnieliśmy wcześniej, że w C++ nazwa funkcji nie jest jej unikalnym identyfikatorem – że kompilator automatycznie dodaje do nazw przyrostek zawierający typy wszystkich argumentów w odpowiedniej kolejności. Ten mechanizm jest wykorzystany do przeciążania nazw funkcji, czyli pozwolenia na to, by w programie mogło występować kilka funkcji o tej samej nazwie. Oczywiście muszą się różnić listą parametrów, natomiast wartość zwracana nie może być wykorzystana do rozróżnienia.

Przykłady prawidłowego i nieprawidłowego przeciążania:

#### Kod programu 4.6

```
1 // prawidłowo
2 int funkcja(int _a);
3 int funkcja(int _a, int _y);
4 int funkcja(double _a);
5 int funkcja(string _a);
6
7 // nieprawidłowo
8 int funkcja(int _a);
9 double funkcja(int _a);
```

Kompilator sam dopasuje odpowiednią funkcję w momencie wywołania, kierując się następującymi zasadami:

- Ścisła zgodność (bez konwersji lub konwersje trywialne – tablica na wskaźnik, stała na zmienną, nazwa funkcji na funkcję)
- Zgodność z zastosowaniem promowania typów całosciowych (bool na int, char na int i odpowiedniki bez znaku)
- Zgodność z zachowaniem standardowych konwersji (całkowite na zmiennoprzecinkowe i vice versa, rzutowania klas, rzutowanie na void\*)
- Zgodność z zachowaniem definiowanych konwersji (dla klas)
- Zgodność z wielokropkiem w deklaracji funkcji.

Przykład:

#### Kod programu 4.7

```
1 void drukuj(int _x);
2 void drukuj(const char* _x);
3 void drukuj(double _x);
4 void drukuj(long _x);
5 void drukuj(char _x);
6
7 void h(char _c, int i, short s, float f)
8 {
9     // ścisła zgodność
10    drukuj(c);
11    drukuj(i);
12    // trywialna konwersja
13    drukuj('c');
14    drukuj(49);
15    drukuj(0);
16    drukuj("A");
17
18    // promocja w zakresie typów całkowitych
19    drukuj(s); // wywoła drukuj(int)
20    // promocja w zakresie konw. standardowych
21    drukuj(f); // wywoła drukuj(double)
22 }
```

Jeśli znaleziono więcej niż jedną pasującą funkcję na tym samym poziomie – będzie błąd kompilacji. Jak się taki błąd pojawi – co robić? Najlepiej ręcznie rozwiązać wątpliwości kompilatora:

#### Kod programu 4.8

```
1 void f1(char);
2 void f1(long);
3
4 void f2(char*);
5 void f2(int*);
6 ...
7 int i;
8 f1(i); // niejednoznaczne
9 f1(long(i)); // ok
10 f2(0); // niejednoznaczne
11 f2(static_cast<int*>(0)); // ok
12
13 double pow(int x, int y);
14 double pow(double x, double y);
15 ...
16 pow(2.0, 2); // niejednoznaczne
17 pow(2.0, (double)2); // i już ok.
```

### 4.1.3 Wskaźniki do funkcji

Wskaźnik na funkcję to ... jak się łatwo domyśleć – adres w pamięci, gdzie zaczyna się jakaś funkcja. To chyba dość oczywiste – zmienne mają adresy, bo są umieszczone w ob-

szarze pamięci przeznaczonym na dane. Funkcje też mają adresy – bo są umieszczone w obszarze pamięci przeznaczonym na program. Jak z nich skorzystać? Całkiem prosto. Po pierwsze – nazwa funkcji standardowo może zostać zinterpretowana jako adres – wystarczy, że napiszemy ją bez nawiasów i argumentów na końcu. Dlatego też poniższy kod jest poprawny:

#### Kod programu 4.9

```
1 double funkcja(double (*wskaznik_na_funkcje)(double), double x) {double wynik;wynik = (*wskaznik_na_funkcje)(x);  
2 cout << funkcja(sin, 1.75) << funkcja(cos, 1.75);
```

To co zrobiliśmy – to po prostu wykorzystaliśmy funkcję pomocniczą, która przyjmuje dwa argumenty. Drugi z nich to zwykła liczba zmiennoprzecinkowa. A pierwszy – to właśnie wskaźnik na funkcję. Definicja wskaźnika na funkcję jest bardzo podobna do definicji wskaźnika na zmienną. Zobaczcie:

#### Kod programu 4.10

```
1 /** Definicja wskaźnika na funkcję przyjmującą jeden parametr  
2 typu double i zwracającą jedną wartość takiego samego typu.  
3 Od tego momentu będzie istniała zmienna wskaźnikowa (przechowująca  
4 adres) o nazwie func. */  
5 double (*func)(double);func = sin; /// przypisanie do zmiennej adresu funkcji sina = func(1.74);
```

Można także skorzystać z tablicy wskaźników na funkcje - deklaracja tablicy wygląda tak:

#### Kod programu 4.11

```
1 funkcje[0] = sin;  
2 funkcje[1] = cos;  
3 funkcje[2] = tan;  
4 funkcje[3] = ctg;
```

Jak widzicie – nie ma w tym jakiegokolwiek magii czy problemów składniowych. Z doświadczenia wiem, że największym problemem jest zrozumienie i zaakceptowanie faktu, iż funkcja też może być zmienną ... więc jednak zrobiliśmy krok w kierunku programowania funkcyjnego. No to pora na krok następny.

### 4.1.4 Wyrażenia lambda

Wyrażenia lambda są w języku C++ nowością – pojawiły się dopiero w momencie definicji standardu C++11 – dlatego nie są też powszechnie znane społeczności programistów C++. Do czego właściwie służą takie wyrażenia? Ich podstawowe zastosowanie to tworzenie anonimowych funkcji. Jeśli ktoś z Was zna choćby podstawy JavaScript to wie, jak takie funkcje bez nazwy mogą być przydatne. Funkcja anonimowa to funkcja która nie posiada nazwy, lecz posiada swoje ciało, i można ją wywołać. Zasady tworzenia ciała funkcji lambda, jak i wszelkie ograniczenia kodu tam zamieszczanego są dokładnie takie same jak w przypadku standardowych funkcji C++ - cała różnica opiera się na nazwie, a dokładniej – jej braku. Więc dlaczego to takie użyteczne? Bo wygodne ... może mi jeszcze nie wierzycie, ale uwierzycie ... ;>



Sama składnia wyrażenia lambda jest następująca:

#### Kod programu 4.12

```
1 [] //początek wyrażenia lambda(*definicje argumentów*)->/*typ wartości zwracanej*/{ //ciało funkcji
```

Przykładowe wyrażenie może więc wyglądać następująco:

#### Kod programu 4.13

```
1 /// odpowiadający mu kod klasyczny:  
2 double dajPi() { return 3.14; }  
3 dajPi();
```

Oczywiście – jeśli wyrażenie lambda zwraca jakąś wartość – to możemy ją odebrać, podobnie jak w przypadku zwykłych funkcji:

#### Kod programu 4.14

```
1
```

No dobrze – to gdzie ta wygoda, pomijając krótszy zapis jak na razie? Wygoda pojawia się, jak zaczniemy korzystać z adresów takiego wyrażenia. Adres możemy uzyskać pomijając nawiasy z wartościami parametrów w definicji wyrażenia. Przykład z dodawaniem dwóch liczb:

#### Kod programu 4.15

```
1 cout << pAdres( 5, 6 ) << pAdres( 7, 8 );  
2 return 0;
```

Ciągle nie do końca widać tą wygodę. Ale ... przejdźmy do funkcji ogólnych, takich jak sort z biblioteki standardowej. Funkcja sort przyjmuje jako parametr wskaźnik do funkcji porównującej dwa pozostałe argumenty. Tą funkcję porównującą zawsze do tej pory musieliśmy definiować – wymyślać jej nazwę, zaśmiecać główną przestrzeń nazw, itp ... Teraz można podać w to miejsce wyrażenie lambda:

#### Kod programu 4.16

```
1 #include <cstdio>  
2 #include <vector>  
3 #include <ctime>  
4 #include <cstdlib>  
5 #include <algorithm>  
6  
7 int main()  
8 {  
9     std::srand( std::time( NULL ) );  
10    typedef std::vector < int > VDaneT;  
11    VDaneT dane( 10 );  
12    for( size_t i = 0; i < dane.size(); ++i )  
13        dane[ i ] = rand() % 50 + 1;  
14
```

```
15  std::sort( dane.begin(), dane.end(), []( const int & a, const int & b )->bool { return a > b; } );
16
17  for( size_t i = 0; i < dane.size(); ++i )
18      std::printf( "%d, ", dane[ i ] );
19
20  return 0;
21 }
```

Teraz mam nadzieję, iż widać że wygodnie ;)

### 4.1.5 Konwencja wywołania funkcji

Trochę trudno w to uwierzyć, ale podanie (zdawałoby się) wszystkiego, co można powiedzieć o danej funkcji: jej parametrów, wartości przezeń zwracanej, nawet nazwy - nie wystarczy kompilatorowi do jej poprawnego wywołania. Będzie on aczkolwiek wiedział, co musi zrobić, ale nikt mu nie powie, jak ma to zrobić. Cóż to znaczy? Celem wyjaśnienia porównajmy całą sytuację do telefonowania. Gdy chcemy zadzwonić pod konkretny numer telefonu, mamy wiele możliwych dróg uczynienia tego. Możemy zwyczajnie pójść do drugiego pokoju, podnieść słuchawkę stacjonarnego aparatu i wystukać odpowiedni numer. Możemy też sięgnąć po telefon komórkowy i użyć go, wybierając na przykład właściwą pozycję z jego książki adresowej. Teoretycznie możemy też wybrać się do najbliższej budki telefonicznej i skorzystać z zainstalowanego tam aparatu. Wreszcie, możliwe jest wykorzystanie modemu umieszczonego w komputerze i odpowiedniego oprogramowania albo też dowolnej formy dostępu do globalnej sieci oraz protokołu VoIP ( Voice over Internet Protocol ). Technicznych możliwości mamy więc mnóstwo i zazwyczaj wybieramy tę, która jest nam w aktualnej chwili najwygodniejsza. Zwykle też osoba po drugiej stronie linii nie odczuwa przy tym żadnej różnicy.

Podobnie rzecz ma się z wywoływaniem funkcji. Znając jej miejsce docelowe (adres funkcji w pamięci) oraz ewentualne dane do przekazania jej w parametrach, możliwe jest zastosowanie kilku dróg osiągnięcia celu. Nazywamy je konwencjami wywołania funkcji.

#### Definicja 4.1

Konwencja wywołania

Konwencja wywołania (ang. calling convention ) to określony sposób wywoływania funkcji, precyzujący przede wszystkim kolejność przekazywania jej parametrów.

Dziwicie się zapewne, dlaczego dopiero teraz mówimy o tym aspekcie funkcji, skoro jasno widać, iż jest on nieodzowny dla ich działania. Przyczyna jest prosta. Wszystkie funkcje, jakie samodzielnie wpisujemy do kodu i dla których nie określimy konwencji wywołania, posiadają domyślny jej wariant, właściwy dla języka C++. Jeżeli zaś chodzi o funkcje biblioteczne, to ich prototypy zawarte w plikach nagłówkowych zawierają informacje o używanej konwencji. Pamiętajmy, że korzysta z nich głównie sam kompilator, gdyż w C++ wywołanie funkcji wygląda składniowo zawsze tak samo , niezależnie od jej konwencji. Jeżeli jednak używamy funkcji do innych celów niż tylko prostego przywoływania (a więc stosujemy choćby wskaźniki na funkcje), wtedy wiedza o konwencjach wywołania staje się potrzebna także i dla nas.

Jak już wspomniałem, konwencja wywołania determinuje głównie przekazywanie parametrów aktualnych dla funkcji, by mogła ona używać ich w swoim kodzie. Obejmuje to

miejsce w pamięci, w którym są one tymczasowo przechowywane oraz porządek, w jakim są w tym miejscu kolejno umieszczane. Podstawowym rejonem pamięci operacyjnej, używanym jako pośrednik w wywołaniach funkcji, jest stos. Dostęp do tego obszaru odbywa się w dość osobliwy sposób, który znajdują zresztą odzwierciedlenie w jego nazwie. Stos charakteryzuje się bowiem tym, że gdy położymy na nim po kolei kilka elementów, wtedy mamy bezpośredni dostęp jedynie do tego ostatniego, położonego najpóźniej (i najwyżej). Jeżeli zaś chcemy dostać się do obiektu znajdującego się na samym dole, wówczas musimy zdjąć po kolei wszystkie pozostałe elementy, umieszczone na stosie później. Czynimy to więc w odwrotnej kolejności niż następowało ich odkładanie na stos.

Jeśli zatem wywołujący funkcję (ang. caller) umieści na stosie jej parametry w pewnym porządku (co zresztą czyni), to sama funkcja (ang. callee - wywoływana albo routine - podprogram) musi je pozyskać w kolejności odwrotnej, aby je właściwie zinterpretować. Obie strony korzystają przy tym z informacji o konwencji wywołania, lecz w opisach "katalogowych" poszczególnych konwencji podaje się wyłącznie porządek stosowany przez wywołującego, a więc tylko kolejność odkładania parametrów na stos. Kolejność ich podejmowania z niego jest przecież dokładnie odwrotna.

Nie myśl jednak, że kompilatory dokonują jakichś złożonych permutacji parametrów funkcji podczas ich wywoływania. Tak naprawdę istnieją jedynie dwa porządki, które mogą być kanwą dla konwencji i stosować się dla każdej funkcji bez wyjątku. Można mianowicie podawać parametry wedle ich deklaracji w prototypie funkcji, czyli od lewej do prawej strony. Wówczas to wywołujący jest w uprzywilejowanej pozycji, gdyż używa bardziej naturalnej kolejności; sama funkcja musi użyć odwrotnej. Drugi wariant to odkładanie parametrów na stos w odwrotnej kolejności niż w deklaracji funkcji; wtedy to funkcja jest w wygodniejszej sytuacji.

Oprócz stosu do przekazywania parametrów można też używać rejestrów procesora, a dokładniej jego czterech rejestrów uniwersalnych. Im więcej parametrów zostanie tam umieszczonych, tym szybsze powinno być (przynajmniej w teorii) wywołanie funkcji.

## Typowe konwencje wywołania

Gdyby każdy programista ustalał własne konwencje wywołania funkcji (co jest teoretycznie możliwe), to oczywiście natychmiast powstałby totalny rozgardiasz w tej materii. Konieczność uwzględniania upodobań innych koderów byłaby z pewnością niezwykle frustrująca. Za sprawą języków wysokiego poziomu nie ma na szczęście aż tak wielkich problemów z konwencjami wywołania. Jedynie korzystając z kodu napisanego w innym języku trzeba je uwzględniać. W zasadzie więc zdarza się to dość często, ale w praktyce cały wysiłek włożony w zgodność z konwencjami ogranicza się co najwyżej do dodania odpowiedniego słowa kluczowego do prototypu funkcji. Często nawet i to nie jest konieczne, jako że prototypy funkcji oferowanych przez przeróżne biblioteki są umieszczane w ich plikach nagłówkowych, zaś zadanie programisty-użytkownika ogranicza się jedynie do włączenia tychże nagłówków do własnego kodu.

Kompilator wykonuje zatem sporą część pracy za nas. Warto jednak przynajmniej znać te najczęściej wykorzystywane konwencje wywołania, a nie jest ich wcale aż tak dużo. Poniższa lista przedstawia je wszystkie:

- cdecl - skrót od C declaration ('deklaracja C') . . Zgodnie z nazwą jest to domyślna

konwencja wywołania w językach C i C++. W Visual C++ można ją jednak jawnie określić poprzez słowo kluczowe `__cdecl`. Parametry są w tej konwencji przekazywane na stos w kolejności od prawej do lewej, czyli odwrotnie niż są zapisane w deklaracji funkcji

- `stdcall` - skrót od Standard Call ('standardowe wywołanie'). Jest to konwencja zbliżona do `cdecl`, posługuje się na przykład tym samym porządkiem odkładania parametrów na stos. To jednocześnie niepisany standard przy pisaniu kodu, który w skompilowanej formie będzie używany przez innych. Korzysta z niego więc chociażby system Windows w swych funkcjach API. Wasze programy powinny korzystać z tej konwencji w przypadku wyposażenia ich np. w system wtyczek. Konwencji tej odpowiada słowo `__stdcall`
- `fastcall` ('szybkie wywołanie') jest, jak nazwa wskazuje, zorientowany na szybkość działania. Dlatego też w miarę możliwości używa rejestrów procesora do przekazywania parametrów funkcji. Zazwyczaj tą konwencję oznacza się poprzez słówko `__fastcall`
- `pascal` budzi słuszne skojarzenia z popularnym ongiś językiem programowania. Konwencja ta była w nim wtedy intensywnie wykorzystywana, lecz dzisiaj jest już przestarzała i coraz mniej kompilatorów (wszelkich języków) wspiera ją
- `thiscall` to specjalna konwencja wywoływania metod obiektów w języku C++. Funkcje wywoływane z jej użyciem otrzymują dodatkowy parametr, będący wskaźnikiem na obiekt danej klasy. Nie występuje on na liście parametrów w deklaracji metody, ale jest dostępny poprzez słowo kluczowe `this`. Oprócz tej szczególnej właściwości `thiscall` jest identyczna z `stdcall`. Ze względu na specyficzny cel istnienia tej konwencji, nie ma możliwości zadeklarowania zwykłej funkcji, która by jej używała.

A zatem dotychczas (nieświadomie!) używaliśmy tylko dwóch konwencji: `cdecl` dla zwykłych funkcji oraz `thiscall` dla metod obiektów.

To zadziwiające, że chyba najważniejsza dla programisty cecha funkcji, czyli jej nazwa, jest niemal zupełnie nieistotna dla działającej aplikacji!. Jak już bowiem mówiłem, "widzi" ona swoje funkcje wyłącznie poprzez ich adresy w pamięci i przy pomocy tych adresów ewentualnie wywołuje owe funkcje. Można dywagować, czy to dowód na całkowity brak skrzyżowania między drogami człowieka i maszyny, ale fakt pozostaje faktem, zaś jego przyczyna jest prozaicznie pragmatyczna. Chodzi tu po prostu o wydajność: skoro funkcje programu są podczas jego uruchamiania umieszczane w pamięci operacyjnej (można ładnie powiedzieć: mapowane), to dlaczego system operacyjny nie miałby używać wygenerowanych przy okazji adresów, by w razie potrzeby rzeczzone funkcje wywoływać? To przecież proste i szybkie rozwiązanie, naturalne dla komputera i niewymagające żadnego wysiłku ze strony programisty. A zatem jest ono po prostu dobre :)

Jedynie w czasie kompilacji kodu nazwy funkcji mają jakieś znaczenie. Kompilator musi bowiem zapewnić ich unikalność w skali całego projektu, tj. wszystkich jego modułów. Nie jest to wcale proste, jeżeli przypomnimy sobie o funkcjach przeciążanych, które z założenia mają te same nazwy. Poza tym funkcje o tej samej nazwie mogą też występować w różnych zakresach: jedna może być na przykład metodą jakiejś klasy, zaś druga zwyczajną funkcją globalną.

Kompilator rozwiązuje te problemy, stosując tak zwane dekorowanie nazw. Wykorzystuje po prostu dodatkowe informacje o funkcji (jej prototyp oraz zakres, w którym została zadeklarowana), by wygenerować jej niepowtarzalną, wewnętrzną nazwę. Zawiera ona wiele różnych dziwnych znaków w rodzaju @ , , ! czy - , dlatego właśnie jest określana jako nazwa dekorowana. Wywołania z użyciem takich nazw są umieszczane w skompilowanych modułach. Dzięki temu linker może bez przeszkód połączyć je wszystkie w jeden plik wykonywalny całego programu.

Ogromna większość funkcji nie może obyć się bez dodatkowych danych, przekazywanych im przy wywoływaniu. Pierwsze strukturalne języki programowania nie oferowały żadnego wspomaganie w tym zakresie i skazywały na korzystanie wyłącznie ze zmiennych globalnych. C++ jednak taki stary nie jest (o czym doskonale wicie). Przyjrzyjmy się zatem dokładniej, jak wywoływane są funkcje z parametrami.

Aby wywołać funkcję z parametrami, kompilator musi znać ich liczbę oraz typ każdego z nich. Informacje te podajemy w prototypie funkcji, zaś w jej kodzie zwykle nadajemy także nazwy poszczególnym parametrom, by móc z nich później korzystać. Parametry pełnią rolę zmiennych lokalnych w bloku funkcji - z tą jednak różnicą, że ich początkowe wartości pochodzą z zewnątrz , od kodu wywołującego funkcję. Na tym wszakże kończą się wszelkie odstępstwa, ponieważ parametrów możemy używać identycznie, jak gdyby było one zwykłymi zmiennymi odpowiednich typów. Po zakończeniu wykonywania funkcji są one niszczone, nie pozostawiając żadnego śladu po ewentualnych operacjach, które mogły być na nich dokonywane kodzie funkcji. Z tego wynika prosty wniosek: Parametry funkcji są w C++ przekazywane przez wartości. Zawsze przez wartości .... niezależnie od tego co pisałem wcześniej o przekazywaniu przez referencję czy o wskaźnikach. W obu wspomnianych wypadkach reguła przekazywania przez wartości jest tylko pozornie łamania. To złudzenie. W rzeczywistości także i tutaj do funkcji są przekazywane wyłącznie wartości - tyle tylko, że owymi wartościami są tu adresy odpowiednich komórek w pamięci. Za ich pośrednictwem możemy więc uzyskać dostęp do rzeczonych komórek, zawierających na przykład jakieś zmienne. Gdy dodatkowo korzystamy z referencji, wtedy nie wymaga to nawet specjalnej składni. Trzeba być jednak świadomym, że zjawiska te dotyczą samej natury wskaźników czy też referencji, nie zaś parametrów funkcji! Dla nich bowiem zawsze obowiązuje przytoczona wyżej zasada przekazywania poprzez wartość.





## Część II Programowanie obiektowe



**KAPITAŁ LUDZKI**  
NARODOWA STRATEGIA SPÓJNOŚCI

**UNIA EUROPEJSKA**  
EUROPEJSKI  
FUNDUSZ SPOŁECZNY





## Wstęp

Druga część naszego podręcznika będzie poświęcona programowaniu obiektowemu i programowaniu zorientowanemu obiektowo. Dla wielu z Was oba te określenia oznaczają to samo ... i tu się mylicie.

Programowanie obiektowe wykorzystuje jedynie pojęcie obiektu, i związaną z tym hermetyzację kodu i ochronę zmiennych. W programowaniu zorientowanym obiektowo mamy do czynienia z czymś więcej – tworzymy hierarchie klas i obiektów, definiujemy ich wzajemne zależności, zmieniamy zachowanie klas w zależności od typów, itp. ... dużo nowych i ciekawych zagadnień przed Wami.



# Rozdział 5

## Programowanie obiektowe

Autorzy:

*Michał Syfert   Paweł Wnuk*

### 5.1 Czym jest programowanie obiektowe?

Skoro mówimy o programowaniu zorientowanym obiektowo, potrzebna jest jakaś definicja obiektu. Zamieścimy Wam jedną:

#### Definicja 5.1

Obiekt to byt świata materialnego lub niematerialnego, który potrafimy wyizolować ze środowiska.

Straszne .... to może inna?

#### Definicja 5.2

Obiekt - struktura danych, występująca łącznie z operacjami dozwolonymi do wykonania na niej.

Ładniejsza? Może ... a która z nich jest prawidłowa?

Obie. W zależności od punktu widzenia klasy i obiekty możemy traktować zarówno jak elementy inżynierii oprogramowania (definicja 1) jak i jako elementy języka programowania (definicja 2). W poprzednim rozdziale (który głównie był przypomnieniem i rozszerzeniem znanej już Wam składni) raczej niewiele zajmowaliśmy się inżynierią, i tłumaczeniem zasad działania pewnych konstrukcji językowych. Teraz to się zmieni. Postaramy się przedstawić Wam w miarę kompletne wprowadzenie do programowania obiektowego i zorientowanego obiektowo.

Na początek chcielibyśmy wyraźnie wprowadzić rozróżnienie pomiędzy dwoma określeniami uważanymi za fundamentalne przy programowaniu obiektowym, a przez początkujących często nie do końca prawidłowo rozróżnianymi.

### 5.1.1 Obiekty i klasy

Zacznijmy od tego, że w przypadku programowania obiektowego program składa się z obiektów, które wchodzi w interakcje, przysyłając sobie nawzajem komunikaty. Wyobraźcie sobie obiekt jako żywą istotę. Obiekty mają coś, co wiedzą (atributy), oraz coś, co mogą zrobić (zachowania lub operacje). Wartości atrybutów obiektu określają jego stan, poprzez wywoływanie zachowań obiekty przechodzą od stanu do stanu.

Klasy to "plany" obiektów. Klasa łączy atrybuty (dane) oraz zachowania (metody lub funkcje) w jedną odrębną jednostkę, opisuje ją syntaktycznie i definiuje zachowania. Z tego punktu widzenia obiekty są egzemplarzami (instancjami) klas.

## Obiekt

Mówiąc bardziej formalnie, obiekt jest strukturą danych, występującą łącznie z operacjami dozwolonymi do wykonania na niej (do tej pory zajmowaliście się strukturami, definiowanymi przez struct które nie miały bezpośredniej łączności składniowej z dozwolonymi operacjami). Obiekt może być złożony, a więc może składać się z innych obiektów. Każdy obiekt ma przypisany typ <link>, tj. wyrażenie językowe, które określa jego budowę (poprzez specyfikację atrybutów) oraz ogranicza kontekst, w którym odwołanie do obiektu może być użyte w programie.

Obiekt może być powiązany z innymi obiektami związkami skojarzeniowymi, odpowiadającymi relacjom zachodzącym między odpowiednimi bytami w dziedzinie problemowej.

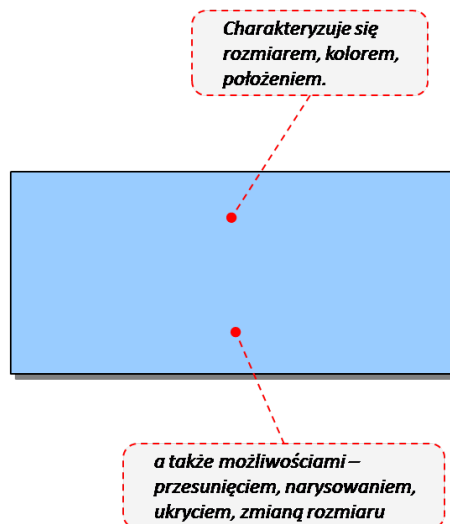
Obiekt jest charakteryzowany poprzez:

- Tożsamość, która odróżnia go od innych obiektów.
- Stan, który może zmieniać się w czasie (bez zmiany tożsamości obiektu). Stan obiektu w danym momencie jest określany przez aktualne wartości atrybutów i powiązań z innymi obiektami;

## Klasa

Klasa to szablon obiektu - jest miejscem przechowywania tych własności grupy obiektów, które są niezmiennie dla wszystkich członków grupy. Dobrze zbudowana klasa jest starannie wydzieloną abstrakcją pochodzącą ze słownictwa dziedziny danego problemu lub rozwiązania. Obejmuje pewien mały, dobrze określony zbiór zobowiązań, z których jest w stanie się w pełni wywiązać. Zapewnia oddzielenie specyfikacji abstrakcji od jej implementacji. Jest zrozumiała i prosta, a przy tym rozszerzalna i dająca się łatwo dostosować do potrzeb (ech ... piękna idea ;) )

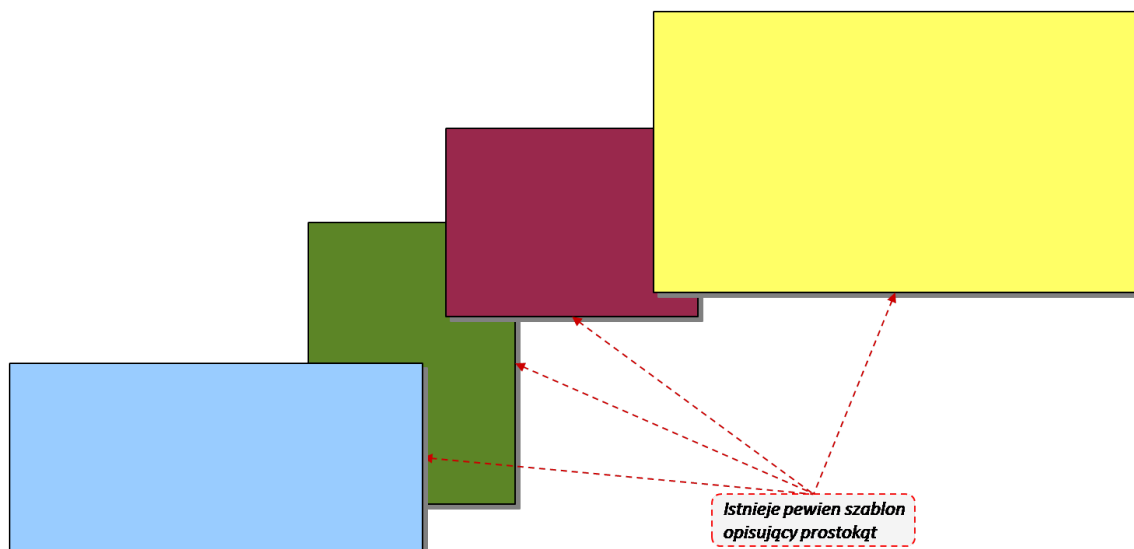
To wszystko brzmi ciągle nieco abstrakcyjnie, więc może prosty przykład: Wyobraźcie sobie prostokąt:



**Rysunek 5.1** – Prostokąt jako obiekt

Prostokąt ma pola różnego typu logicznie powiązane ze sobą – rozmiar, kolor położenie. Powiązaniem logicznym jest to, że i rozmiar, i położenie opisują ten jeden konkretny prostokąt. Prostokąt może również pewne czynności “wykonać” - narysować się, przesunąć, zmienić rozmiar. I to jest właśnie obiekt.

Jeśli natomiast weźmiemy pod uwagę wiele prostokątów:



**Rysunek 5.2** – Prostokąt jako klasa

Możemy zauważyć, że każdy z nich ma kolor (inny co do wartości, lecz tego samego typu), każdy ma rozmiar (podobnie – typ ten sam, różne wartości), każdy możemy narysować (mechanizm jest ten sam, różni się jedynie kolor, pozycja gdzie jest rysowany, rozmiar, itp.). Czyli istnieje pewien szablon, pozbawiony tożsamości i stanu opis prostokąta, nie jednego konkretnego, lecz każdego możliwego prostokąta. I to jest klasa. W przypadku konkretnego prostokąta możemy powiedzieć, że jest niebieski. W przypadku klasy prostokątów – możemy powiedzieć że ma kolor.

## 5.2 Deklaracje i definicje klas

Definicja klasy jest podobna do definicji struktury, musi tylko uwzględniać metody. W przypadku najprostszych klas, w których nie jest potrzebne definiowanie widoczności (dostępu do) poszczególnych pól czy też metod, możemy pozostać przy znanym już słowie kluczowym `struct` lub używać go zamiennie ze słowem `class` (wspominaliśmy, że struktura jest zdegenerowaną klasą w C++), wykorzystanie wszystkich możliwości klas w C++ wymaga już stosowania słowa kluczowego `class`.

Na początek prosty przykład: założmy, że chcemy zdefiniować klasę będącą bardzo prostą komputerową reprezentacją diody. Przyjmijmy, że dioda, traktowana jako ogólne pojęcie, będzie określona przez dwie właściwości (dwa pola):

- ma jakiś kolor (pole typu `string`)
- jest zapalona albo nie (pole typu `bool`)

Na polach tych chcemy móc wykonywać następujące operacje (metody):

- zapalenie diody
- zgaszenie diody
- obserwacja diody (jej stanu: czy świeci i w jakim kolorze).

Klasę opisującą pojęcie diody nazwiemy `CDioda`:

### Kod programu 5.1

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 /* Definicja typu obiektowego - klasy CDioda */
7 class CDioda {
8 public:
9     // definicje pól o nazwach kolor i zapalona
10    string kolor;
11    bool zapalona;
12    // definicja metody zapal
13    void zapal() {
14        zapalona = true;
15    }
16    // definicja metody zgas
17    void zgas() {
18        zapalona = false;
19    }
20    // definicja metody pokaz
21    void pokaz() {
22        if (zapalona)
23            cout << "Swieci w kolorze " << kolor << endl;
24        else
25            cout << "Nie swieci\n";
```

```

26 }
27 };
28
29 int main(int argc, char *argv[])
30 {
31     cout << "Diody ..." << endl;
32
33     // utworzenie dwóch obiektów typu CDioda, o nazwach d1 i d2
34     CDioda d1, d2;
35
36     // ustawienie wartości ich pól
37     d1.kolor = "zielony";
38     d2.kolor = "czerwony";
39     d1.zapalona = false;
40     d2.zapalona = false;
41
42     // główna pętla programu - będzie prosić użytkownika
43     // o podanie działania - i po każdym poleceniu wyświetlać
44     // stan diod
45     char zn;
46     do {
47         cout << "Stan diod:\n";
48         d1.pokaz();
49         d2.pokaz();
50         cout << "\nCo chcesz zrobic?\n1 - zapal diode 1\n";
51         cout << "2 - zgas diode 1\n";
52         cout << "3 - zapal diode 2\n";
53         cout << "4 - zgas diode 2\n";
54         cout << "0 - zakoncz program\n";
55         cin >> zn;
56         switch (zn) {
57             case '1' : d1.zapal(); break;
58             case '2' : d1.zgas(); break;
59             case '3' : d2.zapal(); break;
60             case '4' : d2.zgas(); break;
61         };
62     } while (zn != '0');
63
64     return 0;
65 }

```

No tak ... uważny czytelnik może stwierdzić - tyle pisania, tyle teorii, tyle hałasu, a jedyny zysk to fakt, że zamiast pisać `pokaz(d1)` piszemy `d1.pokaz()`. I będzie miał rację - jak na razie ... przynajmniej dopóki nie pokażemy możliwości ochrony pól i metod w obiekcie. A prawdziwą potęgę programowania obiektowego zobaczycie jak omówimy zupełnie nowe pojęcia - dziedziczenie i polimorfizm.

### 5.2.1 Składnia

Przykład podany przez nas przed chwilą jest prawidłowy, lecz nie do końca elegancki. Prawidłowy – czyli zgodny ze składnią, którą można podać następująco:

#### Kod programu 5.2

```

1 class nazwa_klasy
2 {

```

```
3 // część publiczna
4 public:
5 // pola różnych typów
6 typ_pola nazwa_pola, ...nazwa_pola;
7 ...
8 // metody (z parametrami lub bez) operujące na tych polach
9 typ_zwracany nazwa_metody(lista_parametrów);
10 ...
11 // znowu mogą być inne pola
12 typ_pola nazwa_pola, ...nazwa_pola;
13 ...
14 // i inne metody
15 typ_zwracany nazwa_metody(lista_parametrów);
16 // metody z definicją:
17 typ_zwracany nazwa_metody(lista_parametrów) {
18 // instrukcje (ciało) metody;
19 }
20 // i tak dalej...
21 // część chroniona
22 protected:
23 // to samo co w public
24 ...
25 // część prywatna
26 private:
27 // to samo co w public
28 ...
29 }; // koniec deklaracji
30
31 ...
32 //definicje metod:
33 typ_zwracany nazwa_klasy::nazwa_metody(lista_parametrów) {
34 ...
35 };
```

W C++, mimo że możliwa jest jednoczesna definicja i deklaracja klasy - zwykle postępuje się inaczej. Każdą klasę rozбивa się na dwa pliki:

- plik z deklaracją (a nie definicją) klasy
- plik zawierający implementację klasy, czyli definicje jej metod.

Klasę deklaruje się w pliku nagłówka (\*.h), natomiast definicje metod wchodzących w jej skład umieszcza w implementacji (\*.cpp).

Implementacje (definicje) kolejnych metod tworzą tzw. wnętrze obiektu, i należą w całości do obiektów danej klasy. W implementacji metody konieczne jest więc zastosowanie desygatora (oznacznika), określającego, do jakiej klasy należy dana metoda. Za desygatorem umieszcza się podwójny dwukropek, a dopiero po nim nazwę metody. Pisząc ciało (treść) funkcji będącej metodą jakiejś klasy, możemy odwoływać się do pól tej klasy bezpośrednio, nie podając nazwy klasy, do której dane pole należy - wewnątrz klasy wszystkie pola są znane i bezpośrednio dostępne (to tak jak my - na polecenie "rusz swoją ręką" - wiemy, która ręka jest nasza). Inaczej mówiąc - wewnątrz metody wszystkie pola klasy można traktować jak zdefiniowane zmienne lokalne.

To może klasa z przykładu wprowadzającego, tym razem rozbita już na poszczególne pliki. Zaczniemy od nagłówka:

### Kod programu 5.3

```
1 #ifndef cdiodaH
2 #define cdiodaH
3
4 #include <string>
5 using namespace std;
6
7 /* Deklaracja klasy CDioda */
8 class CDioda {
9 public:
10 // pole pamiętające stan diody
11 bool zapalona;
12 // kolor diody
13 string kolor;
14 // informacja, że dioda ma metodę zapal
15 void zapal();
16 // informacja, że dioda ma metodę zgas
17 void zgas();
18 // informacja, że dioda ma metodę pokaz
19 void pokaz();
20 };
21
22 #endif
```

I następnie definicja metod klasy (przykładowe wykorzystanie sobie darujemy – niczym się nie różni od kodu wykorzystanego w programie wprowadzającym):

### Kod programu 5.4

```
1 #include <iostream>
2 #include <cstdlib>
3
4 #include "cdioda.h"
5
6 using namespace std;
7
8 // definicje metod
9 void CDioda::zapal() {
10     zapalona = true;
11 }
12
13 void CDioda::zgas() {
14     zapalona = false;
15 }
16
17 void CDioda::pokaz() {
18     if (zapalona)
19         cout << "Swieci w kolorze " << kolor << endl;
20     else
21         cout << "Nie swieci\n";
22 }
```

Teraz wreszcie mamy klasę zakodowaną zgodnie ze wszystkimi zasadami ....



## 5.3 Ochrona danych w klasach

Najczęściej do korzystania z obiektów nie potrzebujemy głębokiej wiedzy i wszystkich tajników ich implementacji – by jeździć samochodem nie muszę znać i rozumieć procesu spalania paliwa w cylindrach, zasad działania skrzyni biegów czy też nie muszę wiedzieć z ilu kół zębatych wspomniana skrzynia jest zbudowana. Popatrzcie na klasę `CDIODA` – co prawda ciężko w tym przypadku mówić o jakichkolwiek tajnikach implementacji, lecz w praktyce do jej wykorzystania wystarczy nam wiedza, jak wywołać trzy metody. Wartości pól nie musimy znać, co więcej – nawet nie musimy wiedzieć że jakieś pola istnieją. Ponoć obciążanie sobie głowy niepotrzebną wiedzą jest niezdrowe – więc po co mamy wiedzieć, **jak działa** obiekt danej klasy, skoro wystarczy nam wiedzieć **jak z niego skorzystać**?

Mówiąc bardziej poważnie – ochrona danych to nie tylko wygoda, to także technika, która pozwala nam tak przygotować klasę, by żaden obiekt będący jej instancją w żadnym momencie swojego życia nie znalazł się w stanie niedozwolonym. Przykładowo – jeśli twierdzimy, że czarny kot nie ma prawa bytu, możemy zdefiniować klasę kota, w której będzie pole kolor. I dla tego pola wartość „czarny” będzie niedozwolona, a mechanizmy klasy zapewnią, że nigdy żaden jej użytkownik nie będzie mógł przypisać wartości „czarny” polu kolor.

Fachowo takie mechanizmy nazywa się ochroną pól lub metod. Ochrona oznacza, że autor danej klasy ma możliwość zdefiniowania obszaru widoczności jej pola lub poprzez odpowiednie jej zdefiniowanie. Standard programowania obiektowego (o ile można o czymś takim mówić) przewiduje występowanie trzech typów ochrony zmiennych:

- Pola i metody mogą być publiczne (ang. *public*), co oznacza, że są one dostępne z dowolnego miejsca w programie – nie są chronione. W C++ jest to zachowanie domyślne i jednocześnie jedyne dozwolone dla klasy definiowanej przy pomocy słowa kluczowego `struct`.
- Jeśli chcemy uniemożliwić jakikolwiek dostęp do zmiennej lub metody spoza implementacji metod wchodzących w jej skład, deklarujemy ją jako prywatną (ang. *private*). W C++ jest to zachowanie domyślne dla klasy definiowanej przy pomocy słowa kluczowego `class` (jeśli nie zadeklarujemy inaczej, wszystkie pola i metody będą prywatne).
- Trzecim typem są pola i metody chronione (ang. *protected*) - dostępne podczas implementacji danego obiektu i wszystkich obiektów pochodnych, dla których zastosowano dziedziczenie publiczne lub chronione.

## 5.4 Deklaracje pól

Pola są właściwą treścią każdego obiektu klasy, to one stanowią jego reprezentację w pamięci operacyjnej. Pod tym względem nie różnią się niczym od znanych ci już pól w strukturach i są po prostu zwykłymi zmiennymi, zgrupowanymi w jedną, kompleksową całość. Jako miejsce na przechowywanie wszelkiego rodzaju danych, pola mają kluczowe znaczenie dla obiektów i dlatego powinny być chronione przez niepowołanym dostępem z zewnątrz. Przyjęło się więc, że w zasadzie wszystkie pola w klasach deklaruje się jako

prywatne. Zalecam gorąco trzymania się tej zasady – to będzie sprawdzane w Waszych projektach ;>

By jakoś odróżnić pola od zmiennych lokalnych, stosuje się różne konwencje nazywania jednych i drugich. Konwencji jest wiele – jedni za dawną firmą Borland wszystkie pola w klasie nazywają zaczynając od litery F, inni zaczynają nazwę pola dużą literą a zmiennej lokalnej małą, jeszcze inni stosują podkreślenie ... wybrać można co chcecie – ważne by konsekwentnie trzymać się swojego wyboru.

Dostęp do danych zawartych w polach musi się zatem odbywać za pomocą dedykowanych metod. Rozwiązanie to ma wiele rozlicznych zalet: pozwala chociażby na tworzenie pól, które można jedynie odczytywać, daje sposobność wykrywania niedozwolonych wartości (np. indeksów przekraczających rozmiary tablic itp.) czy też podejmowania dodatkowych akcji podczas operacji przypisywania. Takie funkcje zwyczajowo nazywa się setterami / getterami, i też oznacza w jakiś sposób. Mi osobiście najbardziej odpowiada konwencja nazwania funkcji zwracającej tak samo jak pola, natomiast metody ustawiającej znów nazwą pola, lecz tym razem poprzedzonej słówkiem set. Czyli przykładowo:

### Kod programu 5.5

```
1 class Cmoja {  
2 public:  
3 double wartosc() { return FWartosc; }  
4 void setWartosc(double _v) { FWartosc = _v; }  
5 private:  
6 double FWartosc;  
7 };
```

## 5.5 Metody

Z praktycznego punktu widzenia metody niewiele różnią się od zwyczajnych funkcji - oczywiście poza faktem, iż są deklarowane zawsze wewnątrz jakiejś klasy. Deklaracje te mogą mieć formę prototypów funkcji, a stworzone w ten sposób metody wymagać jeszcze implementacji , czyli wpisania ich kodu. Jak wspomniałem – zazwyczaj podaje się go w pliku implementacji. Warto jednak wiedzieć, że wprowadzanie kodu metod bezpośrednio wewnątrz bloku class sprawi, że kompilator traktuje takie funkcje jako inline , tzn. rozwijane w miejscu wywołania, i wstawia cały ich kod przy każdym odwołaniu się do nich. Dla krótkich, jednolinijskich metod jest to dobre rozwiązanie, przyspieszające działanie programu. Dla dłuższych nie musi wcale takie być – będzie od tego puchł Wasz exe-c ;>

To jeszcze nie koniec zabawy z metodami :) Niektóre z nich można mianowicie uczynić stałymi . Zabieg ten sprawia, że funkcja, na której go zaaplikujemy, nie może modyfikować żadnego z pól klasy , a tylko je co najwyżej odczytywać. Po co komu takie udziwnienie? Teoretycznie jest to pewna wskazówka dla kompilatora, który być może uczyni nam w zamian łaskę poczynienia jakichś optymalizacji. Praktycznie jest to też pewien sposób na zabezpieczenie się przed omyłkowym zmodyfikowaniem obiektu w metodzie, która wcale nie miała czegoś takiego robić. Jednym słowem korzyści są piorunujące ;)

Uczynienie jakiejś metody stałą jest banalnie proste: wystarczy tylko dodać za listą jej parametrów magiczne słówko const , np.:

### Kod programu 5.6

```
1 class CFoo
2 {
3 private :
4 int FPole;
5 public :
6 int Pole() const { return m_nPole; }
7 };
```

Funkcja Pole() (będąca de facto obudową dla zmiennej FPole ) będzie tutaj słusznie metodą stałą.

## 5.5.1 Konstruktor i destruktor

Wspominałem, już parokrotnie o procesie tworzenia obiektów, podkreślając przy tym znaczenie tego procesu. Za chwilę wyjaśni się, dlaczego jest to takie ważne. Decydując się na zastosowanie technik obiektowych w konkretnym programie musimy mieć na uwadze fakt, iż oznacza to zdefiniowane przynajmniej kilku klas oraz instancji tychże. Istotą OOPu jest poza tym odpowiednia komunikacja między obiektami: wymiana danych, komunikatów, podejmowanie działań zmierzających do realizacji danego zdania, itp. Aby zapewnić odpowiedni przepływ informacji, krystalizuje się mniej lub bardziej rozbudowana hierarchia obiektów , kiedy to jeden obiekt zawiera w sobie drugi, czyli jest jego właścicielem . To dość naturalne: większość otaczających nas rzeczy można przecież rozłożyć na części, z których się składają (gorzej może być z powtórным złożeniem ich w całość :D).

Konsekwencje tego stanu rzeczy dla procesu tworzenia (i niszczenia) obiektów są raczej oczywiste: kreacja obiektu zbiorczego musi pociągnąć za sobą stworzenie jego składników; podobnie jest też z jego destrukcją. Jasne, można te kwestie zostawić kompilatorowi, ale paradoksalnie czyni to kod trudniejszym do zrozumienia, pisania i konserwacji.

C++ oferuje nam na szczęście możliwość podjęcia odpowiednich działań zarówno podczas tworzenia obiektu, jak i jego niszczenia. Korzystamy z niej, wprowadzając do naszej klasy dwa specjalne rodzaje metod - są to tytułowe konstruktory oraz destruktory – a konkretniej – destruktor. O ile konstruktorów dla danej klasy może być wiele, o tyle destruktor jest zawsze jeden.

Konstruktor to specyficzna funkcja składowa klasy, wywoływana zawsze podczas tworzenia należącego doń obiektu. Typowym zadaniem konstruktora jest zainicjowanie pól ich początkowymi wartościami, przydzielenie pamięci wykorzystywanej przez obiekt czy też uzyskanie jakichś kluczowych danych z zewnątrz. Deklaracja konstruktora jest w C++ bardzo prosta. Metoda ta nie zwraca bowiem żadnej wartości (nawet void !), a jej nazwa odpowiada nazwie zawierającej ją klasy. Zazwyczaj też konstruktor nie przyjmuje żadnych parametrów, co nie znaczy jednak, że nie może tego czynić. Często są to na przykład startowe dane przypisywane do pól. Co więcej, możliwe są różne postacie konstruktora (przeciążanie) które mogą posłużyć np. do wykonania kopii obiektu przekazanego jako wzorzec.

### Kod programu 5.7

```
1 class CProstokat
2 {
```

```

3 public:
4   // konstruktor 1 - domyślny
5   CProstokat();
6   // konstruktor 2 - kopiujący
7   CProstokat(const CProstokat& wzor);
8   // destruktor
9   ~CProstokat();
10  ...
11 };
12
13 CProstokat::CProstokat()
14 {
15   x1 = y1 = 0;
16   x2 = 1;
17   ...
18 }
19
20 CProstokat::CProstokat(const CProstokat& wzor)
21 {
22   x1 = wzor.x1;
23   y1 = wzor.y1;
24   ...
25 }
26
27 CProstokat::~~CProstokat()
28 {
29 }
30
31 void f(CProstokat p) { ... }
32
33 // konstr. 1 przed uruchomieniem programu
34 CProstokat pr;
35 // konstr. 1 przed uruchomieniem programu
36 CProstokat pr2;
37
38 // konstr. 2
39 pr2 = pr;
40
41 // konstr. 1 po dojściu do tej linii
42 CProstokat *pr3 = new CProstokat();
43 // konstr. 2 po dojściu do tej linii
44 CProstokat *pr3 = new CProstokat(pr);
45
46 // wywołany zostanie destruktor
47 delete pr3
48
49 // konstr. 2 przed wejściem do f
50 f(pr);
51 // destruktor
52
53 // konstr. 2 po przed wejściem do f
54 f(*pr);
55 // destruktor

```

Obiekty zawsze są tworzone przy wykorzystaniu konstruktora. Nawet jeśli nie wywoła się go jawnie, w C++ zostanie wywołany w sposób niejawny (widzicie to w kodzie powyżej). I analogicznie - jeśli nie zostanie w klasie zdefiniowany żaden konstruktor, kompilator

wygeneruje sam jego domyślną wersję - która nic nie robi. Podobnie wygeneruje trywialny płytki konstruktor kopiujący.

Z wiadomych względów konstruktory czynimy prawie zawsze metodami publicznymi. Umieszczenie ich w sekcji `private` dałoby bowiem dość dziwny efekt: taka klasa nie mogłaby być normalnie instancjowana, tzn. niemożliwe byłoby utworzenie z niej obiektu w zwykły sposób. Potem przekonacie się, że to może mieć sens ... ;)

OK, konstruktory mają zatem niebagatelną rolę, jaką jest powoływanie do życia nowych obiektów. Doskonale jednak wiemy, że nic nie jest wieczne i nawet najdłużej działający program kiedyś będzie musiał być zakończony, a jego obiekty zniszczone. Tą niechlubną robotą zajmuje się kolejny, wyspecjalizowany rodzaj metod – destruktor. Destruktor jest specjalną metodą, przywoływaną podczas niszczenia obiektu zawierającej ją klasy.

W naszych przykładowych klasach destruktor nie miałby wiele do zrobienia - zgoła nic, ponieważ żaden z prezentowanych obiektów nie wykonywał czynności, po których należałoby sprzątać. To się však niedługo zmieni, zatem poznanie destruktorów z pewnością nie będzie szkodliwe :) Postać destruktora jest także niezwykle prosta i w dodatku zawsze identyczna. Funkcja ta nie bierze bowiem żadnych parametrów (bo i jakie miałyby brać?) i niczego nie zwraca. Jej nazwą jest zaś nazwa zawierającej klasy poprzedzona znakiem tyldy ( `~` ).

Coś jeszcze?

Pola, zwykłe metody oraz konstruktory i destruktory to zdecydowanie najczęściej spotykane i chyba najważniejsze elementy klas. Aczkolwiek nie jedyne; w dalszej części tego kursu poznamy jeszcze składowe statyczne, funkcje przeciążające operatory oraz tzw. deklaracje przyjaźni (naprawdę jest coś takiego! :D). Poznane tutaj składniki klasy będą jednak zawsze miały największe znaczenie. Można jeszcze wspomnieć, że wewnątrz klasy (a także struktury i unii) możemy zdefiniować kolejną klasę! Taką definicję nazywamy wtedy zagnieżdżoną. Technika ta nie jest stosowana zbyt często, ale jest dostępna. Podobnie zresztą jest z innymi typami, określanymi poprzez `enum` czy `typedef`.

W C++ nie istnieje formalny wymóg definicji konstruktorów czy destruktorów dla każdego typu obiektu, jednak - dobry styl wymaga, aby konstruktor zawsze był, choćby w celu przypisania zmiennym wartości domyślnych. Zabezpiecza nas to przed generowaniem wersji domyślnych, które, co tu dużo pisać – są po prostu głupie ... Przykładowo - kopiowanie jest płytkie – kopiowany jest wskaźnik a nie wskazywana wartość. Co w praktyce oznacza, że poniższy programik spowoduje ulubiony błąd programisty C++ - `access violation` ;>

### Kod programu 5.8

```
1 {
2 public:
3   CMoja() {
4     tbl = new char[255];
5   };
6   ~CMoja() {
7     delete[] tbl;
8   }
9 private:
10  char* tbl;
11 };
12
13 void ff(CMoja m) {
```

```
14 ...
15 }
16
17 int main() {
18     CMoja obiekt;
19     ff(obiekt);
20
21     CMoja *po = new CMoja();
22     ff(obiekt);
23     delete po;
24
25     return 0;
26 }
```

Dlaczego? Mam nadzieję że się domyślacie ...

## 5.6 Implementacja metod

Definicja klasy jest zazwyczaj tylko połową sukcesu i nie stanowi wcale końca jej określania. Dzieje się tak przynajmniej wtedy, gdy umieścimy w niej jakieś prototypy metod, bez podawania ich kodu. Uzupełnieniem definicji klasy jest wówczas jej implementacja, a dokładniej owych prototypowanych funkcji składowych. Polega ona rzecz jasna na wprowadzeniu instrukcji składających się na kod tychże metod w jednym z modułów programu.

Operację tę rozpoczynamy od dołączenia do rzeczzonego modułu pliku nagłówkowego z definicją naszej klasy, np.:

### Kod programu 5.9

```
1 #include "klasa.h"
```

Potem możemy już zająć się każdą z niezaimplementowanych metod; postępujemy tutaj bardzo podobnie, jak w przypadku zwykłych, globalnych funkcji. Składnia metody wygląda analogicznie do składni funkcji, jedyną różnicą jest podanie przed nazwą metody nazwy klasy do której owa metoda należy. Wpisanie jej jest konieczne: po pierwsze mówi ona kompilatorowi, że ma do czynienia z metodą klasy, a nie zwykłą funkcją; po drugie zaś pozwala bezbłędnie zidentyfikować macierzystą klasę danej metody. Między nazwą klasy a nazwą metody widoczny jest operator zasięgu ::, z którym już raz mieliśmy przyjemność się spotkać. Teraz możemy oglądać go w nowej, chociaż zbliżonej roli.

Zaleca się, aby bloki metod dotyczące się jednej klasy umieszczać w zwartej grupie, jeden pod drugim. Czyni to kod lepiej zorganizowanym.

Drugą różnicą jest ewentualny modyfikator `const`, który, jak pamiętamy, czyni metodę stałą. Jego obecność w tym miejscu powinna się pokrywać z występowaniem także w prototypie funkcji. Niezgoda w tej kwestii zostanie srodze ukarana przez kompilator :)

Oczywiście większością implementacji metody będzie blok jej instrukcji, tradycyjnie zawarty między nawiasami klamrowymi. Cóż ciekawego można o nim powiedzieć? Bynajmniej niewiele: nie różni się prawie wcale od analogicznych bloków globalnych funkcji. Dodatkowo jednak ma on dostęp do wszystkich pól i metod swojej klasy - tak, jakby były one jego zmiennymi albo funkcjami lokalnymi.



### 5.6.1 Magiczne słówko `this`

Z poziomu metody mamy dostęp do jeszcze jednej, bardzo ważnej i przydatnej informacji. Chodzi tutaj o obiekt, na rzecz którego nasza metoda jest wywoływana; mówiąc ściśle, o odwołanie ( wskaźnik ) do niego. Cóż to znaczy? Nic ponad bezpośrednie tłumaczenie .... `this` to po polsku „to” lub „ten” ... w ten sposób można jawnie wymusić odwołanie się do własnych pól i metod. Czasem mówi się więc, iż jest to dodatkowy, specjalny parametr metody - występuje przecież w jej wywołaniu. W niektórych językach rola `this` jest znacznie większa niż w C++ - to jedyna technika by odwołać się do własnych pól w klasie pisanej przykładowo w PHP. W C++ nie jest to niezbędne. `This` stosuje się w kilku przypadkach. Większość z nich poznacie później, natomiast teraz ... teraz przedstawię Wam dwa z nich. Pierwszy – to gdy w ciele metody zdefiniujemy zmienną lokalną nazywającą się tak samo jak pole klasy. Wtedy by do pola klasy się odwołać – potrzebujemy `this`. Druga – to coś co podpatrzyłem u studentów. Pisząc w ciele metody `this->` włączamy mechanizm podpowiadania środowiska programistycznego ... i za chwilę za strzałką pojawi nam się lista wszystkich metod i pól danej klasy ;>

#### Kod programu 5.10

1 .

## 5.7 Praca z obiektami

Nawet dziesiątki wyśmienitych klas nie stanowią jeszcze gotowego programu, a jedynie pewien rodzaj reguł, wedle których będzie on realizowany. Wprowadzenie tych reguł w życie wymaga przeto stworzenia obiektów na podstawie zdefiniowanych klas. W C++ mamy dwa główne sposoby ”obchodzenia” się z obiektami; różnią się one pod wieloma względami, inne jest też zastosowanie każdego z nich. Naturalną i rozsądną kolejną rzeczą będzie więc przyjrzenie się im obu :)

Pierwszą strategię znamy już bardzo dobrze, używaliśmy jej bowiem niejednokrotnie nie tylko dla samych obiektów, lecz także dla wszystkich innych zmiennych – tworzymy statyczne zmienne obiektowe. W tym trybie korzystamy z klasy dokładnie tak samo, jak ze wszystkich innych typów w C++ - czy to wbudowanych, czy też definiowanych przez nas samych (jak enum’y, struktury itd.). W tym trybie każde pojawienie się definicji nowej zmiennej, np takiej:

```
CMoja obiekt;
```

Wykonuje jednak znacznie więcej czynności, niż jest to widoczne na pierwszy czy nawet drugi rzut oka. W tym momencie mianowicie:

- wprowadzam nową zmienną obiekt typu `CMoja`. Nie jest to rzecz jasna nowość, ale dla porządku warto o tym przypomnieć.
- tworzę w pamięci operacyjnej obszar, w którym będą przechowywane pola obiektu. To także nie jest zaskoczeniem: pola, jako bądź co bądź zmienne, muszą rezydować gdzieś w pamięci, więc robią to w identyczny sposób jak pola struktur.



- wywołuję domyślny konstruktor klasy CMoja (czyli metodę CMoja::CMoja() ), by dokończył aktu kreacji obiektu. Po jego zakończeniu możemy uznać nasz obiekt za ostatecznie stworzony i gotowy do użycia.

Te trzy etapy są niezbędne, abyśmy mogli bez problemu korzystać z stworzonego obiektu. W tym przypadku są one jednak realizowane całkowicie automatycznie i nie wymagają od nas żadnej uwagi. Przekonamy się później, że nie zawsze tak jest i, co ciekawe, wcale nie będziemy tym zmartwieni :D. Muszę jeszcze wspomnieć o pewnym drobnym wymaganiu, stawianym nam przez kompilator, któremu chcemy podać wiersz kodu umieszczony na początku paragrafu. Otóż klasa CMoja musi tutaj posiadać bezparametrowy konstruktor , albo też nie mieć wcale procedury tego rodzaju (wtedy kompilator sam wygeneruje jej „dummy” wersję).

W innym przypadku potrzebne jest jeszcze przekazanie odpowiednich parametrów konstruktorowi, który takowych wymaga. Konieczność tą realizujemy podobną metodą, co wywołanie zwyczajnej funkcji. Zakładając, że CMoja posiada konstruktor przyjmujący jedną liczbę całkowitą, oraz napis, możliwe jest wywołanie:

```
CMoja moja( 10 , "jakiś tekst" );
```

Zadeklarowane przed chwilą zmienne obiektowe są w istocie takimi samymi zmiennymi, jak wszystkie inne w programach C++. Możliwe jest zatem przeprowadzanie nań operacji, którym podlegają na przykład liczby całkowite, napisy czy tablice. Nie mam tu wcale na myśli jakichś złożonych manipulacji, wymagających skomplikowanych algorytmów, lecz całkiem zwyczajnych i codziennych, jak przypisanie czy przekazywanie do funkcji. Czy można powiedzieć cokolwiek ciekawego o tak trywialnych czynnościach? Okazuje się, że tak. Zwrócimy wprawdzie uwagę na dość oczywiste fakty z nimi związane, lecz znajomość owych "banałów" okaże się później niezwykle przydatna.

Na użytek dalszych wyjaśnień wróćmy i nieco rozszerzmy początkową definicję diody:

### Kod programu 5.11

```
1  #ifndef cdiodaH
2  #define cdiodaH
3
4  #include <string>
5  using namespace std;
6
7  /* Deklaracja klasy CDioda */
8  class CDioda {
9  public:
10     /// konstruktor podstawowy
11     CDioda();
12     /// konstruktor z ustawieniem koloru
13     CDioda(string _kolor);
14     /// metoda włączająca diodę
15     void zapal();
16     /// metoda wyłączająca diodę
17     void zgas();
18     /// metoda wyświetlająca stan diody
19     void pokaz();
20 private:
21     string kolor;
22     bool zapalona;
```

```
23 };  
24  
25 #endif
```

### Kod programu 5.12

```
1  #include <iostream>  
2  #include <cstdlib>  
3  
4  #include "cdioda.h"  
5  
6  using namespace std;  
7  
8  CDioda::CDioda() {  
9      kolor = "bialy";  
10     zapalona = false;  
11 }  
12 CDioda::CDioda(string _kolor) {  
13     kolor = _kolor;  
14     zapalona = false;  
15 }  
16  
17 void CDioda::zapal() {  
18     zapalona = true;  
19 }  
20  
21 void CDioda::zgas() {  
22     zapalona = false;  
23 }  
24  
25 void CDioda::pokaz() {  
26     if (zapalona)  
27         cout << "Swieci w kolorze " << kolor << endl;  
28     else  
29         cout << "Nie swieci\n";  
30 }
```

Natychmiast też zadeklarujemy i stworzymy dwa obiekty należące do naszej klasy:

### Kod programu 5.13

```
1 CDioda dioda1("czerwona"), dioda2("zielona");
```

Tym sposobem mamy więc diody, sztuk dwie, w kolorze czerwonym oraz zielonym. Moglibyśmy użyć ich metod, aby je obie włączyć; zrobimy jednak coś dziwniejszego - przypiszemy jedną lampę do drugiej:

### Kod programu 5.14

```
1 dioda1=dioda2;
```

” A co to za dziwadło?”, słusznie pomyślisz. Taka operacja jest jednak całkowicie poprawna i daje dość ciekawe rezultaty. By ją dobrze zrozumieć musimy pamiętać, że dioda1 oraz dioda2 są to przede wszystkim zmienne, zmienne które przechowują pewne wartości.

Fakt, że tymi wartościami są obiekty, które w dodatku interpretujemy w sposób prawie realny, nie ma tutaj większego znaczenia. Pomyślmy zatem, jaki efekt spowodowałby ten kod, gdybyśmy zamiast klasy CDioda użyli jakiegoś zwykłego, skalaranego typu? Dawna wartość zmiennej, do której nastąpiło przypisanie, zostałaby zapomniana i obie zmienne zawierałyby tę samą liczbę.

Dla obiektów rzecz ma się identycznie: po wykonaniu przypisania zarówno Lampa1, jak i Lampa2 reprezentować będą obiekty zielonych lamp. Czerwona lampa, pierwotnie zawarta w zmiennej Lampa1, zostanie zniszczona, a w jej miejsce pojawi się kopia zawartości zmiennej Lampa2. Nie bez powodu zaakcentowałem wyżej słowo "kopia". Obydwa obiekty są bowiem od siebie całkowicie niezależne. Jeżeli włączylibyśmy jeden z nich:

### Kod programu 5.15

```
1 dioda1.zapal();
```

drugi nie zmieniłby się wcale i nie obdarzył nas swym własnym światłem. Możemy więc podsumować nasz wywód krótką uwagą na temat zmiennych obiektowych:



Zmienne obiektowe przechowują obiekty w ten sam sposób, w jaki czynią to zwykłe zmienne ze swoimi wartościami. Identycznie odbywa się też przypisywanie 2 takich zmiennych - tworzone są wtedy odpowiednie kopie obiektów.

Wspominałem, że wszystko to może wydawać się naturalne, oczywiste i niepodważalne. Konieczne było jednak dokładne wyjaśnienie w tym miejscu tych z pozoru prostych zjawisk, gdyż drugi sposób postępowania z obiektami (który poznamy za moment) wprowadza w tej materii istotne zmiany.

Kontrolowanie obiektu jako całości ma rozliczne zastosowania, ale jednak znacznie częściej będziemy używać tylko jego pojedynczych składników, czyli pól lub metod. Doskonale wiemy już, jak się to robi: z pomocą przychodzi nam zawsze operator wyłuskania - kropka ( . ). Stawiamy więc go po nazwie obiektu, by potem wpisać nazwę wybranego elementu, do którego chcemy się odwołać. Pamiętajmy, że posiadamy wtedy dostęp jedynie do składowych publicznych klasy, do której należy obiekt.

Dalsze postępowanie zależy już od tego, czy naszą uwagę zwróciliśmy na pole, czy na metodę. W tym pierwszym, rzadszym przypadku nie odczuwamy żadnej różnicy w stosunku do pól w strukturach - i nic dziwnego, gdyż nie ma tu rzeczywiście najmniejszej rozbieżności :) Wywołanie metody jest natomiast łudząco zbliżone do uruchomienia zwyczajnej funkcji - tyle że w grę wchodzi tutaj nie tylko jej parametry, ale także obiekt, na rzecz którego daną metodę wywołujemy.

Każdy stworzony obiekt musi prędzej czy później zostać zniszczony, aby móc odzyskać zajmowaną przez niego pamięć i spokojnie zakończyć program. Dotyczy to także zmiennych obiektowych, lecz dzieje się to trochę jakby za plecami programisty. Zauważmy bowiem, iż w żadnym z naszych dotychczasowych programów, wykorzystujących techniki obiektowe, nie pojawiły się instrukcje, które jawnie odpowiadałyby za niszczenie stworzonych obiektów. Nie oznacza to bynajmniej, że zalegają one w pamięci operacyjnej, zajmując ją niepotrzebnie. Po prostu kompilator sam dba o to, by ich destrukcja nastąpiła w stosownej chwili.

A zatem kiedy jest ona faktycznie dokonywana? Nietrudno jest obmyślić odpowiedź na to pytanie, jeżeli przypomnimy sobie pojęcie zasięgu zmiennej. Powiedzieliśmy sobie ongiś, iż jest to taki obszar kodu programu, w którym dana zmienna jest dostępna. Dostępna - to znaczy zadeklarowana, z przydzieloną dla siebie pamięcią, a w przypadku zmiennej obiektowej - posiadająca również obiekt stworzony poprzez konstruktor klasy. Moment opuszczenia zasięgu zmiennej przez punkt wykonania programu jest więc kresem jej istnienia. Jeśli nieszczęsna zmienna była obiektową, do akcji wkracza destruktorki klasy (jeżeli został określony), sprząając ewentualny bałagan po obiekcie i niszcząc go. Dalej następuje już tylko zwolnienie pamięci zajmowanej przez zmienną i jej kariera kończy się w niebycie :)

Prezentowane tu własności zmiennych obiektowych być może wyglądają na nieznane i niespotykane wcześniej. Naprawdę jednak nie są niczym szczególnym, gdyż spotykaliśmy się z nimi od samego początku nauki programowania - w większości (z wyłączeniem wyłuskiwania składników) dotyczą one bowiem wszystkich zmiennych! Teraz wszakże omówiliśmy je sobie nieco dokładniej, koncentrując się przede wszystkim na "życiu" obiektów - chwilach ich tworzenia i niszczenia oraz operacjach na nich. Mając ugruntowaną tę wiedzę, będzie nam łatwiej zmierzyć się z drugim sposobem stosowania obiektów, który jest przedstawiony w następnym paragrafie.

### 5.7.1 Wskaźniki do obiektów

Wskaźniki już znacie (obiekty zapewne też) – to może teraz kilka przykładów zastosowania wskaźników do pracy z obiektami? Zaczniemy więc ... Hem, od czegoż to mielibyśmy zacząć, jeżeli nie od jakiejś zmiennej? W końcu bez zmiennych nie ma obiektów, a bez obiektów nie ma programowania (obiekowego :D). Na początek trywialny przykład:

#### Kod programu 5.16

```
1 CDioda *pDioda1 = new CDioda();
2 CDioda *pDioda2 = pDioda1;
3
4 pDioda2->pokaz();
5 pDioda1->zapal();
6 pDioda2->pokaz();
7
8 /// niszczenie diody
9 delete pDioda1;
10 pDioda2->pokaz() /// Błąd !! - nie ma już diody ...
11 delete pDioda2; /// Błąd !! - nie ma już czego niszczyć
```

To chyba oczywiste – mamy teraz tylko jeden obiekt, i dwie metody dostępu do niego. Wyjaśnienie należy się jednak odności operatora wyłuskania – teraz ma on nieco inną postać, nie jest nim kropka, ale strzałka ( -> ). Otrzymujemy ją, wpisując kolejno dwa znaki: myślnika oraz symbolu większości.

Wszelkie obiekty kiedyś należy zniszczyć; czynność ta, oprócz wyrabiania dobrego nawyku sprząwania po sobie, zwalnia pamięć operacyjną, które te obiekty zajmowały. Po zniszczeniu wszystkich możliwe jest bezpieczne zakończenie programu. Podobnie jak tworzenie, tak i niszczenie obiektów dostępnych poprzez wskaźniki nie jest wykonywane automatycznie. Wymagana jest do tego odrębna instrukcja delete – widzicie ją w kodzie powyżej.

Delete wywołuje destruktora obiektu, a następnie zwalnia pamięć zajęta przez obiekt, który kończy wtedy definitywnie swoje istnienie. To tyle jeśli chodzi o życiorys obiektu. Co się jednak dzieje z samym wskaźnikiem? Otóż nadal wskazuje on na miejsce w pamięci, w którym jeszcze niedawno egzystował nasz obiekt. Teraz jednak już go tam nie ma; wszelkie próby odwołania się do tego obszaru skończą się więc błędem, zwanym naruszeniem zasad dostępu (ang. access violation).

## Rozdział 6

# Programowanie zorientowane obiektowo

Autorzy:

*Paweł Wnuk*

### 6.1 Dziedziczenie

Jednym z głównych powodów, dla którego techniki obiektowe zyskały taką popularność, jest znaczący postęp w kwestii ponownego wykorzystywania raz napisanego kodu oraz rozszerzania i dostosowania go do własnych potrzeb. Cecha ta leży u samych podstaw programowania zorientowanego obiektowo: program konstruowany jako zbiór współdziałających obiektów nie jest już bowiem monolitem, ścisłym połączeniem danych i wykonywanych nań operacji. "Rozdrobniona" struktura zapewnia mu zatem modularność, nie jest trudno dodać do gotowej aplikacji nową funkcję czy też wyodrębnić z niej jeden podsystem i użyć go w kolejnej produkcji. Ułatwia to i przyspiesza realizację kolejnych projektów.

Wszystko zależy jednak od umiejętności i doświadczenia programisty. Nawet stosując techniki obiektowe można stworzyć program, którego elementy będą ze sobą tak ściśle zespolone, że próba ich użycia w następnej aplikacji będzie przypominała wciskanie słonia do szklanej butelki. Istnieje jeszcze jedna przyczyna, dla której kod oparty na programowaniu obiektowym łatwiej poddaje się "recyklingowi", mającemu przygotować go do ponownego użycia. Jest nim właśnie tytułowy mechanizm dziedziczenia.

Korzyści płynące z jego stosowania nie ograniczają się jednakże tylko do wtórnego "prze-robu" już istniejącego kodu. Przeciwnie, jest to fundamentalny aspekt programowania zorientowanego obiektowo niezmiernie ułatwiający i uprzyjemniający projektowanie każdej w zasadzie aplikacji. W połączeniu z technologią funkcji wirtualnych oraz polimorfizmu daje on niezwykle szerokie możliwości, o których szczegółowo traktuje praktycznie cały niniejszy rozdział.

Czymże jest więc owo dziedziczenie?

Człowiek jest taką dziwną istotą, która bardzo lubi posiadać uporządkowany i usystematyzowany obraz świata. Wprowadzanie porządku i pewnej hierarchii co do postrzeganych zjawisk i przedmiotów jest dla nas niemal naturalną potrzebą. Chyba najlepiej przejawia

się to w klasyfikacji biologicznej. Widząc na przykład psa wiemy przecież, że nie tylko należy on do gatunku zwanego psem domowym, lecz także do gromady znanej jako ssaki (wraz z końmi, słoniami, lwami, małpami, ludźmi i całą resztą tej menażerii). Te z kolei, razem z gadami, ptakami czy rybami należą do kolejnej, znacznie większej grupy organizmów zwanych po prostu zwierzętami. Nasz pies jest zatem jednocześnie psem domowym, ssakiem i zwierzęciem:

Gdyby był obiektem w programie, wtedy musiałby należeć aż do trzech klas naraz ... Byłoby to oczywiście niemożliwe, jeżeli wszystkie miałyby być wobec siebie równorzędne. Tutaj jednak tak nie jest: występuje między nimi hierarchia, jedna klasa pochodzi od drugiej. Zjawisko to nazywamy właśnie dziedziczeniem .

### Definicja 6.1

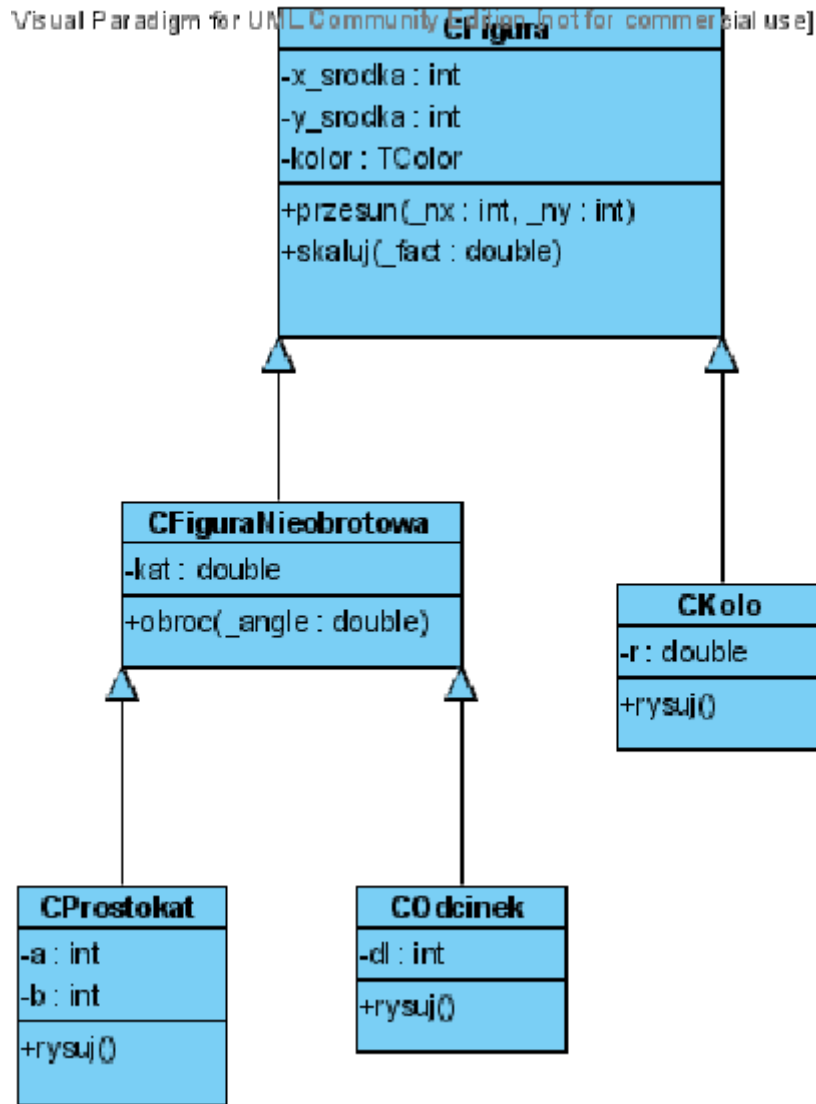
#### Dziedziczenie

Dziedziczenie (ang. inheritance ) to tworzenie nowej klasy na podstawie jednej lub kilku istniejących wcześniej klas bazowych.

Wszystkie klasy, które powstają w ten sposób (nazywamy je pochodnymi ), posiadają pewne elementy wspólne. Części te są dziedziczone z klas bazowych, gdyż tam właśnie zostały zdefiniowane. Ich zbiór może jednak zostać poszerzony o pola i metody specyficzne dla klas pochodnych. Będą one wtedy współistnieć z "dorobkiem" pochodzącym od klas bazowych, ale mogą oferować dodatkową funkcjonalność.

Tak w teorii wygląda system dziedziczenia w programowaniu obiektowym. Najlepiej będzie, jeżeli teraz przyjrzymy się, jak w praktyce może wyglądać jego zastosowanie. Dziedziczenie to główny mechanizm wyróżniający programowanie zorientowane obiektowo od programowania obiektowego. Opieramy się na cechach wspólnych grupy klas – zarówno jeśli chodzi o pola jak i metody. Wspólne cechy lądują w klasie bazowej, cechy szczególne – w klasach pochodnych.





Rysunek 6.1 – Przykład hierarchii dziedziczenia

O podstawowej konsekwencji takiego rozwiązania zdążyłem już wcześniej wspomnieć. Jest nią mianowicie przekazywanie pól oraz metod pochodzących z klasy bazowej do wszystkich klas pochodnych, które się z niej wywodzą. Zatem:



Klasa pochodna zawiera pola i metody odziedziczone po klasach bazowych. Może także posiadać dodatkowe, unikalne dla siebie składowe - nie jest to jednak obowiązkiem.

Oczywiście klas nie definiuje się dla samej przyjemności ich definiowania, lecz dla tworzenia z nich obiektów. Jeżeli więc posiadalibyśmy przedstawioną wyżej hierarchię w jakimś prawdziwym programie, to z pewnością pojawiłyby się w nim także instancje zaprezentowanych klas, czyli odpowiednie obiekty. W ten sposób wracamy do problemu postawionego na samym początku: jak obiekt może należeć do kilku klas naraz? Różnica polega wszak na tym, że mamy już jego gotowe rozwiązanie :) Otóż nasz obiekt psa należałby przede

wszystkim do klasy Pies domowy ; to właśnie tej nazwy użylibyśmy, by zadeklarować reprezentującą go zmienną czy też pokazujący nań wskaźnik. Jednocześnie jednak byłby on typu Ssak oraz typu Zwierzę , i mógłby występować w tych miejscach programu, w których byłby wymagany jeden z owych typów. Fakt ten jest przyczyną istnienia w programowaniu obiektowym zjawiska zwanego polimorfizmem. Ale o tym za chwil parę.

Pozyskawszy ogólne informacje o dziedziczeniu jako takim, możemy zobaczyć, jak idea ta została przełożona na nasz nieoceniony język C++ :) Dowiemy się więc, w jaki sposób definiujemy nowe klasy w oparciu o już istniejące oraz jakie dodatkowe efekty są z tym związane. Mechanizm dziedziczenia jest w C++ bardzo rozbudowany, o wiele bardziej niż w większości pozostałych języków zorientowanych obiektowo. Udostępnia on kilka szczególnych możliwości, które być może nie są zawsze niezbędne, ale pozwalają na dużą swobodę w definiowaniu hierarchii klas. Poznanie ich wszystkich nie jest konieczne, aby sprawnie korzystać z dobrodziejstw programowania obiektowego, jednak wiemy doskonale, że wiedza jeszcze nikomu nie zaszkodziła :D

Zacniemy oczywiście od najbardziej elementarnych zasad dziedziczenia klas oraz przyjrzymy się przykładom ilustrującym ich wykorzystanie.

### 6.1.1 Zasady dostępu do pól i metod w hierarchii

Jak pamiętamy, definicja klasy składa się przede wszystkim z listy deklaracji jej pól oraz metod, podzielonych na kilka części wedle specyfikatorów praw dostępu. W przypadku dziedziczenia pojawia się nowe określenie – `protected`. Same zasady dostępu wyglądają następująco:

- Pola prywatne pozostają prywatne
- Pola publiczne nadal są publiczne
- Pola chronione – dostępne dla danej klasy i wszystkich jej dzieci – dla reszty nie.
- Obowiązują reguły przykrywania nazw
- Nie obowiązują zasady przeciążania nazw funkcji w dziedziczeniu!

Możecie to zobaczyć na poniższym przykładzie:

#### Kod programu 6.1

```
1 class CMojObiekt {
2 public:
3     int zmienna_publiczna;
4     void metoda_publiczna();
5 protected:
6     int zmienna_chroniona;
7     void metoda_chroniona();
8 private:
9     int zmienna_prywatna;
10    void metoda_prywatna();
11 };
12
13 class CMojNastepca : public CMojObiekt {
```

```

14 public:
15 ...
16 void metoda_publiczna_nastepcy();
17 protected:
18 ...
19 private:
20 void metoda_prywatna_nastepcy;
21 };
22 void CMojNastepca::metoda_prywatna_nastepcy {
23 // tak mozna
24 zmienna_chroniona = 1;
25 metoda_chroniona();
26 // tak natomiast nie da rady
27 zmienna_prywatna = 1;
28 metoda_prywatna();
29 };
30 void CMojObiekt::metoda_publiczna_nastepcy {
31 // tak mozna
32 zmienna_chroniona = 1;
33 metoda_chroniona();
34 // tak natomiast nie da rady
35 zmienna_prywatna = 1;
36 metoda_prywatna();
37 };
38
39 CMojNastepca mn;
40 ...
41 // ponizszy kod jest nieprawid³owy
42 mn.zmienna_chroniona = 1;
43 mn.metoda_chroniona();

```

Należy używać specyfikatora `protected`, kiedy chcemy uchronić składowe przed dostępem z zewnątrz, ale jednocześnie mieć je do dyspozycji w klasach pochodnych.

Dopiero posiadając zdefiniowaną klasę bazową możemy przystąpić do określania dziedziczącej z niej klasy pochodnej. Jest to konieczne, bo w przeciwnym wypadku kazalibyśmy kompilatorowi korzystać z czegoś, o czym nie miałyby wystarczających informacji.

Samo dziedziczenie może być wykonane wg trzech różnych schematów. Tak więc mamy dziedziczenie:

- **Publiczne** - Wszystkie odziedziczone pola i metody mają taki zasięg jak zdefiniowano w klasie bazowej. Klasa pochodna i jej pochodne mają dostęp do części chronionej i publicznej, korzystający z obiektów pochodnych mają dostęp do części publicznej
- **Chronione** - Odziedziczone pola i metody publiczne i chronione stają się chronione w klasie pochodnej i jej pochodnych. Klasa pochodna i jej pochodne mają dostęp do części chronionej i publicznej, korzystający z obiektów pochodnych nie mają dostępu do żadnych pól i metod klasy bazowej
- **Prywatne** - Odziedziczone pola i metody publiczne i chronione stają się prywatne w klasie pochodnej. Klasa pochodna ma do nich dostęp, klasy wyprowadzone z pochodnej, oraz korzystający z obiektów pochodnych nie mają dostępu do żadnych pól i metod klasy bazowej. Przerwanie przechodzenia uprawnień!

Dlaczego tak jest? Otóż w 99.9% przypadków nie ma najmniejszej potrzeby zmiany praw dostępu do składowych odziedziczonych po klasie bazowej. Jeżeli więc któreś z nich zostały tam zadeklarowane jako `protected`, a inne jako `public`, to prawie zawsze życzymy sobie, aby w klasie pochodnej zachowały te same prawa. Zastosowanie dziedziczenia `public` czyni zadość tym żądaniom, dlatego właśnie jest ono tak często stosowane. Pozostałe dwa kwalifikatory są rzadko stosowane.

### 6.1.2 Dziedziczenie wielokrotne

ro możliwe jest dziedziczenie z wykorzystaniem jednej klasy bazowej, to raczej naturalne jest rozszerzenie tego zjawiska także na przypadki, w której z kilku klas bazowych tworzymy jedną klasę pochodną. Mówimy wtedy o dziedziczeniu wielokrotnym (ang. `multiple inheritance`). C++ jest jednym z niewielu języków, które udostępniają taką możliwość. Nie świadczy to jednak o jego niebotycznej wyższości nad nimi. Tak naprawdę technika dziedziczenia wielokrotnego nie daje żadnych nadzwyczajnych korzyści, a jej użycie jest przy tym dość skomplikowane. Decydując się na jej wykorzystanie należy więc posiadać całkiem spore doświadczenie w programowaniu.

### 6.1.3 Pułapki dziedziczenia

Chociaż idea dziedziczenia jest teoretycznie całkiem prosta do zrozumienia, jej praktyczne zastosowanie może niekiedy nastroczać pewnych problemów. Są one zazwyczaj specyficzne dla konkretnego języka programowania, jako że występują w tym względzie pewne różnice między nimi. W tym paragrafie zajmiemy się takimi właśnie drobnymi niuansami, które są związane z dziedziczeniem klas w języku C++.

### Co nie jest dziedziczone?

Wydawałoby się, że klasa pochodna powinna przejmować wszystkie składowe pochodzące z klasy bazowej - oczywiście z wyjątkiem tych oznaczonych jako `private`. Tak jednak nie jest, gdyż w trzech przypadkach nie miałyby to sensu. Owe trzy "nieprzechodnie" składniki klas to:

- konstruktory . Zadaniem konstruktora jest zazwyczaj inicjalizacja pól klasy na ich początkowe wartości, stworzenie wewnętrznych obiektów czy też alokacja dodatkowej pamięci. Czynności te prawie zawsze wymagają zatem dostępu do prywatnych pól klasy. Jeżeli więc konstruktor z klasy bazowej zostałby "wrzucony" do klasy pochodnej, to utraciłby z nimi niezbędne połączenie - wszak "zostałyby" one w klasie bazowej! Z tego też powodu konstruktory nie są dziedziczone.
- destruktory . Sprawa wygląda tu podobnie jak punkt wyżej. Działanie destruktorów najczęściej także opiera się na polach prywatnych, a skoro one nie są dziedziczone, zatem destruktory też nie powinny przechodzić do klas pochodnych. Dość ciekawym uzasadnieniem niedziedziczenia konstruktorów i destruktorów są także same ich nazwy, odpowiadające klasie, w której zostały zadeklarowane. Gdyby zatem przekazać je klasom pochodnym, wtedy zasada ich nazewnictwa zostałaby złamana.

- przeciążony operator przypisania (`=`). Zagadnienie przeciążania operatorów omówimy dokładnie w jednym z przyszłych rozdziałów. Na razie zapamiętaj, że składowa ta odpowiada za sposób, w jaki obiekt jest kopiowany z jednej zmiennej do drugiej. Taki transfer zazwyczaj również wymaga dostępu do pól prywatnych klasy, co od razu wyklucza dziedziczenie.

Ze względu na specjalne znaczenie konstruktorów i destruktorów, ich funkcjonowanie w warunkach dziedziczenia jest dość specyficzne. Sposób, w jaki C++ realizuje pomysł dziedziczenia, jest sam w sobie dosyć interesujący. Większość koderów uczących się tego języka z początku całkiem logicznie przypuszcza, że kompilator zwyczajnie pobiera deklaracje z klasy bazowej i wstawia je do pochodnej, ewentualne powtórzenia rozwiązując na korzyść tej drugiej.

W rzeczywistości wewnętrznie używana przez kompilator definicja klasy pochodnej jest identyczna z tą, którą wpisujemy do kodu; nie zawiera żadnych pól i metod pochodzących z klas bazowych! Jakim więc cudem są one dostępne? Odpowiedź jest oczywista: podczas tworzenia obiektu klasy pochodnej dokonywana jest także kreacja obiektu klasy bazowej, który staje się jego częścią. Zatem nasz obiekt pochodny to tak naprawdę obiekt bazowy plus dodatkowe pola, zdefiniowane w jego własnej klasie.

W C++ obowiązuje zasada, iż najpierw wywoływany jest konstruktor "najbardziej bazowej" klasy danego obiektu, a potem te stojące kolejno niżej w hierarchii. Ponieważ klasa może posiadać więcej niż jeden konstruktor, kompilator musiałby podjąć decyzję, który z nich powinien zostać użyty. Można zdefiniować który z konstruktorów będzie wywołany na podstawie kodu konstruktora klasy pochodnej. Jeśli tam jawnie tego nie wyspecyfikujemy – wykorzystany zostanie domyślny konstruktor bezparametrowy. Podobny problem nie istnieje dla destruktorów, gdyż one nigdy nie posiadają parametrów. Podczas niszczenia obiektu są one wywoływane w kolejności od tego z klasy pochodnej do tych z klas bazowych.

## 6.2 Polimorfizm

Idea dziedziczenia w zaprezentowanej dotąd postaci jest nastawiona przede wszystkim na uzupełnianie definicji klas bazowych o kolejne składowe w klasach pochodnych. Tylko czasami zastępowaliśmy już istniejące metody ich nowymi wersjami, właściwymi dla tworzonych klas. Takie sytuacje są jednak w praktyce dosyć częste – albo raczej korzystne jest prowokowanie takich sytuacji, gdyż niejednokrotnie dają one świetne rezultaty i niespotykane wcześniej możliwości przy niewielkim nakładzie pracy. Oczywiście dzieje się tak tylko wtedy, gdy mamy odpowiednie podejście do sprawy.

A to odpowiednie podejście nazywa się polimorfizmem, czyli:

### Definicja 6.2

Polimorfizm

Na czym polega to w praktyce? Przejdźmy do przykładu z życia – i człowiek i małpa wydają odgłosy (to że różne – to też istotne, ale nie na tym poziomie abstrakcji). Toś kto korzysta z obiektu klasy człowiek czy też małpa – nie musi wiedzieć którego typu jest dany osobnik by zmusić go do wydania głosu – wystarczy że wie że osobnik wydaje głosy. Wywołaniem odpowiedniej wersji funkcji zajmie się kompilator.

## Kod programu 6.2

```
1 class CMalpa {
2 public:
3     virtual void dajGlos() {
4         cout << "Hyyyhmyyy\n";
5     }
6 };
7
8 class CCzlowiek : public CMalpa {
9 public:
10     virtual void dajGlos() {
11         cout << "Auuu, boli\n";
12     }
13 };
14
15 int main(int argc, char *argv[])
16 {
17     CMalpa m;
18     CCzlowiek c;
19     CMalpa *cm;
20
21     m.dajGlos();
22     c.dajGlos();
23     cm = &m;
24     cm->dajGlos();
25     cm = &c;
26     cm->dajGlos();
27 }
```

Istotą zróżnicowania są funkcje wirtualne. Są one definiowane przy wykorzystaniu słowa kluczowego `virtual` – przy czym na etapie implementacji słowa `virtual` nie wykorzystujemy. Przy wywołaniu zostanie przeszukana tablica funkcji wirtualnych przynależąca do danego obiektu – i wyszukana wersja najbliższa w hierarchii dziedziczenia. Nie może zmieniać się liczba parametrów (sygnatura) funkcji wirtualnej.

Funkcja zaczyna zachowywać się jak wirtualna w momencie pierwszego pojawienia się słowa `virtual`. Zachowanie wirtualne może (ale nie musi) skończyć się po pierwszym wystąpieniu funkcji bez `virtual` w hierarchii dziedziczenia. Rodzaj dziedziczenia nie wpływa na zachowanie się funkcji wirtualnych (zmienia się jedynie zasięg ich widoczności). No i na koniec – metody statyczne nie mogą być wirtualne.

Przykład jak to działa w praktyce?

## Kod programu 6.3

```
1 public:
2     virtual void f1() { cout << "A f1\n"; }
3     void f2() { cout << "A f2\n"; }
4 };
5
6 class B : public A {
7 public:
8     void f1() { cout << "B f1\n"; }
9     virtual void f2() { cout << "B f2\n"; }
10 };
```



```

11
12 class C : public B {
13 public:
14     void f1() { cout << "C f1\n"; }
15     virtual void f2() { cout << "C f2\n"; }
16 };
17
18 int main(int argc, char *argv[]) {
19     A a; B b; C c;
20     A* pab, *pac;
21     B* pbc;
22     pab = &b;
23     pac = &c;
24     pbc = &c;
25     cout << "Klasa A:\n";
26     a.f1();
27     a.f2();
28     cout << "Klasa B:\n";
29     b.f1();
30     b.f2();
31     cout << "Klasa B jako A:\n";
32     pab->f1();
33     pab->f2();
34     cout << "Klasa C:\n";
35     c.f1();
36     c.f2();
37     cout << "Klasa C jako A:\n";
38     pac->f1();
39     pac->f2();
40     cout << "Klasa C jako B:\n";
41     pbc->f1();
42     pbc->f2();
43 }

```

Funkcje wirtualne mogą być wywoływane w klasach bazowych – zostanie wtedy automatycznie wybrana odpowiednia wersja danej funkcji – odpowiadająca tożsamości obiektu wywołującego. W ten sposób możemy wykorzystywać przy tworzeniu kodu zachowanie jeszcze niezdefiniowane, i zależne od typu obiektu z którym pracujemy. Funkcje wirtualne mogą być również wykorzystane w kodzie korzystającym z obiektów danego typu.

#### Kod programu 6.4

```

1 public:
2     virtual void rysuj() { }
3     void przesun(int _x, int _y)
4     {
5         X = _x;
6         Y = _y;
7         rysuj();
8     };
9 private:
10    int X;
11    int Y;
12 };
13
14 class CKolo :public CFigura {
15 public:

```



```
16 virtual void rysuj() {  
17     // kod rysowania  
18 }  
19 };
```

Jawne wywołanie funkcji bazowej jest możliwe poprzez podanie poprzedzonej dwukropkiem nazwy jej klasy. Nie ma w C++ możliwości niejawnego wywołania odziedziczonej instancji. Ogólnie – zasięg widoczności poszczególnych pól i metod w klasach może być traktowany jako zagnieżdżony z punktu widzenia dziedziczenia:

### Kod programu 6.5

```
1 public:  
2     virtual void f() { cout << "A f1\n"; }  
3 };  
4  
5 class B : public A {  
6 public:  
7     virtual void f() {  
8         A::f();  
9         cout << "B f1\n";  
10    }  
11 };  
12  
13 class C : public B {  
14 public:  
15     virtual void f() {  
16         A::f();  
17         B::f();  
18         cout << "C f1\n";  
19    }  
20 };
```

## 6.2.1 Abstrakcje

Funkcja zadeklarowana w klasie jako wirtualna, która nie ma definicji – jest funkcją abstrakcyjną. W C++ oznaczamy ten fakt pisząc =0 po deklaracji funkcji. Klasa która ma choć jedną funkcję abstrakcyjną – jest klasą abstrakcyjną. Nie można tworzyć obiektów typu klas abstrakcyjnych. Klasa która ma wszystkie metody abstrakcyjne i nie posiada pól – jest interfejsem.

### Kod programu 6.6

```
1 public:  
2     virtual void rysuj() = 0;  
3 };  
4  
5 class Cfigura : public CFiguraBaza {  
6 public:  
7     virtual void rysuj() = 0;  
8     void przesun(int _x, int _y)  
9     {  
10        X = _x;
```

```
11   Y = _y;
12   rysuj();
13 };
14 private:
15   int X;
16   int Y;
17 };
18
19 class CKolo :public CFigura {
20 public:
21   virtual void rysuj() {
22     // kod rysowania
23   }
24 };
```

## 6.2.2 Klonowanie obiektów

W przypadku obiektów będących w hierarchii dziedziczenia, kopiowanie ich nie da się prosto wykonać korzystając jedynie z mechanizmów polimorfizmu – przecież konstruktor nie jest wirtualny i nie jest dziedziczony. Dlatego też rozwiązanie poniżej raczej nie zadziała:

### Kod programu 6.7

```
1 class A {
2 public:
3   A() {};
4   A(const A& _s) { };
5   virtual void f() {
6     cout<<"A nadaje\n";
7   }
8 };
9
10 class B : public A {
11 public:
12   B() {};
13   B(const B& _s) : A(_s) { };
14   virtual void f() {
15     cout<<"B nadaje\n";
16   }
17 };
18
19 int main(int argc, char *argv[])
20 {
21   B *b = new B;
22   A *a1 = b;
23   A *a2 = new A(*a1);
24   //A *a3 = new B(*a1);
25
26   a1->f();
27   a2->f();
28 }
```

Możecie sami się przekonać, uruchamiając powyższy przykład. Mimo że a2 zostało utworzone jako kopia a1 – które jest typu B, to i tak wywołany został konstruktor A, a nie

B. Jeśli znamy typ obiektu pochodnego przed kopiowaniem – to możemy jawnie wywołać konstruktor B. Ale przecież cała zabawa z polimorfizmem polega na tym, by móc korzystać z obiektu jedynie w oparciu o jego interfejs – a więc nie znając docelowego typu obiektu. Czy da się kopiować elementy w taki sposób? Da ... wystarczy odpowiednio wykorzystać mechanizm funkcji wirtualnych.

### Kod programu 6.8

```
1 class A {
2 public:
3   A() { };
4   A(const A& _s) { };
5   virtual void f() { cout<<"A " << FMyNum << " nadaje\n"; }
6   virtual A* clone() { return new A(*this); };
7 protected:
8   static int FCnt;
9   int FMyNum;
10 };
11 int A::FCnt = 0;
12
13 class B : public A {
14 public:
15   B() { };
16   B(const B& _s) : A(_s) { };
17   virtual void f() { cout<<"B " << FMyNum << " nadaje\n"; }
18   virtual A* clone() { return new B(*this); };
19 };
20
21 int main(int argc, char *argv[])
22 {
23   B *b = new B;
24   A *a1 = b;
25   A *a2 = new A(*a1);
26   A *a4 = a1->clone();
27
28   a1->f();
29   a2->f();
30   a4->f();
31 }
```

Kod powyżej jest jeszcze uzupełniony o dodatkową informację – wykorzystując pole statyczne, zliczamy wszystkie egzemplarze danej klasy.



## Część III Szablony



**KAPITAŁ LUDZKI**  
NARODOWA STRATEGIA SPÓJNOŚCI

**UNIA EUROPEJSKA**  
EUROPEJSKI  
FUNDUSZ SPOŁECZNY



## Wstęp

Trzecia część podręcznika – to delikatne wprowadzenie w zagadnienia związane z programowaniem generycznym, czyli szablonami. Same szablony są cechą charakterystyczną języka C++ - na tyle charakterystyczną, że długo nie występowały w innych językach programowania. Długo też nie były silnie wykorzystywane w C++. Sytuacja zmieniła się dopiero po rozpowszechnieniu biblioteki STL. To ona, w połączeniu z jej wersją rozwojową, czyli biblioteką boost, pokazuje rzeczywistą siłę szablonów.

# Rozdział 7

## Funkcje uogólnione

Autorzy:

*Paweł Wnuk*

### 7.1 Funkcje uogólnione

W praktyce programowania często spotykamy się z funkcjami (algorytmami), które można zastosować do szerokiej klasy typów i struktur danych. Typowym przykładem jest funkcja obliczająca maksimum dwu wartości. Ten trywialny, aczkolwiek przydatny kod można zapisać np. w postaci:

#### Kod programu 7.1

```
1 int max(int a,int b) {  
2     return (a>b)?a:b;  
3 };
```

Funkcja `max` wybiera większy z dwu `int`-ów, ale widać, że kod będzie identyczny dla argumentów dowolnego innego typu pod warunkiem, iż istnieje dla niego operator porównania i konstruktor kopiujący. W językach programowania z silną kontrolą typów, takich jak C, C++ czy Java definiując funkcję musimy jednak podać typy przekazywanych parametrów oraz typ wartości zwracanej. Oznacza to, że dla każdego typu argumentów musimy definiować nową funkcję `max`:

#### Kod programu 7.2

```
1 int max(int a, int b)    {return (a>b)?a:b;};  
2 double max(double a,double b) {return (a>b)?a:b;};  
3 string max(string a,string b) {return (a>b)?a:b;};  
4 // skorzystaliśmy tu z dostępnej w C++ możliwości przeładowywania funkcji  
5  
6 main() {  
7     cout<< max(7,5)<<end;  
8     cout<< max(3.1415,2.71)<<endl;  
9     cout<< max("Ania","Basia")<<endl;  
10 }
```

Takie powtarzanie kodu, poza oczywistym zwiększeniem nakładu pracy, ma inne niepożądane efekty, związane z trudnością zapewnienia synchronizacji kodu każdej z funkcji. Jeśli np. zauważymy błąd w kodzie, to musimy go poprawić w kilku miejscach. To samo dotyczy optymalizacji kodu. W powyższym przykładzie kod jest wyjątkowo prosty, ale nie jest trudno znaleźć przykłady, gdzie już taki prosty nie będzie.

### 7.1.1 Funkcje uogólnione bez szablonów

Jak radzili, a właściwie jak radzą sobie programiści bez możliwości skorzystania z szablonów? Tradycyjne sposoby rozwiązywania tego typu problemów to między innymi makra lub używanie wskaźników typów ogólnych, takich jak `void *`, jak np. w funkcji `qsort` ze standardowej biblioteki C:

#### Kod programu 7.3

```
1 #define max(a,b) ( (a>b)?a:b )
2 void qsort(void *base, size_t nmem, size_t size,
3           int(*compar)(const void *, const void *));
```

Mimo iż użyteczne, żadne z tych rozwiązań nie może zostać uznane za wystarczająco ogólne i bezpieczne.

Można się również pokusić o próbę rozwiązania tego problemu za pomocą mechanizmów programowania obiektowego. W sumie jest to bardziej wyrafinowana odmiana rzutowania na `void *`. Polega na zdefiniowaniu ogólnego typu dla obiektów, które mogą być porównywane, i następnie korzystanie z obiektów pochodnych do nich.

#### Kod programu 7.4

```
1 #include <iostream>
2 using namespace std;
3
4 class GreaterThenComparable {
5 public:
6     virtual bool operator>( const GreaterThenComparable &b) const = 0;
7 } ;
8
9 const GreaterThenComparable &max(const GreaterThenComparable &a,
10                                const GreaterThenComparable &b) {
11     return (a>b)? a:b;
12 }
13
14 class Int:public GreaterThenComparable {
15     int val;
16 public:
17     Int(int i = 0):val(i){};
18     operator int() {return val;};
19     virtual bool operator>(const GreaterThenComparable &b) const {
20         return val>static_cast<const Int&>(b).val;
21     };
22 }
```



```
23 main() {  
24  
25 Int a(3),b(2);  
26 Int c;  
27 c = (const Int &)::max(a,b);  
28 cout<<(int)c<<endl;  
29 }
```

Widać więc wyraźnie, że to podejście wymaga sporego nakładu pracy, a więc w szczególności w przypadku tak prostej funkcji jak `max` jest wysoce niepraktyczne. Ogólnie rzecz biorąc ma ono następujące wady:

1. Wymaga dziedziczenia z abstrakcyjnej klasy bazowej `GreaterThanComparable`, czyli może być zastosowane tylko do typów zdefiniowanych przez nas. Inne typy, w tym typy wbudowane, wymagają kopertowania w klasie opakowującej, takiej jak klasa `Int` w powyższym przykładzie.
2. Ponieważ potrzebujemy polimorfizmu funkcja `operator>()` musi być funkcją wirtualną, a więc musi być funkcją składową klasy i nie może być typu `inline`. W przypadku tak prostych funkcji niemożność rozwinięcia ich w miejscu wywołania może prowadzić do dużych narzutów w czasie wykonania.
3. Funkcja `max` zwraca zawsze referencje do `GreaterThanComparable`, więc konieczne jest rzutowanie na typ wynikowy (tu `Int`).

## 7.2 Szablony funkcji

Widać, że podejście obiektowe nie nadaje się najlepiej do rozwiązywania tego szczególnego problemu powielania kodu. Dlatego w C++ wprowadzono nowy mechanizm: szablony. Szablony zezwalają na definiowanie całych rodzin funkcji, które następnie mogą być używane dla różnych typów argumentów. Definicja szablonu funkcji `max` wygląda następująco:

### Kod programu 7.5

1

Przyjrzyjmy się jej z bliska. Wyrażenie `template<typename T>` oznacza, że mamy do czynienia z szablonem, który posiada jeden parametr formalny nazwany `T`. Słowo kluczowe `typename` oznacza, że parametr ten jest typem (nazwą typu). Zamiast słowa `typename` możemy użyć słowa kluczowego `class`. Nazwa tego parametru może być następnie wykorzystywana w definicji funkcji w miejscach, gdzie spodziewamy się nazwy typu. I tak powyższe wyrażenie definiuje funkcję `max`, która przyjmuje dwa argumenty typu `T` i zwraca wartość typu `T`, będącą wartością większego z dwu argumentów. Typ `T` jest na razie niewyspecyfikowany. W tym sensie szablon definiuje całą rodzinę funkcji. Konkretną funkcję z tej rodziny wybieramy poprzez podstawienie za formalny parametr `T` konkretnego typu będącego argumentem szablonu. Takie podstawienie nazywamy konkretyzacją szablonu. Argument szablonu umieszczamy w nawiasach ostrych za nazwą szablonu (w praktyce można uniknąć konieczności jawnej specyfikacji argumentów szablonu, opiszę to później:

### Kod programu 7.6

```
1 k=max<int>(i,j);
```

Takie użycie szablonu spowoduje wygenerowanie identycznej funkcji jak pisana wcześniej. W powyższym przypadku za T podstawiamy int. Oczywiście możemy podstawić za T dowolny typ i używając szablonów program można zapisać następująco:

### Kod programu 7.7

```
1 main() {  
2   cout<<::max<int>(7,5)<<endl;  
3   cout<<::max<double>(3.1415,2.71)<<endl;  
4   cout<<::max<string>("Ania","Basia")<<endl;  
5 }
```

W powyższym kodzie użyliśmy konstrukcji ::max(a,b). Dwa dwukropki oznaczają, że używamy funkcji max zdefiniowanej w ogólnej przestrzeni nazw. Jest to konieczne aby kod się skompilował, ponieważ szablon max istnieje już w standardowej przestrzeni nazw std. W dalszej części wykładu będę te podwójne dwukropki pomijać.

Oczywiście istnieją typy których podstawienie spowoduje błędy kompilacji, np.

### Kod programu 7.8

```
1 max<complex<double>>(c1,c2); //brak operatora >  
2  
3 ( Źródło: max_template.cpp)  
4  
5 lub  
6  
7 class X {  
8   private:  
9     X(const X &){};  
10 };  
11 X a,b;  
12 max<X>(a,b); //prywatny (niewidoczny) konstruktor kopiujący
```

Ogólnie rzecz biorąc, każdy szablon definiuje pewną klasę typów, które mogą zostać podstawione jako jego argumenty.

Dedukcja argumentów szablonu

Użyteczność szablonów funkcji zwiększa istotnie fakt, że argumenty szablonu nie muszą być podawane jawnie. Kompilator może je wydedukować z argumentów funkcji. Tak więc zamiast kodu jawnie specyfikującego typ dla funkcji max, można napisać;

### Kod programu 7.9

```
1 k=max(i,j);
```

i kompilator zauważy, że tylko podstawienie int-a za T umożliwi dopasowanie sygnatury funkcji do parametrów jej wywołania i automatycznie dokona odpowiedniej konkretyzacji.

Może się zdarzyć, że podamy takie argumenty funkcji, że dopasowanie argumentów wzorca będzie niemożliwe, otrzymamy wtedy błąd kompilacji. Trzeba pamiętać, że mechanizm automatycznego dopasowywania argumentów szablonu powoduje wyłączenie automatycznej konwersji argumentów funkcji. Podanie jawnie argumentów szablonu (w nawiasach ostrych za nazwą szablonu) jednoznacznie określa sygnaturę funkcji, a więc umożliwia automatyczną konwersję typów. Ilustruje to poniższy kod:

#### Kod programu 7.10

```
1 main() {
2     cout<<::max(3.14,2)<<endl;
3     // błąd: kompilator nie jest w stanie wydedukować argumentu szablonu, bo typy
4     // argumentów (double,int) nie pasują do (T,T)
5
6     cout<<::max<int>(3.14,2)<<endl;
7     // podając argument jawnie wymuszamy sygnaturę int max(int,int), a co za tym
8     // idzie automatyczną konwersję argumentu 1 do int-a
9
10    cout<<::max<double>(3.14,2)<<endl;
11    // podając argument szablonu jawnie wymuszamy sygnaturę
12    // double max(double,double)
13    // a co za tym idzie automatyczną konwersję argumentu 2 do double-a
14
15    int i;
16    cout<<::max<int *>(&i,i)<<endl;
17    //błąd: nie istnieje konwersja z typu int na int*
```

Może warto zauważyć, że automatyczna dedukcja parametrów szablonu jest możliwa tylko wtedy, jeśli parametry wywołania funkcji w jakiś sposób zależą od parametrów szablonu. Jeśli tej zależności nie ma, z przyczyn oczywistych dedukcja nie jest możliwa i trzeba parametry podawać jawnie. Wtedy istotna jest kolejność parametrów na liście. Jeżeli parametry, których nie da się wydedukować, umieścimy jako pierwsze, wystarczy, że tylko je podamy jawnie, a kompilator wydedukuje resztę. Ilustruje to poniższy kod:

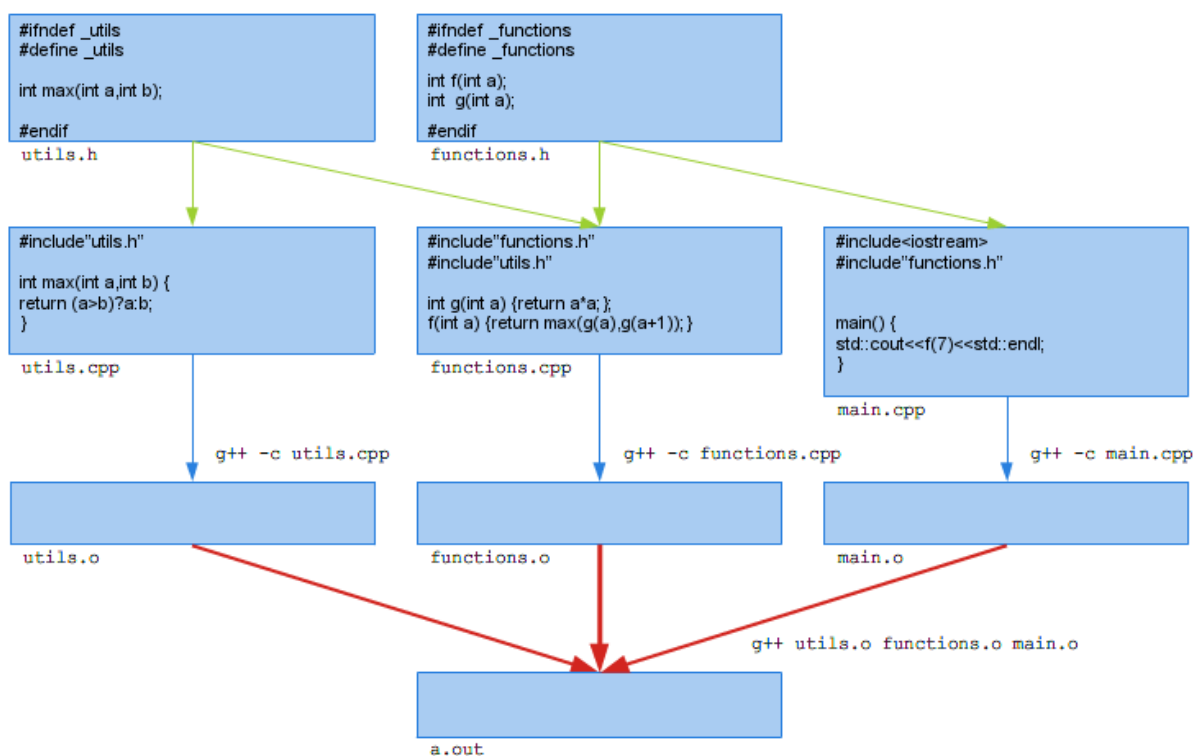
#### Kod programu 7.11

```
1 return (T)u;
2 };
3 template<typename U,typename T> T inv_convert(U u) {
4     return (T)u;
5 };
6 // funkcje różnią się tylko kolejnością parametrów szablonu
7
8 main() {
9     cout<<convert(33)<<endl;
10    // błąd: kompilator nie jest w stanie wydedukować pierwszego parametru
11    // szablonu, bo nie zależy on od parametru wywołania funkcji
12
13    cout<<convert<char>(33)<<endl;
14    // w porządku: podajemy jawnie argument T, kompilator sam dedukuje
15    // argument U z typu argumentu wywołania funkcji
16
17    cout<<inv_convert<char>('a')<<endl;
18    // błąd: podajemy jawnie argument odpowiadający parametrowi U.
19    // Kompilator nie jest w stanie wydedukować argumentu T, bo nie zależy on od argumentu
```

```
20 // wywołania funkcji
21
22 cout<<inv_convert<int,char>(33)<<endl;
23 // w porządku: podajemy jawnie oba argumenty szablonu
24 }
```

## 7.2.1 Używanie szablonów

Z użyciem szablonów wiąże się parę zagadnień niewidocznych w prostych przykładach. W językach C i C++ zwykle rozdzielamy deklarację funkcji od jej definicji i zwyczajowo umieszczamy deklarację w plikach nagłówkowych \*.h, a definicję w plikach źródłowych \*.c, \*.cpp itp. Pliki nagłówkowe są w czasie kompilacji włączane do plików, w których chcemy korzystać z danej funkcji, a pliki źródłowe są pojedynczo kompilowane do plików “obiektowych” \*.o. Następnie pliki obiektywne są łączone w jeden plik wynikowy (zob. rysunek 1.1). W pliku korzystającym z danej funkcji nie musimy więc znać jej definicji, a tylko deklarację. Na podstawie nazwy funkcji konsolidator powiąże wywołanie funkcji z jej implementacją znajdującą się w innym pliku obiekowym. W ten sposób tylko zmiana deklaracji funkcji wymaga rekompilacji plików, w których z niej korzystamy, a zmiana definicji wymaga jedynie rekompilacji pliku, w którym dana funkcja jest zdefiniowana.



Rysunek 7.1 – Typowa organizacja kodu

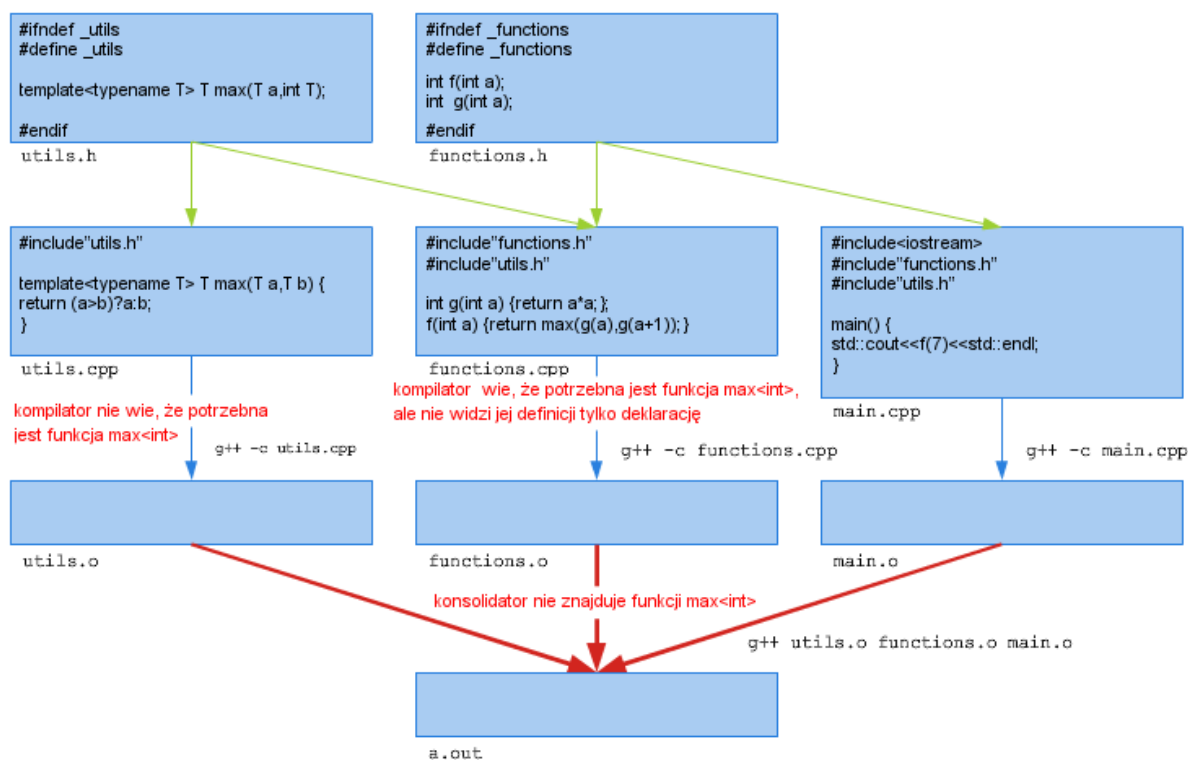
organizacja umożliwia przestrzeganie ”reguły jednej definicji” (one definition rule), wymaganej przez C++. To po to w nagłówkach pojawia się fragment uniemożliwiający

podwójne włączenie tego pliku do jednej jednostki translacyjnej:

### Kod programu 7.12

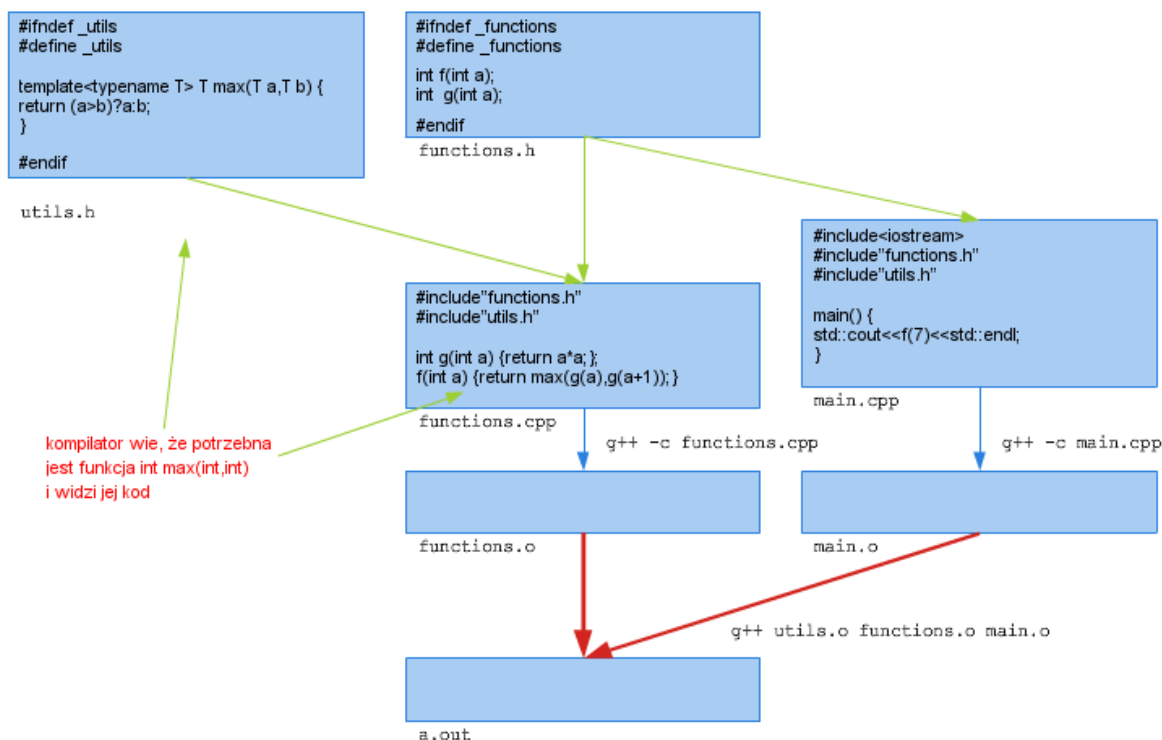
```
1 #ifndef _nazwa_pliku_
2 #define _nazwa_pliku_
3 ...
4 #endif
```

Podobne podejście do kompilacji szablonów się nie powiedzie. Powodem jest fakt, że w trakcie kompilacji pliku `utils.cpp` kompilator nie wie jeszcze, że potrzebna będzie funkcja `max<int>`, wobec czego nie generuje kodu żadnej funkcji, a jedynie sprawdza poprawność gramatyczną szablonu. Z kolei podczas kompilacji pliku `main.cpp` kompilator już wie, że ma skonkretyzować szablon dla `T = int`, ale nie ma dostępu do kodu szablonu.



**Rysunek 7.2** – Takie dołączanie nie zadziała w przypadku szablonów

Istnieją różne rozwiązania tego problemu. Najprościej chyba jest zauważyć, że opisane zachowanie jest analogiczne do zachowania podczas kompilacji funkcji rozwijanych w miejscu wywołania (`inline`), których definicja również musi być dostępna w czasie kompilacji. Podobnie więc jak w tym przypadku możemy zamieścić wszystkie deklaracje i definicje szablonów w pliku nagłówkowym, włączanym do plików, w których z tych szablonów korzystamy. Podobnie jak w przypadku funkcji `inline` reguła jednej definicji zezwala na powtarzanie definicji/deklaracji szablonów w różnych jednostkach translacyjnych, pod warunkiem, że są one identyczne. Stąd konieczność umieszczania ich w plikach nagłówkowych.



**Rysunek 7.3** – Przykład poprawnego dołączenia szablonu

Ten sposób organizacji pracy z szablonami, nazywany modelem włączenia, jest najbardziej uniwersalny. Jego główną wadą jest konieczność rekompilacji całego kodu korzystającego z szablonów przy każdej zmianie definicji szablonu. Również jeśli zmienimy coś w pliku, w którym korzystamy z szablonu, to musimy rekompilować cały kod szablonu włączony do tego pliku, nawet jeśli nie uległ on zmianie. Jeśli się uwzględni fakt, że kompilacja szablonu jest bardziej skomplikowana od kompilacji "zwykłego" kodu, to duży projekt intensywnie korzystający z szablonów może wymagać bardzo długich czasów kompilacji.

Możemy też w jakiś sposób dać znać kompilatorowi, że podczas kompilacji pliku `utils.cpp` powinien wygenerować kod dla funkcji `max<int>`. Można to zrobić dodając jawne żądanie konkretyzacji szablonu.

### Kod programu 7.13

```
1 template int max<int>(int ,int) ; // konkretyzacja jawna
```

Używając konkretyzacji jawnej musimy pamiętać o dokonaniu konkretyzacji każdej używanej funkcji, tak że to podejście nie skaluje się zbyt dobrze. Ponadto w przypadku szablonów klas (omawianych w następnym module) konkretyzacja jawna pociąga za sobą konkretyzację wszystkich metod danej klasy, a konkretyzacja "na żądanie" - jedynie tych używanych w programie.



## 7.2.2 Pozatypowe parametry szablonów

Poza parametrami określającymi typ, takimi jak parametr `T` w dotychczasowych przykładach, szablony funkcji mogą przyjmować również parametry innego rodzaju. Obecnie mogą to być inne szablony, co omówię w następnym podrozdziale lub parametry określające nie typ, ale wartości. Jak na razie (w obecnym standardzie) te wartości nie mogą być dowolne, ale muszą mieć jeden z poniższych typów:

- typ całkowitoliczbowy bądź typ wyliczeniowy
- typ wskaźnikowy
- typ referencyjny.

Takie parametry określające wartość nazywamy parametrami pozatypowymi. W praktyce z parametrów pozatypowych najczęściej używa się parametrów typu całkowitoliczbowego. Np.

### Kod programu 7.14

```
1   T total=0.0;
2   for(size_t i=0;i<N;++i)
3       total += a[i]*b[i] ;
4
5   return total;
6 };
7
8 int main() {
9     double x[3],y[3];
10    dot_product<3>(x,y);
11 }
```

Po raz drugi zwracam uwagę na kolejność parametrów szablonu na liście parametrów. Dzięki temu, że niededukowalny parametr `N` jest na pierwszym miejscu wystarczy podać jawnie tylko jego, drugi parametr typu `T` zostanie sam automatycznie wydedukowany na podstawie przekazanych argumentów wywołania funkcji. Parametry pozatypowe są zresztą "ciężko dedukowalne". Właściwie jedynym sposobem na przekazania wartości stałej poprzez typ argumentu wywołania jest skorzystanie z parametrów będących szablonami klas.

Używając pozatypowych parametrów szablonów musimy pamiętać, że odpowiadające im argumenty muszą być stałymi wyrażeniami czasu kompilacji. Stąd jeżeli używamy typów wskaźnikowych, muszą to być wskaźniki do obiektów łączonych zewnętrznymi, a nie lokalnymi. Ponieważ jednak jeszcze ani razu nie używałem pozatypowych parametrów szablonów innych niż typy całkowite, to nie będę podawał żadnych przykładów takich parametrów na tym wykładzie.

## 7.2.3 Szablony parametrów szablonu

Jak już wspomniałem w poprzednim podrozdziale, parametrami szablonu funkcji mogą być również szablony klas (zob. następny podrozdział). Szablony parametrów szablonu



umożliwiają przekazanie nazwy szablonu jako argumentu szablonu funkcji. Więcej o nich napiszę w drugiej części wykładu. Tutaj tylko pokażę jako ciekawostkę w jaki sposób można dedukować wartości pozatypowych argumentów szablonu.

### Kod programu 7.15

```
1 template< template<int N> class C,int K>
2 /* taka definicja oznacza, że parametr C określa szablon klasy
3 posiadający jeden parametr typu int. Parametr N służy tylko
4 do definicji szablonu C i nie może być użyty nigdzie indziej */
5 void f(C<K>){
6   cout<<K<<endl;
7 };
8
9 template<int N> struct SomeClass {};
10
11 main() {
12   SomeClass<1> c1;
13   SomeClass<2> c2;
14
15   f(c1); C=SomeClass K=1
16   f(c2); C=SomeClass K=2
17 }
```

# Rozdział 8

## Szablony klas

Autorzy:

*Paweł Wnuk*

### 8.1 Typy uogólnione

Uwagi na początku poprzedniego rozdziału odnoszą się w tej samej mierze do klas, jak i do funkcji. I tutaj mamy do czynienia z kodem, który w niezmienionej postaci musimy powielać dla różnych typów. Sztandarowym przykładem takiego kodu są różnego rodzaju kontenery (pojemniki), czyli obiekty służące do przechowywania innych obiektów. Jest oczywiste, że kod kontenera jest w dużej mierze niezależny od typu obiektów w nim przechowywanych. Jako przykład weźmy sobie stos liczb całkowitych. Możliwa definicja klasy stos może wyglądać następująco, choć nie polecam jej jako wzoru do naśladowania w prawdziwych aplikacjach:

#### Kod programu 8.1

```
1 private:
2   int rep[N];
3   size_t top;
4 public:
5   static const size_t N=100;
6   Stack():_top(0) {};
7   void push(int val) {_rep[_top++]=val;}
8   int pop() {return rep[--top];}
9   bool is_empty {return (top==0);}
10 }
```

Ewidentnie ten kod będzie identyczny dla stosu obiektów dowolnego innego typu, pod warunkiem, że typ ten posiada zdefiniowany operator `=()` i konstruktor kopiujący. W celu zaimplementowania kontenerów bez pomocy szablonów możemy próbować podobnych sztuczek jak te opisane w poprzednim rozdziale. W językach takich jak Java czy Smalltalk, które posiadają uniwersalną klasę `Object`, z której są dziedziczone wszystkie inne klasy, a nie posiadają (Java już posiada) szablonów, uniwersalne kontenery są implementowane właśnie poprzez rzutowanie na ten ogólny typ. W przypadku C++ nawet to rozwiązanie nie jest praktyczne, bo C++ nie posiada pojedynczej hierarchii klas.

## 8.2 Szablony klas

Rozwiązaniem są znów szablony, tym razem szablony klas. Podobnie jak w przypadku szablonów funkcji, szablon klasy definiuje nam w rzeczywistości całą rodzinę klas. Szablon klasy Stack możemy zapisać następująco:

### Kod programu 8.2

```
1 public:
2   static const size_t N=100;
3 private:
4   T _rep[N];
5   size_t _top;<br>
6 public:
7   Stack():_top(0) {};
8   void push(T val) {_rep[_top++]=val;}
9   T pop() {return _rep[--_top];}
10  bool is_empty {return (_top==0);}
11 };
12
13 // Tak zdefiniowanego szablonu możemy używać podając jawnie jego argumenty:
14
15 Stack<string> st ;
16 st.push("ania");
17 st.push("asia");
18 st.push("basia");
19 while(!st.is_empty() ){
20   cout<<st.pop()<<endl;
21 }
```

Dla szablonów klas nie ma możliwości automatycznej dedukcji argumentów szablonu, ponieważ klasy nie posiadają argumentów wywołania, które mogłyby do tej dedukcji posłużyć. Jest natomiast możliwość podania argumentów domyślnych, wtedy możemy korzystać ze stosu bez podawania argumentów szablonu i wyrażenie poniżej będzie prawidłowe.

### Kod programu 8.3

```
1 ...
2 }
3
4 Stack s;
5 // jest równoważne wyrażeniu:
6 // Stack<int> s;
```

Dla domyślnych argumentów szablonów klas obowiązują te same reguły, co dla domyślnych argumentów wywołania funkcji. Należy pamiętać, że każda konkretyzacja szablonu klasy dla różniących się zestawów argumentów jest osobną klasą.

## 8.3 Pozatypowe parametry szablonów klas

Zestaw możliwych parametrów szablonów klas jest taki sam jak dla szablonów funkcji. Podobnie najczęściej wykorzystywane są wyrażenia całkowitoliczbowe. W naszym przykładzie ze stosem możemy ich użyć do przekazania rozmiaru stosu:

### Kod programu 8.4

```
1 private:
2 T rep[N];
3 size_t top;
4 public:
5 Stack():_top(0) {};
6
7 void push(T val) {_rep[_top++]=val;}
8 T pop() {return rep[--top];}
9 bool is_empty {return (top==0);}
10 }
```

## 8.4 Szablony parametrów szablonu

Stos jest nie tyle strukturą danych, ile sposobem dostępu do nich. Stos realizuje regułę LIFO czyli Last In First Out. W tym sensie nie jest istotne w jaki sposób dane są na stosie przechowywane. Może to być tablica, jak w powyższych przykładach, ale może to być praktycznie dowolny inny kontener. Np. w Standardowej Bibliotece Szablonów C++ stos jest zaimplementowany jako adapter do któregoś z istniejących już kontenerów. Ponieważ kontenery STL są szablonami, szablon adaptera mógłby wyglądać następująco:

### Kod programu 8.5

```
1 template<typename X > class Sequence=std::deque >
2 class Stack {
3     Sequence<T> _rep;
4 public:
5     void push(T e) {_rep.push_back(e);};
6     T pop() {T top=_rep.top();_rep.pop_back();return top;}
7     bool is_empty() const {return _rep.empty();}
8 };
9
10 // Konkretyzując stos możemy wybrać kontener, w którym będą przechowywane jego elementy:
11
12 Stack<double,std::vector> sv;
```

Można zamiast szablonu użyć zwykłego parametru typu:

### Kod programu 8.6

```
1 template<typename T,typename C > class stos {
2     C rep;
3 public:
4     ...
5 }
6 // wykorzystanie przy tym podejściu:
7 stos<double,std::vector<double> > sv;
```

W przypadku użycia szablonu jako parametru szablonu zapewniamy konsystencję pomiędzy typem T i kontenerem C, uniemożliwiając pomyłkę podstawienia niepasujących parametrów. Uczciwość nakazuje jednak w tym miejscu stwierdzić, że właśnie takie rozwiązanie jest zastosowane w STL-u. Ma ono tę zaletę, że możemy adaptować na stos dowolny kontener, niekoniecznie będący szablonem.

## 8.5 Konkretyzacja na żądanie

Jak już wspomniałem wcześniej, konkretyzacja szablonów może odbywać się "na żądanie". W takim przypadku kompilator będzie konkretyzował tylko funkcje napotkane w kodzie. I tak, jeśli np. nie użyjemy w naszym kodzie funkcji `Stack<int>::pop()`, to nie zostanie ona wygenerowana. Można z tego skorzystać i konkretyzować klasy typami, które nie spełniają wszystkich ograniczeń nałożonych na parametry szablonu. Wszystko będzie w porządku jak długo nie będziemy używać funkcji łamiących te ograniczenia. Np. założymy, że do szablonu `Stack` dodajemy możliwość jego sortowania (wiem, to nie jest zgodne z duchem programowania obiektowego, stos nie posiada operacji sortowania, puryści mogą zastąpić ten przykład kontenerem `list`):

### Kod programu 8.7

```
1 bubble_sort(_rep,N);
2 };
3
4 // Możemy teraz np. używać
5 Stack<std::complex<double> > sc;
6 sc.push( std::complex<double>(0,1));
7 sc.pop();
8
9 // ale nie można użyć
10 sc.sort();
11 // konkretyzacja jawna nie powiedzie się, bo kompilator będzie się starał skonkretyzować wszystkie
12 template Stack<std::complex<double> >;
```

## 8.6 Typy stowarzyszone

W klasach poza metodami i polami możemy definiować również typy, które będziemy nazywali stowarzyszonymi z daną klasą. Jest to szczególnie przydatne w przypadku szablonów. Rozważmy następujący przykład:

### Kod programu 8.8

```
1 public:
2 typedef T value_type;
3 ...
4 }
```

Możemy teraz używać jej w innych szablonach:

### Kod programu 8.9

```
1 typename S::value_type total;
2 słowo typename jest wymagane, inaczej kompilator założy, że
3 S::value_type odnosi się do statycznej składowej klasy
4 while(!s.is_empty() ) {
5     total+=s.pop();
6 }
7 return total;
8 }
```

Bez takich możliwości musielibyśmy przekazać typ elementów stosu w osobnym argumencie. Mechanizm typów stowarzyszonych jest bardzo często używany w uogólnionym kodzie.

# Rozdział 9

## Szablony klas

Autorzy:

*Paweł Wnuk*





## Część IV Przykłady



**KAPITAŁ LUDZKI**  
NARODOWA STRATEGIA SPÓJNOŚCI

**UNIA EUROPEJSKA**  
EUROPEJSKI  
FUNDUSZ SPOŁECZNY



## Wstęp

Ostatnia część podręcznika zawiera zestaw mniej lub bardziej skomplikowanych przykładów. Zamieściliśmy je tutaj, by pokazać Wam praktyczne wykorzystanie wiedzy przekazywanej Wam wcześniej, w zadaniach które najczęściej sprawiają problemy studentom, a więc aplikacji wielowątkowej, komunikacji sieciowej, dynamicznego ładowania bibliotek czy też systemów wtyczek [ang. Plugin].

# Rozdział 10

Autorzy:

*Paweł Wnuk*

## 10.1 Aplikacje wielowątkowe

Współczesne procesory praktycznie zawsze są wyposażone w więcej niż jedną jednostkę wykonawczą. W takim wypadku – by w pełni wykorzystać moc takiego procesora, przygotowuje się dla niego aplikacje, które mają więcej niż jeden tok wykonania. Takie toki wykonania nazywa się wątkami. Pamiętajcie:

- Każda uruchomiona instancja programu to proces
- Każdy proces ma co najmniej jeden wątek
- Każdy wątek musi być przypisany do jakiegoś procesu

W ogromnym skrócie – różnica pomiędzy wątkiem a procesem polega głównie na dostępie do pamięci. Każdy proces ma przydzielany własny obszar pamięci na stosie i na sterckie, który nie jest dostępny dla innych procesów. W przypadku wątków sytuacja jest inna – każdy wątek posiada własny stos, lecz stertę współdzieli z innymi wątkami w ramach tego samego procesu. Skoro współdzieli – to zawsze może pojawić się sytuacja, w której dwa wątki próbują zapisać coś do tego samego obszaru pamięci. Konsekwencje tego – jak się zapewne domyślacie – są tragiczne ... dla programu oczywiście ;> Aby się przed tym chronić, stosuje się kilka technik zabezpieczających.

## 10.2 Sekcja krytyczna

Sekcja krytyczna to współbieżny fragment kodu programu, w którym korzysta się z zasobu dzielonego, a co za tym idzie w danej chwili może być wykorzystywany przez co najwyżej jeden wątek. System operacyjny dba o synchronizację, jeśli więcej wątków żąda wykonania kodu sekcji krytycznej, dopuszczany jest tylko jeden wątek, pozostałe zaś są wstrzymywane. Dąży się do tego, aby kod sekcji krytycznej był krótki - by wykonywał się szybko. Sekcje krytyczne realizuje wykorzystując API systemu operacyjnego i są ważne jedynie w obrębie jednego procesu. Brak wzajemnego wykluczania się wykonywania sekcji krytycznych może spowodować błędy wykonania, np. dwukrotne zapisanie danej albo

niepoprawna modyfikacja zasobu. Sekcje krytyczne poza API systemu realizowane mogą być również przez klasy je opakowujące, np. `TCriticalSection` z bibliotek VCL firmy CodeGear. Czasami sekcja krytyczna bywa mylona z nieco bardziej zaawansowanym tworem jakim jest mutex

## 10.3 Mutex

Mutex ang. Mutual Exclusion, oznacza wzajemne wykluczanie. Jest to element stosowany w programowaniu służący do kontroli dostępu do zasobów niemożliwych do współdzielenia, podobny do sekcji krytycznej, z tą podstawową różnicą, że może być wykorzystany również do ochrony zasobów pomiędzy procesami. Implementacja muteksu musi spełniać następujące warunki:

- Niedoprowadzanie do blokady – gdy kilka procesów wyraża chęć wejścia do sekcji krytycznej, jeden z nich musi w końcu wejść.
- Nie może prowadzić do zagłodzenia – jeśli pewien proces wyraża chęć wejścia do sekcji krytycznej, to musi być w końcu dopuszczony. Jest to tak zwana uczciwość. Wyróżnia się:
  - uczciwość słabą, jeśli proces ciągle zgłasza żądanie, to w końcu zostanie dopuszczony do sekcji krytycznej.
  - uczciwość mocną, gdy proces zgłasza żądanie nieskończenie wiele razy, to zostanie w końcu dopuszczony do sekcji krytycznej
  - oczekiwanie liniowe, jeśli proces zgłasza żądanie, będzie dopuszczony do sekcji krytycznej nie później niż po tym jak każdy inny proces zostanie obsłużony jeden raz. (ilość zasobów).
  - FIFO, tak jak w przypadku kolejki (bufora FIFO – pierwszy wszedł, pierwszy wyjdzie), procesy zostaną obsłużone w kolejności zgłoszenia żądań.
- Nie może prowadzić do blokady w przypadku braku współzawodnictwa. Gdy tylko jeden proces wyraża chęć wejścia do sekcji krytycznej, powinien zostać dopuszczony. Jednym ze sposobów na zastosowanie wzajemnego wykluczania jest uniemożliwienie występowania przerwania podczas wykonywania krytycznych fragmentów kodu. Jednak takie rozwiązanie sprawdza się w przypadku systemów jednoprosesorowych. Przy większej ilości procesorów implementuje się pętlę sprawdzającą wartość zmiennej (tzw. flagi) wskazującej na możliwość dostępu do sekcji krytycznej.

Podobnie jak sekcje krytyczne – mutex-y także są zazwyczaj dostępne z poziomu API systemu operacyjnego. Jak sekcję czy też mutex wykorzystuje się do zapewnienia bezpiecznego wielodostępu do obiektów w C++?

## 10.4 Klasa zabezpieczona przed wielodostępem

Poniżej znajdziecie kod klasy bazowej zabezpieczonej przed wielodostępem. Kod został przygotowany do kompilacji warunkowej, wykorzystując API \*nixów lub Windows – więc powinien bez problemu uruchomić się na dowolnym z tych systemów.

## Kod programu 10.1

Nagłówek

```
1  #define CPDU5THREADSAFE_H
2
3  #ifdef __WIN32__
4  #include <windows.h>
5  #else
6  #include <pthread.h>
7  #endif
8
9  /** Klasa strażnika - blokuje dostęp do obiektu w którym jest
10   utworzona instancja klasy aż do momentu skasowania instancji. */
11  #ifdef __WIN32__
12  class CLocker {
13  public:
14      CLocker(CRITICAL_SECTION *_l) {
15          FLock = _l;
16          EnterCriticalSection(FLock);
17      }
18      virtual ~CLocker() {
19          LeaveCriticalSection(FLock);
20      }
21  private:
22      CRITICAL_SECTION *FLock;
23  };
24  #else
25  class CLocker {
26  public:
27      CLocker(pthread_mutex_t *_l) {
28          FLock = _l;
29          pthread_mutex_lock(FLock);
30      }
31      virtual ~CLocker() {
32          pthread_mutex_unlock(FLock);
33      }
34  private:
35      pthread_mutex_t *FLock;
36  };
37  #endif
38
39  /** Klasa bazowa dla klas z zabezpieczeniem przed wielodostępem */
40  class CThreadSafe
41  {
42  public:
43      CThreadSafe();
44      virtual ~CThreadSafe();
45      void lock();
46      void unlock();
47  #ifdef __WIN32__
48      CRITICAL_SECTION* getLock() { return &FLock; }
49  #else
50      pthread_mutex_t* getLock() { return &FLock; }
51  #endif
52  private:
53  #ifdef __WIN32__
54      CRITICAL_SECTION FLock;
```

```
55 #else
56 pthread_mutex_t FLock;
57 #endif
58 };
59
60 #endif
```

### Kod programu 10.2

Implementacja

```
1 CThreadSafe::CThreadSafe()
2 {
3 #ifdef __WIN32__
4 InitializeCriticalSection(&FLock);
5 #else
6 pthread_mutexattr_t a;
7 pthread_mutexattr_init(&a);
8 pthread_mutexattr_settype(&a, PTHREAD_MUTEX_RECURSIVE);
9 pthread_mutex_init(&FLock, &a);
10 #endif
11 }
12
13 CThreadSafe::~CThreadSafe() {
14 #ifdef __WIN32__
15 DeleteCriticalSection(&FLock);
16 #else
17 pthread_mutex_destroy(&FLock);
18 #endif
19 }
20
21 void CThreadSafe::lock() {
22 #ifdef __WIN32__
23 EnterCriticalSection(&FLock);
24 #else
25 pthread_mutex_lock(&FLock);
26 #endif
27 }
28
29 void CThreadSafe::unlock() {
30 #ifdef __WIN32__
31 LeaveCriticalSection(&FLock);
32 #else
33 pthread_mutex_unlock(&FLock);
34 #endif
35 }
```

## 10.5 Wielowątkowa aplikacja sortująca

Na koniec tego rozdziału mam dla Was jeszcze jeden przykład – aplikację która sortuje tablicę w dwóch wątkach, korzystając przy tym z jednego obiektu statystyk. Aplikacja została przygotowana do pracy w środowisku CodeGear RAD Studio.

### Kod programu 10.3

Plik *f\_dane.h*

```

1  #define f_daneH
2
3  #include <SyncObjs.hpp>
4
5  class ICMYUpdate {
6  public:
7      virtual void updateScreen(const int _p) = 0;
8  };
9
10 const int tSize = 200;
11
12 extern int aTab[tSize], bTab[tSize];
13
14 extern TCriticalSection *crsA, *crsB;
15
16 void tRandom(int _t[]);
17 void tSortBubble(int _t[], ICMYUpdate *_u);
18 void tSortSelection(int _t[], ICMYUpdate *_u);
19
20 #endif

```

### Kod programu 10.4

Plik *f\_dane.cpp*

```

1  #include <cstdlib>
2
3  #include "f_dane.h"
4  #include "form_main.h"
5
6  //-----
7
8  #pragma package(smart_init)
9
10 int aTab[tSize], bTab[tSize];
11
12 TCriticalSection *crsA = NULL, *crsB = NULL;
13
14
15 void tRandom(int _t[]) {
16     for (int i=0; i<tSize; i++)
17         _t[i] = std::rand() % 255;
18 }
19
20 void tSortBubble(int _t[], ICMYUpdate* _u) {
21     bool isAnyChange;
22     do {
23         isAnyChange = false;
24         for (int i=0; i<tSize-1; i++)
25             if (_t[i] > _t[i+1]) {
26                 int tt = _t[i];
27                 _t[i] = _t[i+1];
28                 _t[i+1] = tt;
29                 isAnyChange = true;
30                 _u->updateScreen(i/(double)tSize);
31             }
32     } while (isAnyChange);

```



```

33 }
34
35 void tSortSelection(int _t[], ICMYUpdate* _u) {
36     int pmin;
37     for (int i=0; i<tSize-1; i++) {
38         pmin = i;
39         for (int j = i+1; j<tSize; j++)
40             if (_t[j] < _t[pmin])
41                 pmin = j;
42
43         int tt = _t[i];
44         _t[i]=_t[pmin];
45         _t[pmin] = tt;
46         _u->updateScreen(i/(double)tSize);
47     }
48
49 }

```

### Kod programu 10.5

plik *thr\_sort\_bubble.h*

```

1  #ifndef thr_sort_bubbleH
2  #define thr_sort_bubbleH
3  //-----
4  #include <Classes.hpp>
5
6  #include "f_dane.h"
7
8  //-----
9  class ThrMySort : public TThread
10 {
11 private:
12 protected:
13     void __fastcall Execute();
14     void __fastcall updatePicture();
15 public:
16     __fastcall ThrMySort(bool CreateSuspended);
17 };
18 //-----
19 #endif

```

### Kod programu 10.6

plik *thr\_sort\_bubble.cpp*

```

1  //-----
2
3  #include <vcl.h>
4  #pragma hdrstop
5
6  #include "thr_sort_bubble.h"
7  #include "form_main.h"
8  #pragma package(smart_init)
9
10 /*class CMyThrUpdate : public ICMYUpdate {

```

```

11 public:
12     void updateScreen(const int _p) {
13         pThread->Synchronize(updatePicture);
14     }
15     ThrMySort *pThread;
16 };*/
17
18 //-----
19
20 // Important: Methods and properties of objects in VCL can only be
21 // used in a method called using Synchronize, for example:
22 //
23 //     Synchronize(&UpdateCaption);
24 //
25 // where UpdateCaption could look like:
26 //
27 //     void __fastcall ThrMySort::UpdateCaption()
28 //     {
29 //         Form1->Caption = "Updated in a thread";
30 //     }
31 //-----
32
33 __fastcall ThrMySort::ThrMySort(bool CreateSuspended)
34 : TThread(CreateSuspended)
35 {
36 }
37 //-----
38 void __fastcall ThrMySort::Execute()
39 {
40     int* _t = aTab;
41
42     int cntr = 0;
43     bool isAnyChange;
44     do {
45         isAnyChange = false;
46         for (int i=0; i<tSize-1; i++)
47             if (_t[i] > _t[i+1]) {
48                 crsA->Enter();
49                 try {
50                     int tt = _t[i];
51                     _t[i] = _t[i+1];
52                     _t[i+1] = tt;
53                     isAnyChange = true;
54                     //         if ((rand() % 100) == 1)
55                     //             throw new Exception("Niedobrze, ojoj...");
56                 } __finally {
57                     crsA->Leave();
58                 }
59
60                 Synchronize(&updatePicture);
61             }
62         //         if (++cntr % 10 == 9)
63         //             Suspend();
64
65     } while (isAnyChange && !Terminated);
66
67 }
68 //-----

```

```

69 void __fastcall ThrMySort::updatePicture() {
70     Form1->PaintBox1->Refresh();
71 }

```

### Kod programu 10.7

plik `thr_sort_selection.h`

```

1  #ifndef thr_sort_selectionH
2  #define thr_sort_selectionH
3  //-----
4  #include <Classes.hpp>
5
6  #include "f_dane.h"
7
8  //-----
9  class ThrSortSelection : public TThread
10 {
11 private:
12     int tTab[tSize];
13 protected:
14     void __fastcall Execute();
15     void __fastcall updatePicture();
16 public:
17     __fastcall ThrSortSelection(bool CreateSuspended);
18 };
19 //-----
20 #endif

```

### Kod programu 10.8

Plik `thr_sort_selection.cpp`

```

1  #include <vcl.h>
2  #pragma hdrstop
3
4  #include "thr_sort_selection.h"
5  #include "form_main.h"
6  #pragma package(smart_init)
7  //-----
8
9  // Important: Methods and properties of objects in VCL can only be
10 // used in a method called using Synchronize, for example:
11 //
12 //     Synchronize(&UpdateCaption);
13 //
14 // where UpdateCaption could look like:
15 //
16 //     void __fastcall ThrSortSelection::UpdateCaption()
17 //     {
18 //         Form1->Caption = "Updated in a thread";
19 //     }
20 //-----
21
22 __fastcall ThrSortSelection::ThrSortSelection(bool CreateSuspended)
23     : TThread(CreateSuspended)

```

```
24 {
25     memcpy(tTab, bTab, tSize*sizeof(int));
26 }
27 //-----
28 void __fastcall ThrSortSelection::Execute()
29 {
30     int* _t = tTab;
31
32     int pmin;
33     for (int i=0; i<tSize-1; i++) {
34         pmin = i;
35         for (int j = i+1; j<tSize; j++)
36             if (_t[j] < _t[pmin])
37                 pmin = j;
38
39         int tt = _t[i];
40         _t[i]=_t[pmin];
41         _t[pmin] = tt;
42         Synchronize(&updatePicture);
43
44     }
45 }
46 //-----
47
48 void __fastcall ThrSortSelection::updatePicture() {
49     memcpy(bTab, tTab, tSize*sizeof(int));
50     Form1->PaintBox2->Refresh();
51 }
```

# Rozdział 11

## Programowanie gniazd sieciowych

Autorzy:

*Paweł Wnuk*

### 11.1 Protokół sieciowy

Większość aplikacji sieciowych komunikuje się przy wykorzystaniu jakiegoś protokołu. Poniżej zamieszczam Wam przykład tworzenia takiego protokołu:

#### 11.1.1 Klasa protokołu

W tym punkcie powinniście dokonać określenia klasy, do jakiej będzie się zaliczał Wasz protokół. Dla ułatwienia podam tutaj krótki opis klas protokołów:

##### **Klasyczne protokoły tekstowe**

Wszystkie polecenia i odpowiedzi są zapisywane w postać tekstowych komend (np. HTTP, FTP, NNTP, POP3, SMTP, ...).

Podstawowe zalety takiego rozwiązania:

- Protokół jest zrozumiały dla człowieka
- Brak problemów z konwersją danych między urządzeniami i systemami różnego typu
- Łatwy debugging

Główne wady:

- Mało efektywne (duży narzut w ilości wysyłanych danych, np. przesłanie liczby typu long to binarnie 4 bajty, a w protokole tekstowym do 18 bajtów)
- Konieczność parsowania i budowania komunikatów tekstowych
- problemy z przesyłaniem skomplikowanych struktur danych

## Protokoły oparte o XML

Wszystkie polecenia i odpowiedzi są przesyłane tekstem sformatowanym wg reguł XML (przykład – XMPP). Takie podejście pozwala zachować zalety protokołów tekstowych (jedynie nieznacznie utrudniając ich czytanie przez człowieka), dodatkowo rozszerzając listę zalet o:

- prostą kontrolę poprawności przesyłanych komunikatów (w oparciu o istniejące narzędzia i schematy XML)
- sporą liczbę standardów opisujących treści określonego typu
- prostą obsługę wielu kodowań znaków
- prostą obsługę złożonych struktur danych

Główne wady:

- Jeszcze mniej efektywne niż tekstowe (w sensie ilości przesyłanych danych)
- Pozostający narzut obliczeniowy

## Protokoły XML udostępniające RPC

Ta klasa może być traktowana jako podklasa poprzedniej, jednakże ze względu na dostępność narzędzi programistycznych można ją wyróżnić. Ogólnie w przypadku tych protokołów udostępniana jest możliwość zdalnego wywołania procedury / funkcji (przez klienta na serwerze) oraz odebrania wyników jej działania. Wszystkie takie protokoły umożliwiają w miarę swobodne definiowanie parametrów (nawet obiektowych). Inna nazwa pod którą występują protokoły tej klasy to Web Services. Przykłady – SOAP, XML-RPC

Zalety:

- duża standaryzacja (opis w języku WSDL)
- duża liczba narzędzi
- generatory kodu, które z WSDL generują kod dla wielu różnych języków
- możliwość korzystania z portu 80 i protokołu HTTP jako transportowego – nie ma problemów z firewall-ami

Wady:

- są bezpołączeniowe – każde zapytanie to autoryzacja
- rośnie narzut czasowy

## Protokoły binarne dedykowane

W przypadku takich protokołów znaczenie każdego bajtu w ramce określa programista. Przykładem jest tutaj choćby sam TCP – będący binarnym protokołem pracującym przy

wykorzystaniu protokołu IP. Wy również możecie przygotować własny protokół tego typu – wykorzystując TCP lub UDP.

#### Protokoły binarne udostępniające RPC

Różne techniki umożliwiające zdalne wywołanie procedur między komponentami w sieciach komputerowych, wykorzystując przy tym strumienie bitów trudno dekodowane przez człowieka. Przykłady – CORBA, RMI, AMF. Zazwyczaj nieprzenośne między językami programowania, ale pozwalają na pominięcie części lub całości zagadnień związanych z komunikacją sieciową.

#### Protokoły oparte na ASN.1 (Abstract Syntax Notation One)

ASN.1 jest standardem służącym do opisu metod kodowania, dekodowania i przesyłania danych. Jest to obecnie standard ISO/IEC 8824. Pozwala on zdefiniować za pomocą sformalizowanego opisu składnie pól w komunikatach, a następnie wygenerować kod serializujący i deserializujący te pakiety. Standard ASN.1 nie definiuje bezpośrednio binarnego formatu przesyłanych komunikatów. Mogą być one serializowalne według jednej z proponowanych zasad. W szczególności istnieją trzy najpopularniejsze możliwości w tej kwestii:

- BER - (Basic encoding rules) - zapamiętuje każde pole w postaci binarnej jako trójkę: znacznik, długość, wartość.
- PER - (Packed encoding rules) - podobnie jak BER, ale metoda bardzo zwraca uwagę na efektywność pod względem rozmiaru pakietów.
- XER (XML encoding rules) - komunikaty są przesyłane w postaci paczek XML.

Dla Was może to być nieco skomplikowane narzędzie, szczególnie że nie ma jeszcze dużej ilości łatwo dostępnych generatorów. Przykład – LDAP

### 11.1.2 Protokół przykładowego programu

#### Wybór klasy

W omawianym zastosowaniu przesyłamy jedną do kilkunastu – kilkudziesięciu sekwencji znaków reprezentujących tekst, co w sposób naturalny preferuje zastosowanie protokołów tekstowych. Protokoły oparte o XML jest sens stosować gdy wymagane jest przesłanie informacji o złożonej strukturze (np. obiektów). W naszym przypadku przesyłane są tylko i wyłącznie typy proste, co oznacza, że najlepszym wyborem będzie protokół oparty na liniach.

#### Jednostka logiczna dla komunikacji w protokole

Jednostka logiczna to spójny ciąg danych o ściśle określonym formacie. W przypadku protokołów tekstowych – zazwyczaj jest to linia zakończona znakiem końca linii \n. W przypadku protokołów binarnych – jest to pakiet, często nazywany także paczką (by odróżnić go od pakietu definiowanego w TCP/IP i podkreślić ich wzajemną niezależność).

Ten protokół jako jednostkę logiczną stosuje linię którą będziemy nazywali komunikatem.

#### Typ komunikacji



W zasadzie dostępne mamy dwa główne znane typy komunikacji – pytanie / odpowiedź i praca peer-to-peer. Przykładowy protokół jest oparty o pary pytanie / odpowiedź. Klient wysyła zapytanie zawierające jedno z dwóch obsługiwanych poleceń, i otrzymuje na nie odpowiedź. Odpowiedź jest obowiązkowa – nie ma w protokole przewidzianych zapytań bez odpowiedzi. Każdy cykl komunikacyjny jest inicjowany przez klienta. Nie przewiduje się wbudowanych w protokół mechanizmów delegacji, przekierowania czy też tunelowania zapytań i odpowiedzi.

### Struktura komunikatu

Jest tylko jedna dopuszczalna struktura komunikatu. Komunikat zawsze zaczyna się kodem polecenia, po którym występuje od zera do wielu parametrów. Parametry nie są nazywane – o ich znaczeniu decyduje kolejność pojawiania się w strukturze komunikatu. Każde polecenie ma zdefiniowany dopuszczalny zakres liczby parametrów. Poszczególne elementy komunikatu są oddzielone znakiem „#”. Przyjmując, że para [] oznacza dokładnie jeden element, para {} od zera do nieskończenia wiele elementów, LF – znak końca linii, strukturę pojedynczego komunikatu można przedstawić następująco:

[kod polecenia]#{parametr\_n#}LF

Znak „#” jest znakiem niedozwolonym w wartości parametrów oraz kodach poleceń. Drugim znakiem zabronionym jest znak końca linii. Pozostałe znaki są dozwolone.

Dopuszcza się stosowanie jedynie kodowania UTF8 do reprezentacji znaków – dzięki temu nie musimy uzgadniać kodowania między klientem a serwerem.

### Lista poleceń

Zaprezentowana poniżej lista poleceń jest mocno ograniczona – protokół jest bezstanowy.

Tabela zawiera kompletny wykaz komunikatów wraz z kierunkiem przepływu ( $S \rightarrow C$  oznacza od serwera do klienta, \* oznacza oba kierunki) oraz listą parametrów

Nazwa

Kod

Znaczenie

Kierunek

Liczba parametrów

Opis parametrów

C-HELLO

HEL

Powitanie

$C \rightarrow S$

1

Nazwa klienta – musi być unikalna

C-ACK

ACK

Potwierdzenie

\*

0

C-REJECT

RJC

Odrzucenie prośby

\*

0

C-MESSAGE

MSG

Wysłanie komunikatu do jednego odbiorcy

$C \rightarrow S$

2

1 – nazwa odbiorcy

2 – treść komunikatu

C-BROADCAST

BRD

Rozgłoszenie komunikatu do wszystkich odbiorców

$C \rightarrow S$

1

1 – treść komunikatu

C-WHOIS

WHO

Pobranie listy użytkowników podpiętych do serwera

$C \rightarrow S$

0

C-WHOIS-RESPONSE

WHR

Odebranie listy odbiorców

$S \rightarrow C$

\*

Zwracanych jest od zera do bardzo wielu odbiorców

C-CHECK

CHK

Sprawdzenie czy jest jakiś komunikat dla mnie

$C \rightarrow S$

0

C-CHECK-RESPONSE

CHR

Odpowiedź zawierająca listę komunikatów dla danego klienta

$S \rightarrow C$

\*

Lista par nadawca – komunikat.

C-PROTO-ERR

ERR

Błąd protokołu

\*

0

## 11.2 Przykładowy program klienta

### Kod programu 11.1

Klient małego GG

```
1  #include <iostream>
2  #include <string>
3  #include <windows.h>
4  #include <winsock2.h>
5  #include <stdio.h>
6
7  using namespace std;
8
9  #define g_PORT 7401 // port do połączenia
10 #define g_BSIZE 512 // rozmiar bufora
11
12 int main()
13 {
14     cout << "Serwer malego GG" << endl;
15
16     std::string serverIPAddress = "127.0.0.1";
17     /// @todo Przydałoby się poprosić użytkownika o adres
18
19     WSADATA wsaData;
20     int iResult;
21
22     // To specyficzne dla windows
23     iResult = WSASStartup(MAKEWORD(2, 2), &wsaData);
24     if (iResult != 0)
25     {
26         cout << "WSASStartup nie udało się: " << iResult << endl;
27         return 1;
28     }
29
```

```

30  int sock_fd;           // deskryptor gniazda
31
32  struct sockaddr_in srv_addr;  // mój adres
33
34  // utworzenie gniazda TCP
35  if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == SOCKET_ERROR)
36  {
37      cout << "Bład tworzenia gniazda\n";
38      WSACleanup();
39      exit(1);
40  }
41
42  // ustawienie adresu
43  srv_addr.sin_family = AF_INET;  // INET - czyli na dziś jedyna słuszna opcja
44  srv_addr.sin_port = htons(g_PORT); // port musi być konwertowany
45  srv_addr.sin_addr.s_addr = inet_addr(serverIPAddress.c_str());
46  memset(&(srv_addr.sin_zero), '\0', 8); // zerowanie reszty
47
48  // połączenie
49  if (connect(sock_fd, (struct sockaddr *)&srv_addr, sizeof(struct sockaddr)) == SOCKET_ERROR)
50  {
51      cout << "Bład połączenia\n";
52      WSACleanup();
53      exit(1);
54  }
55
56  // tu już mamy połączenie - można wysłać hello
57  char buf[g_BSIZE];
58  sprintf(buf, "HEL#janic#\n");
59  if (send(sock_fd, buf, strlen(buf), 0) == SOCKET_ERROR)
60  {
61      cout << "Bład wysyłania\n";
62      WSACleanup();
63      exit(1);
64  }
65
66  // odebranie odpowiedzi
67  int nbytes = recv(sock_fd, buf, g_BSIZE, 0);
68  if (nbytes == SOCKET_ERROR)
69  {
70      cout << "Bład odbierania odpowiedzi\n";
71      WSACleanup();
72      exit(1);
73  }
74  buf[nbytes] = '\0';
75  // wyświetlamy co odebraliśmy
76  cout << buf << endl;
77
78  /// @todo tu jest miejsce na Waszą inwencję ;>
79  // przykład sprawdzenia kto jest na sieci
80  sprintf(buf, "WHO#\n");
81  if (send(sock_fd, buf, strlen(buf), 0) == SOCKET_ERROR) {
82      cerr << "Bład wysyłania\n";
83      WSACleanup();
84      exit(1);
85  }
86  nbytes = recv(sock_fd, buf, g_BSIZE, 0); // odpowiedź
87  if (nbytes == SOCKET_ERROR) {

```

```
88     cerr << "Bład odbierania odpowiedzi\n";
89     WSACleanup();
90     exit(1);
91 }
92 buf[nbytes] = '\0';
93 cout << buf << endl;
94
95 // przykład wysłania komunikatu do wszystkich:
96 sprintf(buf, "BRD#Witam wszystkich obecnych#\n");
97 if (send(sock_fd, buf, strlen(buf), 0)==SOCKET_ERROR) {
98     cerr << "Bład wysyłania\n";
99     WSACleanup();
100    exit(1);
101 }
102 nbytes = recv(sock_fd, buf, g_BSIZE, 0); // odpowiedź
103 if (nbytes==SOCKET_ERROR) {
104     cerr << "Bład odbierania odpowiedzi\n";
105     WSACleanup();
106     exit(1);
107 }
108 buf[nbytes] = '\0';
109 cout << buf << endl;
110
111 // przykład sprawdzenia czy jest coś dla mnie
112 sprintf(buf, "CHK#\n");
113 if (send(sock_fd, buf, strlen(buf), 0)==SOCKET_ERROR) {
114     cerr << "Bład wysyłania\n";
115     WSACleanup();
116     exit(1);
117 }
118 nbytes = recv(sock_fd, buf, g_BSIZE, 0); // odpowiedź
119 if (nbytes==SOCKET_ERROR) {
120     cerr << "Bład odbierania odpowiedzi\n";
121     WSACleanup();
122     exit(1);
123 }
124 buf[nbytes] = '\0';
125 cout << buf << endl;
126
127 closesocket(sock_fd);
128 WSACleanup();
129 return 0;
130 }
```

## 11.3 Przykładowy program serwera

### Kod programu 11.2

Serwer małego GG

```
1 * File:   CDemoParser.h
2 * Author: Paweł Wnuk
3 *
4 */
5
```

```

6  #ifndef CDEMOPARSER_H
7  #define CDEMOPARSER_H
8
9  #include <string>
10 #include <deque>
11
12 #define DEBUG_LEVEL 4
13
14 enum EnCommand {
15     cmdUnknown, cmdProtocolError, cmdHello, cmdAccept, cmdReject, cmdMessage, cmdBroadcast, cmdWhois,
16 };
17
18 /** Pokazany niżej parser jest bardzo prosty - w zasadzie dzieli linię na fragmenty wyznaczone prze
19 class CDemoParser {
20 public:
21
22     /** Funkcja parsująca komunikat:
23     *
24     * Funkcja parsuje przekazany jej komunikat, zwracając true jeśli parsowanie
25     * się powiodło, false w przeciwnym wypadku.
26     *
27     * Parsowanie sprawdza poprawność składniową komunikatu, oraz testuje liczbę
28     * parametrów.
29     *
30     * @param _src komunikat źródłowy
31     * @param _command wynik parsowania - polecenie (parametr wyjściowy)
32     * @param _params wynik parsowania - parametry (parametr wyjściowy)
33     *
34     * @return true w przypadku powodzenia, false - w pozostałych przypadkach */
35     static bool parseLine(const std::string& _src, EnCommand& _command, std::deque<std::string>& _para
36
37     /** Funkcja formatująca komunikat:
38     *
39     * Funkcja formatuje przekazane polecenie wraz z parametrami i zwraca przygotowany
40     * komunikat. Liczba przekazanych parametrów nie jest sprawdzana.
41     *
42     * @param _command polecenie
43     * @param _params parametry
44     *
45     * @return komunikat w formacie przykładowego protokołu */
46     static std::string formatLine(const EnCommand _command, const std::deque<std::string>& _params);
47 private:
48
49 };
50
51 #endif /* CDEMOPARSER_H */
52
53
54 /**
55 * File: CDemoParser.cpp
56 * Author: Paweł Wnuk
57 *
58 */
59
60 #include <cstdlib>
61 #include <string>
62 #include <deque>
63

```

```

64 #include "CDemoParser.h"
65
66 /// Zakazany znak protokołu
67 const std::string BREAK_CHAR = "#";
68
69 // polecenia
70 const std::string C_HELLO = "HEL";
71 const std::string C_ACK = "ACK";
72 const std::string C_REJECT = "RJC";
73 const std::string C_MESSAGE = "MSG";
74 const std::string C_BROADCAST = "BRD";
75 const std::string C_WHOIS = "WHO";
76 const std::string C_WHOIS_RESPONSE = "WHR";
77 const std::string C_CHECK = "CHK";
78 const std::string C_CHECK_RESPONSE = "CHR";
79 const std::string C_PROTO_ERR = "ERR";
80
81 bool CDemoParser::parseLine(const std::string& _src, EnCommand& _command, std::deque<std::string>&
82     size_t cp;
83     _command = cmdUnknown;
84     _params.clear();
85
86     if (_src.size() < 3) // błąd - w tak krótkim łańcuchu nie da się zapisać polecenia
87         return false;
88
89     cp = _src.find(BREAK_CHAR, 0);
90     if (cp==std::string::npos) {
91         // błąd - brak znaku przerywnika
92         return false;
93     }
94
95     // Najpierw następuje podział linii na polecenie i parametry
96
97     // pierwsze polecenie
98     std::string cmd = _src.substr(0, cp);
99     // potem ewentualnie parametry
100    size_t ck = _src.find(BREAK_CHAR, cp+1);
101    while (ck!=std::string::npos) {
102        _params.push_back(_src.substr(cp+1, ck-cp-1));
103        cp = ck;
104        ck = _src.find(BREAK_CHAR, cp+1);
105    }
106
107    // Potem - rozpoznajemy polecenia i sprawdzamy, czy zgadza się liczba parametrów
108
109    // rozpoznanie i test liczby parametrów
110    if (C_PROTO_ERR == cmd) {
111        _command = cmdProtocolError;
112        // nie sprawdzamy liczby parametrów - bo i tak mamy błąd
113        // następne polecenia są bez parametrów
114    } else if (C_ACK == cmd) {
115        _command = cmdAccept;
116        if (!_params.empty())
117            return false;
118    } else if (C_REJECT == cmd) {
119        _command = cmdReject;
120        if (!_params.empty())
121            return false;

```



```

122 } else if (C_WHOIS == cmd) {
123     _command = cmdWhois;
124     if (!_params.empty())
125         return false;
126 } else if (C_CHECK == cmd) {
127     _command = cmdCheck;
128     if (!_params.empty())
129         return false;
130     // następne polecenia mają dokładnie jeden parametr
131 } else if (C_HELLO == cmd) {
132     _command = cmdHello;
133     if (_params.size() != 1)
134         return false;
135 } else if (C_BROADCAST == cmd) {
136     _command = cmdBroadcast;
137     if (_params.size() != 1)
138         return false;
139     // polecenia o dwóch parametrach
140 } else if (C_MESSAGE == cmd) {
141     _command = cmdMessage;
142     if (_params.size() != 2)
143         return false;
144     // polecenia o otwartej liście parametrów
145 } else if (C_WHOIS_RESPONSE == cmd) {
146     _command = cmdWhoisResponse;
147 } else if (C_CHECK_RESPONSE == cmd) {
148     _command = cmdCheckResponse;
149 } else { // nie rozpoznane polecenie
150     _command = cmdUnknown;
151     return false;
152 }
153
154 return true;
155 }
156
157 std::string CDemoParser::formatLine(const EnCommand _command, const std::deque<std::string>& _param
158 std::string rval;
159 switch(_command) {
160     case cmdProtocolError : rval = C_PROTO_ERR; break;
161     case cmdHello : rval = C_HELLO; break;
162     case cmdAccept : rval = C_ACK; break;
163     case cmdReject : rval = C_REJECT; break;
164     case cmdMessage : rval = C_MESSAGE; break;
165     case cmdBroadcast : rval = C_BROADCAST; break;
166     case cmdWhois : rval = C_WHOIS; break;
167     case cmdWhoisResponse : rval = C_WHOIS_RESPONSE; break;
168     case cmdCheck : rval = C_CHECK; break;
169     case cmdCheckResponse : rval = C_CHECK_RESPONSE; break;
170     default : return "";
171 }
172 // teoretycznie - to tu też powinniśmy jeszcze sprawdzić czy liczba parametrów jest odpowiednia.
173 rval += BREAK_CHAR;
174 for(size_t i=0; i<_params.size(); i++) {
175     rval += _params[i] + BREAK_CHAR;
176 }
177 rval += "\n";
178 return rval;
179 }

```

```

180
181 /*
182  * File:   CDemoSrvSocket.h
183  * Author: Paweł Wnuk
184  *
185  */
186
187 #ifndef CAGENTSrvSOCKET_H
188 #define CAGENTSrvSOCKET_H
189
190 #include <TcpSocket.h>
191 #include <ISocketHandler.h>
192
193 #include <string>
194
195 /** Implementacja przykładowego serwera - nie jest specjalnie skomplikowana,
196  * ponieważ większość logiki jest zaimplementowana w klasach parsera oraz
197  * mapy użytkowników.
198  *
199  * Ta klasa odpowiada w zasadzie jedynie za pobranie linii i przygotowania
200  * odpowiedzi.
201  */
202 class CDemoSrvSocket : public TcpSocket {
203 public:
204     CDemoSrvSocket(ISocketHandler &_h);
205     virtual ~CDemoSrvSocket();
206
207     /** Funkcja wywoływana po każdym przyjsciu nowej linii. Działanie funkcji zależy
208     * od stanu serwera
209     *
210     * @param _line Linia z komunikatem */
211     void OnLine(const std::string& _line);
212
213     /** Funkcja wywoływana po rozłączeniu się klienta
214     *
215     * @param _line Linia z komunikatem */
216     void OnDisconnect();
217 };
218
219 #endif /* CAGENTSrvSOCKET_H */
220
221 /*
222  * File:   CDemoSrvSocket.cpp
223  * Author: Paweł Wnuk
224  *
225  */
226
227 #include <iostream>
228 #include <set>
229
230 #include "CDemoSrvSocket.h"
231 #include "CDemoParser.h"
232 #include "CDemoUsersMap.h"
233
234 #define RSIZE TCP_BUF_SIZE_READ
235
236 using namespace std;
237

```

```

238 CDemoSrvSocket::CDemoSrvSocket(ISocketHandler &_h) : TcpSocket(_h) {
239     // ustawiamy protokół oparty o linie
240     SetLineProtocol();
241 }
242
243 CDemoSrvSocket::~CDemoSrvSocket() {
244 }
245
246 void CDemoSrvSocket::OnLine(const std::string& _line)
247 {
248     // Definicja makra DEBUG_LEVEL jest w pliku nagłówka:
249     #if DEBUG_LEVEL > 2
250     cout << this->GetSocket() << ": przyszło: " << _line << endl; // drukowanie do celów inf.
251     #endif
252     // uzyskanie instancji mapy użytkowników - przyda się później
253     CDemoUsersMap *pUM = CDemoUsersMap::getInstance();
254
255     // zmienne pomocnicze
256     EnCommand command, resp_command = cmdUnknown;
257     std::deque<std::string> args, resp_args;
258     std::string response;
259
260     // parsowanie komunikatu
261     if (!CDemoParser::parseLine(_line, command, args)) {
262         // jeśli parsowanie nie powiodło się - to odsyłamy błąd protokołu
263         std::cout << "Błędny format polecenia: " << _line << endl;
264         resp_command = cmdProtocolError;
265     } else {
266         // w przeciwnym wypadku - rozpoznajemy polecenie i je wykonujemy
267         switch (command) {
268             case cmdHello: // Nowy użytkownik na serwerze - dodajemy jego nazwę do
269                 // listy wszystkich użytkowników.
270                 // przy dodawaniu użytkownika wykorzystujemy także numer gniazda
271                 // - potrzebne potem do rozpoznania z kim mamy do czynienia
272                 if (pUM->addUser(this->GetSocket(), args[0])) {
273                     // jeśli dodanie się powiedzie - nazwa użytkownika jest ok
274                     resp_command = cmdAccept;
275                 } else {
276                     // Duplikat nazwy użytkownika - odsyłamy informację o błędzie
277                     resp_command = cmdReject;
278                 };
279                 break;
280             case cmdMessage: // Przyszedł komunikat do określonego odbiorcy
281                 if (pUM->addMessage(this->GetSocket(), args[0], args[1])) { // jeśli jest taki odbiorca
282                     resp_command = cmdAccept;
283                 } else { // brak odbiorcy
284                     resp_command = cmdReject;
285                 };
286                 break;
287             case cmdBroadcast: // przyszedł komunikat rozgłoszeniowy
288                 pUM->addBroadcast(this->GetSocket(), args[0]); // tu nie sprawdzamy czy jest jakiś odbiorca
289                 resp_command = cmdAccept;
290                 break;
291             case cmdWhois: // prośba o listę osób na serwerze
292                 pUM->fillUsers(resp_args); // przygotowanie takiej listy
293                 resp_command = cmdWhoisResponse;
294                 break;
295             case cmdCheck: // sprawdzenie, czy dla danego nadawcy nie ma jakiegoś komunikatu

```

```

296     pUM->fillMessages(this->GetSocket(), resp_args);
297     resp_command = cmdCheckResponse;
298     break;
299     default: // nigdy nie może się zdarzyć
300         cerr << "Zgłupiałem ...\n";
301         resp_command = cmdProtocolError;
302     };
303 }
304
305 response = CDemoParser::formatLine(resp_command, resp_args);
306 #if DEBUG_LEVEL > 2
307 cout << response; /// drukowanie na ekranie przygotowanej odpowiedzi
308 #endif
309 if (resp_command != cmdUnknown)
310     Send(response);
311 }
312
313 void CDemoSrvSocket::OnDisconnect() {
314     #if DEBUG_LEVEL > 2
315         cout << this->GetSocket() << ": rozłączenie\n"; // drukowanie do celów inf.
316     #endif
317     // wyrzucamy nazwę użytkownika z listy
318     CDemoUsersMap::getInstance()->delUser(this->GetSocket());
319 }
320
321 /*
322 * File: CDemoUsersMap.h
323 * Author: Paweł Wnuk
324 *
325 */
326
327 #ifndef CDEMOTRANSLATIONSMAP_H
328 #define CDEMOTRANSLATIONSMAP_H
329
330 #include <string>
331 #include <map>
332 #include <set>
333 #include <deque>
334
335 class CDemoSrvSocket;
336
337 /** Prosta struktura wykorzystywana do pamiętania pojedynczego komunikatu */
338 struct SMessageWithSender {
339     std::string Sender;
340     std::string Message;
341 };
342
343 /** Klasa zawierająca mapę użytkowników na serwerze - singleton.
344 *
345 * Jest to typowy przykład implementacji singletonu - mapa użytkowników w programie
346 * musi być dokładnie jedna. Odpowiedzialnością tej klasy jest przechowywanie
347 * zarówno listy wszystkich użytkowników, jak i wszystkich komunikatów jeszcze
348 * nie przesłanych do odbiorcy.
349 *
350 * Użytkownicy są pamiętani wraz z gniazdem na którym są połączeni.
351 */
352 class CDemoUsersMap {
353 public:

```

```

354 static CDemoUsersMap* getInstance();
355 static void freeInstance();
356
357 /** Dodanie użytkownika połączonego na danym gnieździe do mapy */
358 bool addUser(int _sc, std::string _un);
359 /** Dodanie komunikatu dla danego użytkownika. Numer gniazda jest nam
360 * potrzebny do rozpoznania kto jest nadawcą */
361 bool addMessage(int _sc, std::string _to, std::string _msg);
362 /** Wysłanie komunikatu do wszystkich odbiorców. */
363 void addBroadcast(int _sc, std::string _msg);
364
365 /** Pobranie listy wszystkich użytkowników.
366 * @param _list - jest to parametr wyjściowy, w nim zamieścimy wynik */
367 void fillUsers(std::deque<std::string>& _list);
368 /** Pobranie listy wszystkich komunikatów dla danego użytkownika.
369 * użytkownik jest rozpoznawany przez numer gniazda na którym jest podłączony
370 * @param _sc gniazdo
371 * @param _list Lista komunikatów dla danego użytkownika. Lista składa się
372 * z kolejno występujących po sobie nazwy użytkownika oraz treści komunikatu */
373 void fillMessages(int _sc, std::deque<std::string>& _list);
374 /** Usunięcie użytkownika z listy
375 * @param _sc gniazdo */
376 void delUser(int _sc);
377
378 private:
379 CDemoUsersMap();
380 virtual ~CDemoUsersMap();
381
382 std::string getUserNamForSocket(int _sc);
383
384 void lock();
385 void unLock();
386
387 static CDemoUsersMap* FInstance;
388
389 std::map<std::string, int> FUserNames;
390 std::map<int, std::string> FSocket2UserName;
391 std::map<std::string, std::deque<SMessageWithSender> > FWaitingMessages;
392 };
393
394 #endif /* CDEMOTRANSLATIONSMAP_H */
395
396
397 /*
398 * File: CDemoTranslationsMap.cpp
399 * Author: czesio
400 *
401 * Created on 28 kwiecień 2011, 10:18
402 */
403 #include <cstdlib>
404 #include <iostream>
405 #include "CDemoUsersMap.h"
406
407 CDemoUsersMap* CDemoUsersMap::FInstance = NULL;
408
409
410 CDemoUsersMap* CDemoUsersMap::getInstance() {
411 if (!FInstance)

```

```

412   FInstance = new CDemoUsersMap();
413   return FInstance;
414 }
415
416 void CDemoUsersMap::freeInstance() {
417     if (FInstance)
418         delete FInstance;
419     FInstance = NULL;
420 }
421
422
423 bool CDemoUsersMap::addUser(int _sc, std::string _un) {
424     if (FUserNames.find(_un) != FUserNames.end()) // nazwa użytkownika jest już wykorzystana
425         return false;
426     // nazwa jeszcze nie jest wykorzystana - przypisujemy ją do gniazda
427     FUserNames[_un] = _sc;
428     // dodajemy jeszcze mapowanie na potrzebę szybkiego wyszukiwania nazwy użytkownika
429     // na podstawie jego numeru gniazda
430     FSocket2UserName[_sc] = _un;
431     return true;
432 }
433
434 void CDemoUsersMap::delUser(int _sc) {
435     std::string un = FSocket2UserName[_sc];
436     FSocket2UserName.erase(_sc);
437     FUserNames.erase(un);
438 }
439
440 bool CDemoUsersMap::addMessage(int _sc, std::string _to, std::string _msg) {
441     SMessageWithSender mss;
442     mss.Sender = getUserForSocket(_sc);
443     // jeśli nie znaleziono użytkownika przypisanego do gniazda - to wychodzimy
444     if (mss.Sender == "")
445         return false;
446     // w przeciwnym wypadku możemy zaakceptować komunikat
447     mss.Message = _msg;
448     FWaitingMessages[_to].push_back(mss);
449     return true;
450 }
451
452 void CDemoUsersMap::addBroadcast(int _sc, std::string _msg) {
453     if (FSocket2UserName.find(_sc) == FSocket2UserName.end()) // nie ma takiego
454         return;
455     for (std::map<std::string, int>::const_iterator it = FUserNames.begin(); it != FUserNames.end(); it++)
456         addMessage(_sc, it->first, _msg);
457 }
458 }
459
460 void CDemoUsersMap::fillUsers(std::deque<std::string>& _list) {
461     _list.clear();
462     for (std::map<std::string, int>::const_iterator it = FUserNames.begin(); it != FUserNames.end(); it++)
463         _list.push_back(it->first);
464 }
465 }
466
467 void CDemoUsersMap::fillMessages(int _sc, std::deque<std::string>& _list) {
468     _list.clear();
469     std::string nm = getUserForSocket(_sc);

```



```

470 for (size_t i=0; i<FWaitingMessages[nm].size(); i++) {
471     _list.push_back(FWaitingMessages[nm][i].Sender);
472     _list.push_back(FWaitingMessages[nm][i].Message);
473 }
474 FWaitingMessages[nm].clear();
475 }
476
477
478 CDemoUsersMap::CDemoUsersMap() {
479 }
480
481 CDemoUsersMap::~CDemoUsersMap() {
482 }
483
484 void CDemoUsersMap::lock() {
485 }
486
487 void CDemoUsersMap::unlock() {
488 }
489
490 std::string CDemoUsersMap::getUserNameForSocket(int _sc) {
491     if (FSocket2UserName.find(_sc) == FSocket2UserName.end()) // nie ma takiego
492         return "";
493     return FSocket2UserName[_sc]; // w przeciwnym przypadku zwracamy nazwę użytkownika
494 }
495
496
497 /*
498 * File:   main.cpp
499 * Author: czesio
500 *
501 * Created on 12 luty 2011, 09:59
502 */
503
504 #include <cstdlib>
505 #include <iostream>
506 #include <map>
507 #include <string>
508
509 #include <SocketHandler.h>
510 #include <ListenSocket.h>
511
512 #include "CDemoSrvSocket.h"
513
514 #define g_PORT_NUMBER 7401
515
516 using namespace std;
517
518 bool koniec = false;
519
520 int main(int argc, char** argv) {
521
522     cout << "Serwer Malego Gadania gotowy:\n";
523     cout << "Slucham na porcie " << g_PORT_NUMBER << endl;
524     cout.flush();
525
526     // utworzenie obiektu zarządzającego połączeniami sieciowymi
527     SocketHandler h;

```



```
528 ListenSocket<CDemoSrvSocket> l(h);
529
530 if (l.Bind(g_PORT_NUMBER)) {
531     exit(-1);
532 }
533 h.Add(&l);
534 h.Select(1,0);
535 while (!koniec) {
536     h.Select(1,0);
537 }
538
539 return 0;
540 }
```

---

# Słownik

## Logika rozmyta

Jedna z logik wielowartościowych (ang. *multi-valued logic*), stanowi uogólnienie klasycznej dwuwartościowej logiki. Jest ściśle powiązana z teorią zbiorów rozmytych i teorią prawdopodobieństwa. Została zaproponowana przez Lotfi Zadeha w 1965 roku. W logice rozmytej między stanem 0 (fałsz) a stanem 1 (prawda) rozciąga się szereg wartości pośrednich, które określają stopień przynależności elementu do zbioru.



Logika rozmyta okazała się bardzo przydatna w zastosowaniach inżynierskich, czyli tam, gdzie klasyczna logika klasyfikująca jedynie według kryterium prawda/fałsz nie potrafi skutecznie poradzić sobie z wieloma niejednoznacznościami i sprzecznościami. Znajduje wiele zastosowań, między innymi w [przykład tworzenia listy numerowanej - zastosowanie stylu akapitu 'ud\_ListOrdered']:

1. elektronicznych systemach sterowania (maszynami, pojazdami i automatami),
2. zadaniach eksploracji danych,
3. budowie systemów ekspertowych.

Metody logiki rozmytej wraz z algorytmami ewolucyjnymi i sieciami neuronowymi stanowią nowoczesne narzędzia do budowy inteligentnych systemów mających zdolności uogólniania wiedzy.

# Bibliografia