Programowanie w języku C++.

Wiesław Porębski

SKRÓCONY SPIS TREŚCI:

Przedmowa

C++ należy do grupy obiektowych języków programowania. Korzenie tych języków sięgają końca lat sześćdziesiątych; rok 1967 uważa się za datę powstania pierwszego języka obiektowego, Simula-67. Język Simula-67 bazował na wcześniejszej pracy nad specjalizowanym językiem Simula-1, przeznaczonym do symulacji zdarzeń dyskretnych, zachodzących w reaktorach jądrowych. Twórcami Simuli byli dwaj badacze norwescy, Kristen Nygaard i Ole-Johan Dahl, pracujący w Norweskim Centrum Obliczeniowym. W języku tym wprowadzono szereg pojęć i koncepcji, które z niewielkimi zmianami przejęły współczesne języki obiektowe. W szczególności, obok typów wbudowanych (integer, real, Boolean), podobnych do stosowanych w najbardziej wówczas znanym języku proceduralnym ALGOL, wprowadzono w Simuli pojęcie klasy. Wprowadzono również mechanizm dziedziczenia, który przekazuje klasie potomnej cechy klasy rodzicielskiej.
Kolejnym znaczącym krokiem w rozwoju języków obiektowych było opracowanie przez Alana Kay z Xerox PARC języka Smalltalk. Język ten powstał w latach 1970-1972 i posłużył jako pierwowzór dla współczesnej jego wersji, opracowanej przez Adele Goldberg w roku 1983; wersja ta znana jest obecnie pod nazwą Smalltalk-80.
Okresem szczególnej aktywności w badaniach nad językami programowania były lata siedemdziesiąte. Powstało wtedy prawdopodobnie kilka tysięcy różnych języków i ich dialektów, ale przetrwało w szerszym obiegu zaledwie kilka. Tak więc na rynku utrzymały się języki: Smalltalk, Ada (sukcesor języków ALGOL 68 i Pascal, z pewnym wkładem od języków Simula, Alphard i CLU), C++ (pochodzący z mariażu języków C i Simula), Eiffel (oryginalne dzieło Bertranda Meyera, bazujący po części na Simuli) oraz obiektowe wersje języków Pascal (Object Pascal, Turbo Pascal) i Modula-2 (Modula-3).
Powstało również szereg języków obiektowych dla konstruowania systemów ekspertowych oraz tzw. sztucznej inteligencji. Wśród nich także nastąpiła ostra selekcja i powszechnie stosowanych języków pozostało niewiele; można tu wymienić dwa szerzej znane: CLOS (akronim od Common Lisp Object System) oraz CLIPS (akronim od C Language Integrated Production System), który po rozszerzeniu o mechanizm dziedziczenia znany jest obecnie pod nazwą COOL (akronim od. CLIPS Object-Oriented Language).
Spróbujmy teraz określić, jakie wspólne własności mają wszystkie wymienione języki. Obecnie uważa się, że język programowania można uznać za obiektowy, jeżeli spełnia następujące wymagania:

- Pozwala definiować *klasy* i ich wystąpienia, nazywane *obiektami*. Definicja klasy w języku obiektowym jest podobna do definicji abstrakcyjnego typu danych w językach proceduralnych (typu, który nie jest wbudowany, lecz może być zdefiniowany przez programistę). W językach Eiffel, Smalltalk i Simula klasa jest niepodzielną jednostką syntaktyczną, zawierającą definicje struktur danych i definicje operacji wykonywanych na tych strukturach. Natomiast w językach hybrydowych, jak np. Object Pascal, Turbo Pascal, CLOS i C++ klasa jest traktowana jako "deklaracja typu" umieszczana w jednym pliku, a definicje operacji (nazywane metodami, funkcjami składowymi, lub funkcjami pierwotnymi) umieszcza się zwykle w innym pliku. Konkretna operacja może być implementowana za pomocą jednej tylko metody lub wielu metod. W pierwszym przypadku nazywamy ją *monomorficzną*, zaś w drugim Ä *polimorficzną* albo *wirtualną*. Wystąpienia obiektów danej klasy są tworzone według tego samego wzorca, zawartego w definicji klasy. W rezultacie wszystkie obiekty danej klasy mają takie same struktury danych (atrybuty) i operacje. Tym niemniej każdy obiekt ma własną "tożsamość", a więc, po utworzeniu, istnieje niezależnie od innych obiektów tej samej klasy.

- Zapewnia *ukrywanie informacji* (hermetyzację, ang. encapsulation), co można rozumieć jako zamknięcie obiektu w swego rodzaju "czarnej skrzynce" lub "kapsułce". W większości języków obiektowych hermetyzacja nie zawsze jest pełna, ponieważ zawierają one mechanizmy kontroli dostępu do elementów klasy. Jako zasadę przyjmuje się ukrywanie przed użytkownikiem definicji struktur danych i definicji operacji, przy czym same operacje (lub tylko część z nich) są publicznie

dostępne. Zbiór publicznie dostępnych operacji dla obiektu danej klasy nazywa się często *publicznym interfejsem* obiektu. Taka konstrukcja klas jest logiczna, ponieważ dla użytkownika klasy istotne jest tylko to, jak się nazywa dana operacja, do czego służy i jak się ją uaktywnia (wywołuje). Hermetyzację realizuje się zwykle w ten sposób, że użytkownik nie ma dostępu do kodu źródłowego z definicjami klas i operacji, a jedynie do publicznych interfejsów. Dzięki temu zapewnia się ochronę klas (przede wszystkim predefiniowanych klas bibliotecznych) przed nieuprawnionym dostępem.

- Posiada wbudowany *mechanizm dziedziczenia*, dzięki któremu można tworzyć klasy potomne (podklasy, klasy pochodne) od jednej lub kilku klas rodzicielskich, nazywanych superklasami lub klasami bazowymi. Klasy potomne mogą z kolei być klasami rodzicielskimi dla swoich klas pochodnych, co pozwala tworzyć "drzewa" (hierarchie) klas. W praktyce klasa bazowa jest na ogół prostą konstrukcją językową, zaś klasa pochodna jest jej specjalizacją, co zwykle prowadzi do rozszerzenia definicji klasy bazowej o nowe elementy. Elementami tymi mogą być nowe struktury danych i nowe metody, bądź też operacje o tych samych nazwach co w klasie bazowej, ale o innych definicjach.
  Mechanizm dziedziczenia jest niezwykle efektywny: nie wymaga kopiowania kodu źródłowego klasy bazowej, ponieważ klasa pochodna automatycznie dziedziczy wszystkie lub wybrane cechy klasy bazowej. W rezultacie mamy lepszą, bardziej przejrzystą organizację programu.

- Dysponuje cechami polimorfizmu. *Polimorfizm* jest terminem zapożyczonym z biologii i oznacza dosłownie wielopostaciowość. W odniesieniu do programów obiektowych polimorfizm można określić krótko: jeden interfejs (operacja), wiele metod. Najprostszą postacią (wbudowany polimorfizm ad hoc) polimorfizmu jest wykorzystanie tego samego symbolu dla semantycznie nie związanych operacji. Polimorfizm tego rodzaju jest charakterystyczny dla większości współczesnych języków programowania wysokiego poziomu, nie tylko obiektowych. Przykładem może być używanie tych samych symboli operacji arytmetycznych, np. symbolu mnożenia "*", przy mnożeniu liczb całkowitych i rzeczywistych, chociaż w każdym konkretnym przypadku będzie wywoływana inna metoda mnożenia. Realizacja takiego wywołania wymaga wyznaczenia adresu danej metody; jeżeli adresy odpowiednich metod są przekazywane w fazie kompilacji, to proces ten nazywany jest *wiązaniem wczesnym* lub statycznym. W językach obiektowych mechanizm wiązania wczesnego wykorzystuje się również przy *przeciążaniu operatorów*, tj. nadawaniu innego znaczenia operatorom wbudowanym w język oraz (co jest specyfiką języka C++) przy przeciążaniu funkcji. Jednak o sile języka obiektowego decyduje możliwość wykorzystania *wiązania późnego* (dynamicznego), gdy adres wywoływanej metody staje się znany dopiero w fazie wykonania programu. Wiązanie późne występuje dla hierarchii klas zaprojektowanej w taki sposób, że zdefiniowane w pierwotnej klasie bazowej ("korzeniu" drzewa klas) metody są redefiniowane w klasach pochodnych z zachowaniem tej samej nazwy, typu, liczby i typów argumentów. Tego rodzaju metody nazywa się *wirtualnymi*. Zauważmy, że konstrukcja taka nie pozwala na związanie wywołania operacji z jej metodą w fazie kompilacji, ponieważ postać wywołania jest dokładnie taka sama dla każdej operacji w całej hierarchii klas. Dopiero w fazie wykonania, gdy zostanie utworzony obiekt odpowiedniej klasy, można wywołać metodę wirtualną dla tego obiektu.

Język C++ zawiera wszystkie wymienione cechy, a ponadto takie, które czynią go bardzo efektywnym; dzięki temu staje się de facto standardem przemysłowym. De facto, ponieważ dotychczasowe prace komitetów normalizacyjnych ANSI X3J16 oraz ISO WG-21 doprowadziły jedynie do opublikowania w roku 1994 zarysu standardu. Tekstem źródłowym dla obu komitetów była wykładnia języka, opublikowana w książce Margaret Ellis i twórcy C++, Bjarne Stroustrupa, *The Annotated C++ Reference Manual* [2].
Pierwotnym zamysłem Stroustrupa było wyposażenie bardzo efektywnego języka C w klasy, wzorowane na klasach Simuli. Zrealizował go w roku 1980, konstruując preprocesor rozszerzonego w ten sposób języka C. Nowy język, nazwany "C with classes", był wtedy traktowany raczej jako dialekt języka C, chociaż zawierał już większość cech języka C++ (dziedziczenie, kontrola dostępu, konstruktory i destruktory, klasy zaprzyjaźnione, silna typizacja). W latach 80-tych Stroustrup wraz z grupą współpracowników wprowadzał dalsze mechanizmy i konstrukcje językowe (funkcje rozwijalne,

argumenty domyślne, przeciążanie operatorów i funkcji, referencje, stałe symboliczne, zarządzanie pamięcią, dziedziczenie mnogie, funkcje wirtualne, szablony klas i funkcji, obsługa wyjątków), co wraz ze ściślejszą kontrolą typów doprowadziło język C++ do obecnego kształtu.
C++ jest językiem wysoce modularnym; każdy program można zdekomponować na oddzielnie kompilowane moduły z publicznym interfejsem i ukrytą implementacją. I odwrotnie: do każdego programu można dołączać wcześniej opracowane moduły, przechowywane na dysku w postaci tzw. plików nagłówkowych.
Kluczowym pojęciem w C++ jest *klasa*, traktowana jako typ definiowany przez użytkownika. W definicji języka nie przewidziano standardowej biblioteki klas jako części środowiska programowego, chociaż opracowanie [4], sygnowane przez AT&T, zawiera opis i sposób korzystania z bibliotek wejścia/wyjścia. W praktyce można więc spotkać wiele bibliotek klas, opracowanych niezależnie od standardu AT&T, jak np.

– bibliotekę NIH (USA, National Institutes of Health), wzorowaną na bibliotece języka Smalltalk;
– bibliotekę Interviews, która pozwala na dogodne używanie systemu X Window z poziomu C++;
– bibliotekę GNU C++ (g++), opracowaną w ramach projektu GNU;
– biblioteki dla tworzenia obiektów trwałych (POET, ObjectStore, ONTOS, Versant).
– biblioteki specjalizowane, jak np. RHALE++ (dla obliczeń matematycznych w fizyce), SIMLIB (dla symulacji sieci przełączanych).

Z bieżących informacji wynika, że komitety ANSI/ISO przyjęły już standardy dla następujących klas bibliotecznych: array (szablon tablic), dynarray (szablon tablic dynamicznych), string (szablon łańcuchów), wstring (szablon łańcuchów z rozszerzonym kodem znaków), bits<N> (szablon zbioru bitowego o ustalonej liczności), bitstring (szablon zbioru o zmiennej liczności) i complex (liczby zespolone). W najbliższym czasie można się spodziewać przyjęcia standardów dla szablonów klas: vector, list i associative array (map).

Ze względu na brak niektórych standardów, biblioteki klas są używane w niniejszej książce raczej oszczędnie; prawie wyłącznie będą to biblioteki wejścia/wyjścia z bardzo nielicznymi odstępstwami. Dzięki temu zamieszczone w tekście przykłady (a jest ich ponad 160) były kompilowane i wykonywane zarówno w środowisku Windows'95 (kompilator Borland C++, wersja 5.01, kompilator Visual C++ v.4.0), jak i w środowisku Unix (kompilatory CC, GNU gcc, GNU g++, v.2.8.1).

Prezentowany tekst należy traktować raczej jako wstęp do programowania w języku C++, a nie jako wyczerpujący podręcznik (zarówno w sensie kompletności wykładu, jak i znużenia potencjalnego czytelnika). W stwierdzeniu tym nie należy upatrywać samokrytyki; w książce o rozsądnej objętości można dokładnie opisać składnię i semantykę języka C++, ale pragmatyka może mieć potencjalnie (i ma) tak wiele kontekstów, że nie jest praktycznie możliwe opisanie wszystkich możliwych wariantów i niuansów.

Chociaż język C++ został "nadbudowany" nad językiem C, zrozumienie prezentowanego tekstu, przykładów i programów, nie wymaga umiejętności programowania w języku C. Tym niemniej założono, że czytelnik ma pewne doświadczenia programistyczne w którymś z języków wysokiego poziomu, jak np. Pascal, Modula-2, czy wreszcie wspomniany język C. Bardzo polecam uważne przestudiowanie zamieszczonych przykładów; ponieważ zdecydowana większość z nich to kompletne programy, warto je skompilować i wykonać w dostępnym środowisku programowym. Sądzę, że towarzyszące przykładom dyskusje i analizy programów okażą się interesujące nie tylko dla początkujących, ale i dla zaawansowanych programistów, których uwadze polecam rozdziały 10 i 11, poświęcone obsłudze wyjątków i dynamicznej kontroli typów. Zawarte w tych rozdziałach opisy i dyskusje oparto na ostatnich ustaleniach wspomnianych komitetów ANSI/ISO, a kompilacja i wykonanie programów wymagają dostępu do najnowszych kompilatorów, np. Borland C++ w wersji 5.01 lub CC w wersji 4.0.

Gdańsk, w sierpniu 1998 W. M. Porębski

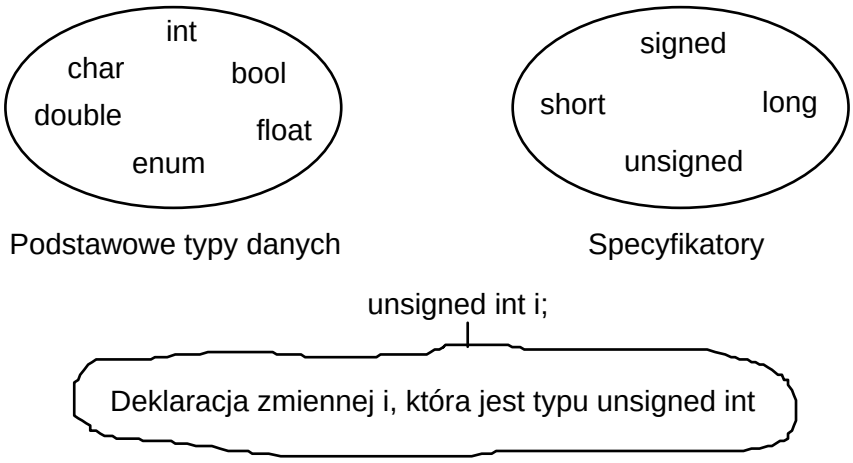
1. Podstawowe elementy języka C++

W języku C++ wykorzystuje się zbiór znaków ASCII, zdefiniowany normą ANSI3.4-1968. Jest to amerykański wariant międzynarodowego, 7-bitowego kodu ISO 646-1983. Elementami tego zbioru są:
małe i duże litery alfabetu łacińskiego,
cyfry od 0 do 9,
znaki przestankowe i inne symbole specjalne.
Każdy znak zbioru ASCII ma swój numer porządkowy (kod znaku); np. kodem znaku 'A' jest liczba 65 w dziesiętnym systemie liczenia, lub 101 w systemie oktalnym (ósemkowym). Znaki kodu ASCII zestawiono w Dodatku A.

Ponieważ C++ jest językiem o silnej typizacji, zatem każda nazwa (identyfikator) w programie musi mieć związany z nią typ, podawany w deklaracji identyfikatora. Typ określa zbiór wartości, jakie można przypisać danej nazwie oraz zbiór operacji, jakie można zastosować do takiej nazwy (tj. do reprezentowanej przez tę nazwę struktury danych), a także sposób interpretacji poszczególnych operacji.

1.1. Wbudowane typy danych

Każda implementacja języka C++ zawiera dwie kategorie typów danych: typy wbudowane i typy definiowane przez użytkownika. Zarówno pierwsze, jak i drugie są abstrakcyjnymi reprezentacjami rzeczywistych struktur danych.
Pośród typów wbudowanych wyróżnia się typy podstawowe: typ char, typ int, typ float, typ double, typ bool i typ enum. Nazwy typów podstawowych można poprzedzać tzw. specyfikatorami typu, tj. słowami kluczowymi short, long, signed i unsigned, które zmieniają wewnętrzną reprezentację danych tych typów.
Każdy typ wbudowany ma także szereg skojarzonych z nim typów pochodnych: wskaźniki, referencje i tablice.
Rysunek 1-1 ilustruje typy wbudowane.



Rysunek 1-1 Podstawowe typy danych C++

Poniżej zestawiono sensowne (i najczęściej używane) kombinacje typów i specyfikatorów.

Dla wielkości (stałych i zmiennych) całkowitych: char, short int, int, long int. Norma ANSI/ISO stanowi, że zapisów “short int” oraz “long int” nie można w programach skracać do “short” i “long”.
Dla wielkości (stałych i zmiennych) zmiennoprzecinkowych: float, double, long double.
Dla wielkości całkowitych bez znaku, wartości logicznych, tablic bitowych: unsigned char, unsigned short int, unsigned int, unsigned long int.

Przy jawnym deklarowaniu wielkości ze znakiem: signed char, signed short int, signed int.

☞ *Uwaga1. Jeżeli na wielkościach typów char oraz int mają być przeprowadzane operacje arytmetyczne lub logiczne, to są one najpierw niejawnie przekształcane do typu int. W analogicznej sytuacji dane typu float są przekształcane do typu double.*

W języku C++ istnieje ponadto typ **void**, który posiada pusty zbiór wartości. Z punktu widzenia składni typ void zachowuje się analogicznie do typów podstawowych (int, char, float, double, enum). Jednakże można go używać jedynie jako fragmentu typu pochodnego, ponieważ nie ma obiektów typu void. Na pierwszy rzut oka może się wydawać dziwnym posiadanie typu, dla którego nie ma zdefiniowanych wartości. Jednak w praktyce typ ten jest bardzo użyteczny, szczególnie w zastosowaniu do funkcji, które nie zwracają żadnej wartości, a więc odpowiadają znanym z innych języków programowania procedurom.
☞ Uwaga 2. Typy char, int (z ewentualnymi specyfikatorami) oraz enum są łącznie nazywane typami całkowitymi. Typy całkowite i zmiennopozycyjne są łącznie nazywane typami arytmetycznymi.

1.2. Jednostki leksykalne

W języku C++ znajdujemy cztery rodzaje jednostek leksykalnych: identyfikatory, słowa kluczowe, literały oraz różne separatory. Spacje, znaki tabulacji poziomej i pionowej, znaki nowego wiersza i nowej strony oraz komentarze (nazywane łącznie "białymi znakami") są w ogólności ignorowane, chyba że służą do separacji jednostek leksykalnych. W procesie kompilacji programu jednostki leksykalne są wyodrębniane (ang. parsing) z wejściowego strumienia znaków w ten sposób, że jako następną jednostkę bierze się najdłuższy ciąg znaków po białym znaku lub po sekwencji takich znaków.

1.2.1. Identyfikatory

Nazwa (identyfikator) jest sekwencją liter i cyfr o dowolnej długości. Pierwszy znak musi być literą, przy czym znak podkreślenia '_' jest traktowany jako litera. Rozróżniane są litery małe i duże. Należy unikać stosowania nazw, zaczynających się od znaku podkreślenia lub dwóch kolejnych znaków podkreślenia, ponieważ nazwy takie są zarezerwowane dla predefiniowanych identyfikatorów, m. in. dla bibliotek.

1.2.2. Komentarze

Komentarze są, w pewnym sensie, wizytówką programisty, ponieważ ich zadaniem jest zwiększenie czytelności programu. Krótkie komentarze mają postać dowolnego ciągu znaków, zapisywanych w jednym wierszu po symbolu komentarza '//'. Taki komentarz kończy się wraz z końcem danego wiersza. Dłuższe komentarze zaleca się umieszczać pomiędzy parami znaków '/*' i '*/'. Jeżeli po symbolu '//' lub pomiędzy symbolami '/*' i '*/' wystąpią pary znaków '//', bądź pary znaków '/*' i '*/', to będą one traktowane jak zwykłe znaki (w C++ nie ma komentarzy zagnieżdżonych). Podobnie jak w innych językach programowania, komentarze wpływają tylko na wielkość kodu źródłowego, ponieważ są usuwane z programu przez kompilator przed generacją kodu wynikowego.
Ponieważ komentarze nie są włączane do wynikowego kodu programu, mogą być w nich używane polskie znaki diakrytyczne, np. ą, ę, etc. Jest to pewne ułatwienie dla polskiego programisty, który jest zmuszony korzystać wyłącznie ze 127 znaków ASCII we wszystkich pozostałych elementach programu.

1.2.3. Słowa kluczowe i operatory

Zestawione niżej słowa kluczowe są niepodzielnymi ciągami znaków. Są to zastrzeżone identyfikatory, które można używać jedynie w ściśle zdefiniowanym kontekście.

Asm	do	inline	Short	typeid
Auto	double	int	Signed	typename
Bool	dynamic_cast	long	Sizeof	union
Break	else	mutable	static	unsigned
Case	enum	namespace	static_cast	using
Catch	explicit	new	struct	virtual
Char	extern	operator	switch	void
Class	false	private	template	volatile
Const	float	protected	this	wchar_t
const_cast	for	public	throw	while
Continue	friend	register	true	
Default	goto	reinterpret_cast	try	
Delete	if	return	typedef	

Jeżeli reprezentacja wewnętrzna kodu źródłowego jest zapisywana w kodzie ASCII, to poniższe symbole jednoznakowe są używane jako znaki przestankowe lub operatory:

```
!   %  ^      &      *      (      )      -      +      =      {      }      |      ~

[   ]  \      ;      '      :      "      <      >      ?      ,      .      /
```

Operatorami są również następujące symbole dwu- i trzyznakowe:

```
->  ++ --     .*     ->*    <<     >>     <=     >=     ==     !=     &&
||  *=        /=     %=     +=     -=     <<=    >>=    &=     ^=     |=     ::
```

Ponadto wiersze tekstu programu źródłowego, które mają być przetwarzane przez preprocesor, znakuje się symbolem `'#'` w pierwszej kolumnie nowego wiersza.

1.2.4. Stałe całkowite

Stałe całkowite są przykładami *literałów stałych*; literałów, ponieważ mówimy jedynie o ich wartościach; stałych, ponieważ ich wartości nie można zmieniać. Każdy literał jest pewnego typu; np. 2 jest typu **int**. Stała całkowita jest traktowana jako **całkowita liczba dziesiętna**, jeżeli składa się z ciągu cyfr dziesiętnych. Ciąg cyfr dziesiętnych z zakresu 0-7 jest traktowany jako **oktalna (ósemkowa) liczba całkowita**, jeżeli pierwszym znakiem ciągu jest cyfra 0. **Szesnastkowa (ang. hexadecimal) stała całkowita** jest ciągiem cyfr szesnastkowych (cyfry 0-9 i/lub litery a-f, bądź A-F), poprzedzonych dwuznakowymi symbolami '0x' (zero-x)lub '0X'. Przykłady:

```
89      //liczba dziesiętna
037     //liczba ósemkowa
0x12    //liczba szesnastkowa
0X7F    //liczba szesnastkowa
```

Typ stałej całkowitej przyjmowany domyślnie przez kompilator zależy od jej postaci, wartości i przyrostka. Jeżeli jest to liczba dziesiętna i nie ma przyrostka, to typ domyślny zależy od jej wartości i jest typem **int**, **long int**, lub **unsigned long int**. Jeżeli jest to liczba oktalna lub szesnastkowa i nie ma przyrostka, to typ domyślny zależy od jej wartości i jest typem **int**, **unsigned int**, **long int**, **unsigned**

long int. Dodając specyfikator **u** bądź **U** (dla **unsigned int**), i/lub **l** bądź **L** (dla **long int**) możemy wymusić inny sposób reprezentacji stałej całkowitej, np. 25UL, 127u, 38000L.

1.2.5. Stałe zmiennopozycyjne

Stałe zmiennopozycyjne należą do podzbioru liczb rzeczywistych. Można je zapisywać w notacji dziesiętnej z kropką dziesiętną, np.

0.0 .28 2. -84.17
lub w notacji wykładniczej, np.

1.18e12 -3.1415E-3 3e8

Jeżeli po liczbie nie podano specyfikatora typu, to kompilator nadaje jej typ domyślny **double**. Dokładność reprezentacji stałej zmiennopozycyjnej można wymusić, dodając po zapisie liczby specyfikator **f** lub **F** (dla typu **float**albo **l** lub **L** (dla typu **long double**), np.

-84.17f .28F, 1.0L 3.14159e-3L

1.2.6. Stałe znakowe

Stała (literał) znakowa jest to ciąg, złożony z jednego lub większej liczby znaków, ujęty w pojedyncze apostrofy, np. 'x'. Stałe jednoznakowe są typu **char**. Wartością stałej jednoznakowej jest wartość numeryczna znaku w maszynowym zbiorze znaków (np. dla zbioru znaków ASCII, wartością 'A' jest 65 dziesiętnie lub 101 oktalnie). Typem stałej wieloznakowej jest **int**. Pewne znaki, które nie mają reprezentacji graficznej na ekranie monitora, czy też na papierze drukarki, mogą być reprezentowane w programie przez tzw. *sekwencje ucieczki*, zapisywane ze znakiem **'\'** (ang. **escape sequences**; słowo "ucieczka" mówi o tym, że następny po \ znak "ucieka" od przypisanego mu standardowego znaczenia), jak pokazano w tablicy 1.1.
Wartości znaków podane w tablicy 1.1 są zapisane w systemie oktalnym lub szesnastkowym.

Tablica 1.1 Sekwencje ucieczki dla znaków kodu ASCII

Nazwa sekwencji	Symbol	Zapis znakowy	Wartość liczbowa
nowy wiersz (new-line)	NL (LF)	\n	12
tabulacja pozioma (horizontal tab)	HT	\t	11
tabulacja pionowa (vertical tab)	VT	\v	13
← (backspace)	BS	\b	10
powrót karetki (carriage return)	CR	\r	15
nowa strona (form feed)	FF	\f	14
dzwonek (alert)	BEL	\a	7
\ (backslash)	\	\\	x5c
znak zapytania (question mark)	?	\?	x3f
pojedynczy apostrof (single quote)	'	\'	x27
Podwójny apostrof (double quote)	"	\"	x22
znak zerowy integer()	NUL	\0	0
liczba oktalna (octal number)	ooo	\ooo	ooo
liczba szesnastkowa (hex number)	hhh	\xhh	xhh

1.2.7. Stałe łańcuchowe

Stała (literał) łańcuchowy jest to ciąg o długości zero lub więcej znaków, ujęty w podwójne apostrofy. Jeżeli w ciągu występują znaki niedrukowalne (np. BEL), to są one reprezentowane przez ich sekwencje ucieczki. W reprezentacji wewnętrznej do każdego łańcucha jest dodawany terminalny znak zerowy '\0' o wartości 0; tak więc np. łańcuch "abcd" ma długość 5 (a nie 4) znaków, ponieważ po znaku 'd' kompilator doda znak zerowy '\0'. Jeżeli łańcuch rozciąga się na kilka wierszy, to na końcu każdego wiersza można dodać znak '\', który sygnalizuje kompilatorowi, że stała łańcuchowa jest kontynuowana w następnym wierszu.

2. Struktura i elementy programu

2.1. Deklaracje i definicje

Jak już wspomniano, wszystkie wielkości występujące w programie muszą być przed ich użyciem zadeklarowane. Deklaracje ustalają nieodzowne odwzorowanie pomiędzy strukturami danych i operacjami, a reprezentującymi je konstrukcjami programowymi. Każda deklaracja wiąże podany przez użytkownika identyfikator z odpowiednim typem danych. Większość deklaracji, znanych jako **deklaracje definiujące** lub **definicje**, powoduje także utworzenie definiowanej wielkości, tj. przydzielenie (alokację) jej fizycznej pamięci i ewentualne zainicjowanie. Pozostałe deklaracje, nazywane **deklaracjami referencyjnymi** lub **deklaracjami**, mają znaczenie informacyjne, ponieważ jedynym ich zadaniem jest podanie deklarowanej nazwy i jej typu do wiadomości kompilatorowi. Tak zadeklarowany identyfikator musi być w programie zdefiniowany – albo później w tym samym, albo w oddzielnym pliku z kodem źródłowym. Dla danego identyfikatora może wystąpić wiele deklaracji referencyjnych, ale tylko jedna deklaracja definiująca (przypadki takie występują najczęściej w programach wieloplikowych). Oczywistą jest zasada, że żaden identyfikator nie może być użyty w programie przed jego punktem deklaracyjnym w kodzie źródłowym. Deklaracja zakończona średnikiem nazywa się **instrukcją deklaracji.**

Najczęściej deklarowanymi wielkościami są zmienne. **Zmienną** określa się jako pewien obszar pamięci o zadanej symbolicznej nazwie, w którym można przechowywać wartości, interpretowane zgodnie z zadeklarowanym typem zmiennej. Język C++ uogólnia to pojęcie: wymieniony obszar pamięci może nie posiadać nazwy, może mieć nie jedną lecz kilka nazw, zaś adres początku obszaru pamięci może być dostępny dla programisty. Przykład 2.1 ilustruje różnorodność możliwych deklaracji i definicji.

Przykład 2.1.

```
char znak;
char litera = 'A';
char* nazwa = "Cplusplus";
extern int ii;
int j = 10;
const double pi = 3.1415926;
enum day { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
struct mystruct;
struct complex { float re, im;};
complex zespolone;
typedef complex punkt;
class myclass;
template<class T> abs(T x) { return x < 0 ? -x : x; }
```

Dyskusja. Jak widać z powyższego przykładu, w deklaracji można umieścić o wiele więcej informacji dla kompilatora, niż jedynie wskazać, że dana nazwa jest związana z określonym typem. Tylko 3 spośród nich

```
extern int ii;                    struct mystruct;     class myclass;
```

są deklaracjami referencyjnymi, a zatem muszą im towarzyszyć odpowiednie definicje. Definicje struktury `mystruct` i klasy `myclass` muszą się znaleźć w tym samym pliku, w którym podano ich deklaracje, zaś definicja zmiennej `ii` musi być podana w jednym z plików programu. Pozostałe deklaracje są jednocześnie definicjami:

- Wielkości `znak`, `litera` oraz `j` są zmiennymi typów podstawowych. Stosownie do ich typów kompilator przydzieli im odpowiednie obszary pamięci. Zauważmy także, że zmienne `litera`, `nazwa` oraz `j` zostały zainicjowane odpowiednimi dla ich typów wartościami.
- Wielkość `pi` typu **double** jest **stałą symboliczną**, o czym informuje słowo kluczowe (deklarator stałej) **const** na początku deklaracji. Wartość tej stałej (3.1415926) jest znana kompilatorowi, zatem nie trzeba dla niej rezerwować pamięci.
- Wielkości `day` i `complex` są definicjami nowych typów; identyfikator `zespolone` jest zmienną typu **complex**, zaś identyfikator `punkt` jest zastępczą nazwą (synonimem) dla `complex`.

2.1.1. Deklaracje stałych

Język C++ pozwala definiować szczególnego rodzaju "zmienne", których wartości są ustalone i niezmienne w programie. Jeżeli definicję zmiennej zainicjowanej, np.

```
int cyfra = 7;
```

poprzedzimy słowem kluczowym **const**

```
const int cyfra = 7;
```

to przekształcimy w ten sposób symboliczną zmienną `cyfra` z pierwszej definicji w **stałą symboliczną** o tej samej nazwie. Wartość tak zdefiniowanej stałej symbolicznej pozostaje niezmienna w programie, a każda próba zmiany tej wartości będzie sygnalizowana jako błąd. Słowo kluczowe **const**, które zmienia interpretację definiowanej wielkości, jest nazywane **modyfikatorem typu**. Nazwa uzasadniona jest tym, że **const** ogranicza możliwości użycia definiowanej wielkości tylko do odczytu, ale zachowuje informację o typie (w przykładzie jak wyżej typ stałej `cyfra` został zdefiniowany jako **int**). Dzięki temu stałe symboliczne w języku C++ można używać zamiast literałów tego samego typu.
Zauważmy też, że skoro nie można zmieniać wartości stałej symbolicznej po jej zdefiniowaniu, to musi ona być zainicjowana. Np. zapis
`const double pi;`
jest błędny, ponieważ wielkość `pi` nie została zainicjowana.
Podobnie jak wszystkie obiekty programu, stała symboliczna może mieć tylko jedną definicję. Po zdefiniowaniu jest ona (domyślnie) widoczna tylko w pliku, w którym umieszczono jej definicję. W większych programach, składających się z wielu plików, możemy uczynić ją widoczną dla innych plików, umieszczając w nich deklaracje, informujące kompilator o tym, że w jednym z plików programu znajdzie jej definicję. Informację tę przekazujemy kompilatorowi za pomocą słowa kluczowego **extern**, umieszczonego przed deklaracją. Np. stałą `cyfra` możemy udostępnić innym plikom programu, umieszczając w nich deklaracje o postaci:
`extern const int cyfra;`

☞ *Uwaga. Starsze kompilatory języka C++ "odgadują" typ stałej symbolicznej na podstawie jej wartości numerycznej i formatu definicji. Jednak standard wymaga jawnego podawania typu każdej stałej.*

2.1.2. Wyliczenia

Alternatywnym, a często bardziej przydatnym sposobem definiowania symbolicznych stałych całkowitych jest użycie do tego celu typu wyliczeniowego (ang. enumerated type). Wyliczenie deklaruje się ze słowem kluczowym **enum**, po którym następuje wykaz stałych całkowitych (ang. enumerators) oddzielonych przecinkami i zamkniętych w nawiasy klamrowe. Wymienionym stałym są przypisywane wartości domyślne: pierwszej z nich – wartość 0, a każdej następnej – wartość o 1 większa od poprzedzającej. Np. wyliczenie:

```
enum {mon, tue, wed, thu, fri, sat, sun};
```

definiuje siedem sta³ych ca³kowitych i przypisuje im warto□ci od 0 do 6. £atwo zauwaæyć, æe powyæszy zapis jest krótszy niæ sekwencja deklaracji sta³ych:

```
const int mon = 0;
const int tue = 1;
   .
   .
   .
const int sun = 6;
```

Stałym typu wyliczeniowego można również przypisywać wartości jawnie, przy czym wartości te mogą się powtarzać. Np. deklaracja:

```
enum { false, fail = 0, pass, true = 1 };
```

przypisuje wartość 0 do `false` i `fail` (do `false` domyślnie) oraz wartość 1 do `pass` i `true` (do `pass` domyślnie).
W wyliczeniach można po **enum** umieścić identyfikator, który stanie się od tego momentu nazwą nowego typu. Np.

```
enum days { mon, tue, wed, thu, fri, sat, sun };
```

definiuje nowy typ `days`. Pozwala to od tej chwili deklarować zmienne typu `days`, np.

```
days anyday = wed;
```

W taki sam sposób można wprowadzić zapis, który imituje typ **bool** (jeśli nasz kompilator nie zawiera implementacji tego typu)

```
enum bool {false, true};
bool found, success;
//lub np. bool success = true;
```

Deklaracje zmiennych typu wyliczeniowego można również umieszczać pomiędzy zamykającym nawiasem klamrowym a średnikiem, np.

```
enum bool {false,true} found, success;
```

☞ *Uwaga. Typ wyliczeniowy jest podobny do typów **char** oraz **shortint** w tym, że nie można na jego wartościach wykonywać żadnych operacji arytmetycznych. Gdy wartość typu wyliczeniowego pojawia się w wyrażeniach arytmetycznych, to jest niejawnie przekształcana do typu **int** przed wykonaniem operacji.*

2.2. Dyrektywy preprocesora

Kompilacja programu źródłowego, napisanego w języku C++, przebiega zwykle w czterech lub pięciu kolejnych krokach; produktem końcowym jest kod ładowalny. W pierwszym kroku uruchamiany jest program, nazywany **preprocesorem**. Preprocesor przetwarza te wiersze tekstu programu, które zaczynają się znakiem # w pierwszej kolumnie. W wierszach tych zapisujemy polecenia, nazywane **dyrektywami**. W odróżnieniu od instrukcji, które są wykonywane po zakończeniu kompilacji i uruchomieniu programu, dyrektywy są poleceniami do natychmiastowego wykonania – przed rozpoczęciem właściwego procesu kompilacji. Ponieważ dyrektywy przywołują predefiniowane wielkości biblioteczne, z reguły umieszcza się je w pierwszych wierszach tekstu programu.

2.2.1. Dyrektywa #include

Dyrektywa `#include` pozwala zebrać razem wszystkie fragmenty programu źródłowego w jeden **moduł**, nazywany **plikiem źródłowym** lub **jednostką translacji**. Jeżeli dyrektywa ta występuje w postaci:

```
#include <nazwa-pliku.h>
```

to *nazwa-pliku.h* odnosi się do standardowego, predefiniowanego pliku nagłówkowego (bibliotecznego), umieszczonego w standardowym katalogu plików dołączanych (ang. standard include directory). Polecenie o takiej postaci zleca preprocesorowi poszukiwanie pliku tylko w standardowych miejscach bez przeszukiwania katalogu zawierającego plik źródłowy.

Jeżeli dyrektywa `#include` występuje w postaci:

```
#include "nazwa-pliku.h"
```

to preprocesor zakłada, że plik o nazwie *nazwa-pliku.h* został założony przez użytkownika. W takim przypadku poszukiwanie pliku zaczyna się od katalogu bieżącego; jeżeli wynik poszukiwania jest niepomyślny, to jest ono kontynuowane w katalogu standardowym.

☞ *Uwaga 1. W implementacjach Unix-owych standardowym katalogiem dla plików nagłówkowych jest często /usr/include/CC. Jeżeli jest inaczej, to można wymusić początkową ścieżkę poszukiwania, dodając opcję -I w wierszu rozkazowym, np.*

$ CC -I incl -I/usr/local/include myprog.cc.

W implementacji C++ firmy Borland pod systemem MS-DOS katalogiem standardowym będzie katalog c:\borlandc\include

☞ *Uwaga 2. Zarówno dla standardowych, jak i przygotowanych przez użytkownika plików nagłówkowych można w programie podać jawnie pełną ścieżkę do pliku, np.*

#include</usr/local/include/CC/iostream.h>

czy też < c:\borlandc\include\iostream.h >

2.2.2. Dyrektywa #define

Najprostsza postać dyrektywy `#define` ma składnię:

```
#define nazwa
```

co czytamy: "zdefiniuj identyfikator `nazwa`". Dyrektywę `#define` używa się najczęściej w kontekście z dyrektywą `#include`, dyrektywami `#if – #elif – #else – #endif` oraz `#ifdef – #else – #endif` i `#ifndef – #endif`. Konteksty te stosuje się głównie w celu uniknięcia wielokrotnego definiowania tego samego identyfikatora i/lub wielokrotnego wstawiania tego samego pliku bibliotecznego do pliku źródłowego.

Dyrektywę `#define` można również zapisać w postaci:

```
#define nazwa sekwencja-znaków
```

co czytamy: "zastępuj wszystkie wystąpienia identyfikatora `nazwa` podaną sekwencją znaków".

Przykład 2.2.

```
#if              SYSTEM == SYSV
#define  HDR "sysv.h"
#elif      SYSTEM == BSD
#define  HDR "bsd.h"
#elif      SYSTEM == MSDOS
#define  HDR == "msdos.h"
#else
#define  HDR "default.h"
#endif
#include   HDR
```

Dyskusja. Pokazana wyżej sekwencja dyrektyw testuje nazwę `SYSTEM` dla podjęcia decyzji, którą wersję pliku nagłówkowego należy włączyć do programu. Dyrektywa `#if` sprawdza wartość wyrażenia: jeżeli wartość ta jest różna od zera (prawda), to przetwarzana jest następująca po niej dyrektywa `#define` i dyrektywa `#include`; jeżeli nie – to sprawdzane są kolejne alternatywy (`elif` odpowiada `else if`), po czym przetwarzana jest dyrektywa `#include`.

Przy włączaniu do programu standardowych identyfikatorów i standardowych plików bibliotecznych, użytkownik może czuć się zwolniony od takiego sprawdzania, ponieważ obowiązek ten spoczywa na twórcach kompilatorów. Niżej pokazano tego rodzaju testy, zastosowane w plikach nagłówkowych iostream.h dla dwóch kompilatorów.

Przykład 2.3.

```
//Borland C++, plik iostream.h
#ifndef __IOSTREAM_H
#define __IOSTREAM_H
#if !defined(__DEFS_H)
#include <_defs.h>
#endif
// definicje z plików _defs.h i iostream.h
#endif

//Borland C++, plik ver.h
...
typedef       int     BOOL;
#define         TRUE        1
#define         FALSE 0
```

Dyskusja. Dyrektywa `#ifndef` daje wartość `TRUE` (1) dla warunku "nie zdefiniowany". Zatem `#ifndef identyfikator` daje taki sam efekt, jak `#if 0` jeżeli identyfikator został już zdefiniowany i taki sam efekt, jak `#if 1` jeżeli identyfikator nie jest jeszcze zdefiniowany. Zatem `#ifndef __IOSTREAM_H` sprawdza, czy nazwa `__IOSTREAM_H` została już zdefiniowana. Jeżeli nie, to przetwarzana jest dyrektywa `#define __IOSTREAM_H` (definiująca `__IOSTREAM_H`) i następująca po niej dyrektywa `#if`, która sprawdza, czy jest zdefiniowana nazwa `__DEFS_H` i w zależności od wyniku testu włącza lub nie plik `_defs.h` i pozostałą zawartość pliku `iostream.h`. Jeżeli nazwa `__IOSTREAM_H` jest już zdefiniowana, to preprocesor pominie dyrektywę `#define` i następujące po niej dyrektywy, dzięki czemu zawartość pliku `iostream.h` nie zostanie wstawiona po raz drugi.

Przykład 2.4.

```
//plik iostream.h dla kompilatora CC Sun Microsystems
#ifndef IOSTREAMH
#define IOSTREAMH
//definicje z pliku iostream.h
...
#ifndef NULL
#define NULL 0
#endif
...
#ifdef EOF

#if EOF ! = -1
#define EOF (-1)
#endif

#else
#define EOF (-1)
#endif
...
#endif
```

Dyrektywa `#define` bywa niekiedy stosowana dla definiowania stałych. Zapisuje się ją wtedy w drugiej z podanych postaci, tj.

```
#define nazwa sekwencja-znaków
```

np.

```
#define           WIERSZ    80
```

Taka postać dyrektywy zleca preprocesorowi zastępowanie dalszych wystąpień identyfikatora podanym ciągiem znaków.
Zapisaną wyżej dyrektywę można zastąpić definicją stałej symbolicznej

```
const        int WIERSZ = 80;
```

i używać nazwy `WIERSZ` w programie w taki sam sposób.

Definiowanie stałych za pomocą `#define` jest mniej korzystne niż za pomocą modyfikatora const z następujących powodów:
- Zastąpienie identyfikatora ciągiem znaków nie wnosi żadnej dodatkowej informacji, ponieważ tak zdefiniowanej stałej nie przypisuje się żadnego typu.
- Stała symboliczna definiowana z **const** niesie informację o jej typie i może być używana przez program uruchomieniowy (ang. symbolic debugger).

2.3. Struktura prostych programów

Program w języku C++, niezależnie od jego rozmiaru, jest zbudowany z jednej lub kilku "funkcji", opisujących żądane operacje procesu obliczeniowego. W tym sensie funkcje są podobne do podprogramów, znanych w innych językach programowania, a umożliwiających strukturalizację i modularyzację programów. Każdy program musi zawierać funkcję o zastrzeżonej nazwie *main*. Jest to funkcja, od której rozpoczyna się działanie programu. Funkcja main zawiera zwykle wywołania różnych funkcji, przy czym niektóre z nich są definiowane w tym samym programie, zaś inne pochodzą z bibliotek uprzednio napisanych funkcji. W rezultacie program jest zbiorem odrębnych definicji i

deklaracji funkcji. Komunikacja pomiędzy funkcjami odbywa się za pośrednictwem argumentów i wartości zwracanych przez funkcje, bądź (rzadziej) przez zmienne zewnętrzne, których zakres obejmuje jeden plik źródłowy programu. Pokazany niżej przykład wprowadzający ilustruje strukturę prostego programu.

Przykład 2.5.

```
// Struktura programu w jezyku C++
#include <iostream.h>
void czytaj();
void sortuj();
void pisz();

int   main() {
      czytaj() ;
      sortuj() ;
      pisz() ;
      return 0;
}

void czytaj()     { cout << "czytaj()\n";     }
void sortuj()     { cout << "sortuj()\n";     }
void pisz()    { cout << "pisz()\n";   }
```

Po wykonaniu programu na ekranie monitora pojawią się trzy wiersze tekstu:

czytaj()
sortuj()
pisz()

Dyskusja. Przykład prezentuje kilka charakterystycznych cech języka C++. Pierwszy wiersz:

```
// Struktura programu w jezyku C++
```

jest komentarzem. Drugi wiersz:

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

jest instrukcją sterującą, która zleca kompilatorowi włączenie do programu zawartości pliku nagłówkowego o nazwie iostream.h. Plik **iostream.h** jest *standardowym* (predefiniowanym) plikiem nagłówkowym; jest on wyszukiwany przez kompilator w standardowym katalogu (bibliotece) bez przeszukiwania katalogu zawierającego plik źródłowy. W pliku tym jest zdefiniowany symbol **cout**, który reprezentuje tzw. *strumień wyjściowy*. Jak widać z przykładu, strumień wyjściowy służy do wyprowadzenia tekstu na ekran monitora. Wstawienie tekstu do strumienia wyjściowego jest wykonywane przez *operator wstawiania* (ang. insertion operator), oznaczony symbolem '<<'.
Symbol ten jest bardzo trafnie wybrany, ponieważ sugeruje on kierunek przepływu danych: z postaci wyrażenia

```
cout << "czytaj()\n"
```

możemy łatwo się domyślić, że chodzi o wstawienie do strumienia wyjściowego `cout` ujętego w podwójne apostrofy tekstu "czytaj()\n".

Każda funkcja programu składa się z czterech części: typu zwracanego (tutaj void oraz int), nazwy funkcji, wykazu (listy) argumentów i ciała funkcji. Trzy pierwsze części są łącznie nazywane *prototypem funkcji*. Zakończone średnikami zapisy:

```
      void czytaj();     void sortuj();     void pisz();
```

są *deklaracjami funkcji* (ściślej – instrukcjami deklarującymi). Instrukcja deklaracji jest jedyną instrukcją, którą można zapisać na zewnątrz funkcji (w tym przypadku na zewnątrz funkcji main). W omawianym programie deklaracje funkcji zawierają ich prototypy: typem zwracanym jest wbudowany typ prosty **void**, nazwami funkcji są identyfikatory `czytaj`, `sortuj` i `pisz`, zaś wykazy argumentów, ujęte w nawiasy okrągłe, są puste. Deklarowanie funkcji przed ich wywołaniem jest w języku C++ obowiązkowe z oczywistych względów. Natomiast ich definicje można umieszczać w dowolnym miejscu programu. W rozważanym przypadku definicje te umieszczono za funkcją `main`.
Jak widać z przykładu, *definicja funkcji* składa się z typu zwracanego, nazwy funkcji, listy argumentów (formalnych) oraz ciała funkcji, objętego parą nawiasów klamrowych "{}". Ciało funkcji może zawierać instrukcje, tj. polecenia do wykonania przez dany program. *Instrukcja wywołania funkcji*, np. czytaj() składa się z nazwy funkcji i ujętej w nawiasy okrągłe *listy argumentów aktualnych* (w naszym przypadku lista ta jest pusta). Para nawiasów okrągłych "()" jest w tym kontekście nazywana *operatorem wywołania*. Nazwa ta jest uzasadniona tym, że wartościowanie funkcji polega na zastosowaniu operatora "()" do nazwy funkcji. Jeżeli funkcja ma niepustą listę argumentów, to argumenty aktualne są umieszczane wewnątrz operatora wywołania. Operację tę nazywa się *przekazywaniem argumentów* do funkcji. Funkcja main zawiera cztery instrukcje (zauważmy, że każda z nich kończy się średnikiem). Sekwencję instrukcji, ujętą w parę nawiasów klamrowych, nazywa się *instrukcją złożoną*. Instrukcja złożona jest traktowana jako pojedyncza jednostka i może się pojawić wszędzie tam, gdzie ze względów syntaktycznych powinna się znaleźć pojedyncza instrukcja. Zauważmy, że instrukcja złożona nie kończy się średnikiem; jej terminatorem jest zamykający nawias klamrowy '**}**'. Instrukcje złożone mogą być zagnieżdżane. Instrukcja złożona może także zawierać deklaracje; w takim przypadku nazywa się ją *blokiem*. Najprostszy program w języku C++ może wyglądać następująco:

```
void main() {}
```

Definiuje się w nim funkcję typu **void**, o nazwie `main()`, która nie przyjmuje argumentów i nic nie robi.

Wykonanie funkcji `main` kończy się instrukcją `return 0;` jest to wymagany przez język C++ sposób opuszczenia bloku `main`. W ogólności instrukcja `return` może wystąpić w dwóch postaciach:

```
    return;
oraz return wyrażenie;
```

Pierwszą z nich można stosować w przypadku funkcji typu **void** jako opcję.
Zauważmy, że w naszym przykładzie funkcja `main()` jest typu **int**. W większości kompilatorów przy deklarowaniu i definiowaniu funkcji, słowo kluczowe **int** można pominąć. W takim przypadku kompilator przyjmuje **int** jako domyślny typ zwracany.

Następne dwa przykłady ilustrują użycie w programach literałów znakowych i łańcuchowych.

Przykład 2.6.

```
//Komentarze, literaly znakowe i tekstowe
#include <iostream.h> //Dyrektywa preprocesora
int   main() {
      char c;//deklaracja zmiennej c typu char
      c  = '\101';
//101 jest kodem oktalnym znaku 'A' (ASCII)
      char c1 = '\11' ;
/*11 jest kodem oktalnym znaku tabulacji poziomej (ASCII) */
      char c2='\x42';
//42 jest kodem heksalnym znaku 'B' (ASCII)
      char c3 = '\'';
```

```
//Nadanie zmiennej c3 wartosci ' (apostrof)
      cout << c << c1 << c2 << '\n';
//Wydruk wartosci zmiennych c,c1,c2
      cout << "abcde12345" << '\n';
//Wydruk lancucha znakow "abcde12345"
      cout << c3 << endl;
//Wydruk wartosci zmiennej c3
      cout << int(c2); //Wydruk kodu ASCII znaku 'B'
      return 0;
}
```

Wydruk ma postać:

A B
abcde12345
'
66

Dyskusja. Zwróćmy uwagę na następujące elementy programu:

- Krótkie komentarze po średnikach, które kończą zapisy instrukcji.
- Deklaracje zmiennych (np. char c).
- Inicjowanie zmiennych podczas ich deklaracji (np. char c1 = '\11').
- Kilkakrotne użycie operatora wstawiania **<<** tekstu do strumienia wyjściowego.
- Oddzielne użycie znaku nowego wiersza **'\n'** po operatorze **<<**.
- Użycie tzw. manipulatora **endl** zamiast znaku **'\n'**. Różnica pomiędzy nimi jest taka, że **'\n'** jedynie kieruje wyjście do następnego wiersza, zaś **endl** wykonuje tę samą czynność oraz opróżnia bufor wyjściowy.
- Zastosowanie jawnej konwersji (ang. cast) typów w wyrażeniu int(c2) dla wyprowadzenia kodu ASCII znaku 'B', przypisanego do zmiennej c2 w postaci literału '\x42'.

Przykład 2.7.

```
//Literaly tekstowe, operator sizeof
#include <iostream.h>

int   main()  {
      cout << "abcd\n";
      cout << "Liczba znakow w \"abcd\" = " << sizeof "abcd";
      cout << "" << endl; //Lancuch pusty
      cout << "Wyrazy przedzielone\t tabulatorem" << "\n";
      cout << "Tekst dwuwierszowy \
pojawi sie w jednym wierszu";
      return 0;
}
```

Postać wydruku:

abcd
Liczba znakow w "abcd" = 5
Wyrazy przedzielone tabulatorem
Tekst dwuwierszowy pojawi sie w jednym wierszu

Dyskusja. Nowe elementy w powyższym przykładzie są następujące:

– Użycie operatora `sizeof`, który zwraca liczbę bajtów łańcucha "abcd", włącznie z terminalnym znakiem zerowym `\0`.
– Użycie "sekwencji ucieczki" dla podwójnego apostrofu i znaku tabulacji.
– Użycie znaków spacji oraz znaku kontynuacji w stałych łańcuchowych.

☞ *Uwaga. Znak nowego wiersza może być reprezentowany bądź jako stała znakowa '**\n**' bądź jako jednoznakowy łańcuch "**\n**".*

2.4. Zasięg i czas życia obiektów programu

Obiekty C++ (niekoniecznie w sensie używanym w programowaniu obiektowym, ponieważ C++ jest językiem hybrydowym) mogą być alokowane w pamięci statycznej (obiekty globalne), na stosie podprogramu (obiekty lokalne) i w pamięci swobodnej (obiekty dynamiczne). Obiekty globalne istnieją przez cały czas wykonania programu; obiekty lokalne, powoływane do życia np. w bloku funkcji, istnieją tylko do momentu wyjścia z bloku; obiekty dynamiczne istnieją od momentu powołania ich do życia za pomocą specjalnego operatora (**new**) do chwili ich zniszczenia za pomocą specjalnego operatora (**delete**).
Identyfikatory obiektów w programie muszą być unikatowe. Nie znaczy to jednak, że dana nazwa może być użyta w programie tylko jeden raz: można ją użyć ponownie, pod warunkiem istnienia pewnego kontekstu, który pozwala rozróżnić poszczególne wystąpienia. Jednym z takich kontekstów jest *sygnatura funkcji*, tj. wykaz jej argumentów (oraz ich typów), oddzielonych przecinkami. Pozwala to rozróżniać funkcje *przeciążone*, tj. dwie funkcje o takiej samej nazwie, ale o różnych sygnaturach.
Drugim, bardziej ogólnym kontekstem jest zasięg (ang. scope). Język C++ wspiera trzy rodzaje zasięgu: zasięg pliku, zasięg lokalny i zasięg klasy. Każda zmienna ma skojarzony z nią zasięg, który wraz z jej nazwą jednoznacznie ją identyfikuje. Zmienna jest widzialna dla pozostałej części programu jedynie w obrębie swojego zasięgu.
Identyfikatory o zasięgu pliku, nazywane także *globalnymi*, sa deklarowane na zewnątrz wszystkich bloków i klas i automatycznie inicjowane wartością zero. Ich zasięg rozciąga się od punktu deklaracji do końca pliku źródłowego. Tak więc identyfikator zadeklarowany np. przed funkcją *main* może być wykorzystany w jej bloku, zaś deklaracja po bloku funkcji *main* czyni go niewidocznym w tym bloku. Można go jednak uczynić widocznym przez umieszczenie w bloku funkcji dodatkowej deklaracji, poprzedzonej słowem kluczowym **extern**, np.

```
int i; // definicja zmiennej globalnej
int   main()  {
      double d; // definicja zmiennej lokalnej
      ...
      extern int z; // deklaracja referencyjna
      z = 17;
      ...
}
      int z; // definicja zmiennej globalnej
```

Jeżeli definicja identyfikatora globalnego, np. `int ii;`
jest zawarta w jednym pliku, a chcemy go używać również w innych plikach, to w plikach tych umieszczamy jego deklarację, również poprzedzoną słowem kluczowym **extern**:
`extern int ii;`
Deklaracja poprzedzona słowem kluczowym **extern** nie jest definicją – nie powoduje alokacji pamięci. Jest to jedynie informacja dla kompilatora, że gdzieś w programie istnieje definicja `int ii`. Tak więc identyfikator globalny może być widoczny dla całego programu, na który składa się więcej niż jeden plik.
Z powyższego wynika, że obiekty globalne istnieją także przez cały czas wykonania programu, składającego się z wielu plików.
Jeżeli identyfikator globalny poprzedzimy słowem kluczowym **static**, wtedy staje się on niewidzialny na zewnątrz pliku, w którym został zdefiniowany; np.

```
      static int z;
```

pozwala używać zmiennej `z` w innym pliku tego samego programu w innym znaczeniu bez obawy wystąpienia kolizji nazw.
Identyfikatorom statycznym przydzielana jest pamięć od momentu rozpoczęcia wykonania programu do chwili jego zakończenia. Są one inicjowane wartością zero (lub NULL dla wskaźników) przy braku jawnego inicjatora. Z punktu widzenia czasu życia wszystkie zmienne (obiekty) o zasięgu pliku można uważać za statyczne. Podobnie wszystkie funkcje zachowują się jak obiekty statyczne.
Identyfikatory o zasięgu lokalnym (np. o zasięgu bloku lub funkcji) stają się widzialne od punktu ich deklaracji, a przestają być widzialne wraz z końcem bloku lub funkcji zawierającego ich deklaracje (tj. po zamykającym nawiasie klamrowym **}**) . Np. zmienna lokalna zdefiniowana w bloku funkcji jest dostępna jedynie w obrębie tej funkcji; dzięki temu jej nazwa może być ponownie użyta w innych fragmentach programu o innym zasięgu bez obawy konfliktu.
Ponieważ bloki mogą być zagnieżdżone, zatem każdy blok, który zawiera instrukcję deklaracji, utrzymuje swój własny zasięg, np. deklaracje

```
{ ... int ii = 1; ... { ... int ii = 2; ...} }
```

definiują dwie zmienne lokalne `ii` o rozdzielnych zasięgach, a więc dwie *różne* zmienne.
Obiekty lokalne powinny być jawnie inicjowane – w przeciwnym razie ich zawartość jest nieokreślona. Obiekty takie, z punktu widzenia czasu ich trwania, możemy uczynić statycznymi. Jeżeli np. w bloku funkcji zapiszemy deklarację:

```
      static int x = 1;
```

to tym samym zdefiniujemy *statyczną zmienną lokalną* `x`. Czas życia takiej zmiennej będzie taki sam jak dla zmiennych globalnych, ale zasięg pozostanie lokalny.

2.4.1. Operator zasięgu

Każda deklaracja identyfikatora wprowadza go w pewien zasięg. Oznacza to, że dana nazwa jest widoczna i można ją używać jedynie w określonej części programu. Załóżmy np., że w programie jednoplikowym chcemy używać tej samej nazwy (identyfikatora) dla zmiennej globalnej i zmiennej lokalnej, która ukrywa zmienną globalną. Dla uzyskania dostępu do ukrytej zmiennej globalnej wykorzystuje się jednoargumentowy **operator zasięgu** o symbolu **::**. Poprzedzenie nazwy zmiennej symbolem **::** informuje kompilator, że operacja będzie wykonywana na zmiennej globalnej. Podany niżej prosty przykład ilustruje wykorzystanie operatora zasięgu.

Przykład 2.8.

```
// Operator zasiegu ::
#include <iostream.h>
int z;
//zmienna globalna typu int inicjowana zerem
void main() {
      char znak;
      int x,y ;      // zmienne lokalne typu int
      double z;      // zmienna lokalna typu double
      z = 1.25816;  // przypisanie do zmiennej lokalnej
      ::z = 12;     // przypisanie do zmiennej globalnej
/*    Teraz przypisanie do zmiennej lokalnej
      wartosci zmiennej globalnej z                  */
      y = ::z;
      cout << "Podaj liczbe typu int: " ; cin  >> x;
      cout << "Wartosc x wynosi: " << x << endl;
      cout << "Wartosc zmiennej lokalnej z wynosi: "
           << z << endl;
      cout << "Wartosc zmiennej globalnej z wynosi: "
           << y << endl;
      cout << "Podaj dowolny znak, po czym ENTER: " ;
      cin  >> znak;
}
```

Dyskusja. W podanym przykładzie deklaracja definiująca `int z;` czyni zmienną `z` widzialną dla całego programu. Inaczej mówiąc, zasięg zmiennej `z` typu **int** obejmuje cały plik z programem źródłowym. Tak określona zmienna `z` może być używana w bloku funkcji `main` aż do deklaracji `double z;`, która przesłoniła (ukryła) zmienną globalną `z` typu **int**. Mimo to, dostęp do zmiennej globalnej nie został bezpowrotnie stracony, a to dzięki operatorowi zasięgu o symbolu "::". Identyfikator poprzedzony operatorem zasięgu zapewnia dostęp do zmiennej globalnej. Po uruchomieniu programu i wprowadzeniu z klawiatury wartości x, np. 16, a następnie znaku, np. 'a', wygląd ekranu monitora będzie taki:

Podaj liczbe typu int: 16
Wartosc x wynosi: 16
Wartosc zmiennej lokalnej z = 1.25816
Wartosc zmiennej globalnej z = 12
Podaj dowolny znak, po czym ENTER: a

Zwróćmy uwagę na następujące nowe elementy w programie:

– Funkcja main jest typu **void**, a więc nie zwraca żadnej wartości do otoczenia (systemu operacyjnego). Wpisane jako typ argumentu słowo kluczowe **void** oznacza, że do funkcji main nie jest przekazywany z otoczenia żaden argument. W języku C++ zapisy

```
fun();
```

oraz

```
fun(void);
```

są równoważne i oznaczają, że lista argumentów funkcji jest pusta. W związku z tym druga z deklaracji zawiera rozwlekłość (redundancję) zapisu. Tym niemniej argument **void** umieszcza się niekiedy wewnątrz operatora wywołania funkcji dla celów dokumentacyjnych.

– Charakterystyczny jest zapis instrukcji przypisania y = ::z; zmiennej globalnej z do zmiennej lokalnej y.

– W prezentowanym programie pojawił się *operator pobierania* (ang. extraction operator) '>>', który czyta wartości ze standardowego strumienia wejściowego cin. Strumień cin, podobnie jak cout, jest zdefiniowany w pliku iostream.h. Analogicznie jak operator wstawiania ('<<'), operator pobierania "wskazuje" kierunek przepływu danych: z postaci wyrażenia

```
cin >> x;
```

łatwo się domyślić, że chodzi tutaj o przesłanie danych do obiektu `x`.

3. Instrukcje i wyrażenia

W języku C++ każde działanie jest związane z pewnym wyrażeniem. Termin **wyrażenie** oznacza sekwencję operatorów i operandów (argumentów), która określa **operacje**, tj. rodzaj i kolejność obliczeń. **Operandem** nazywa się wielkość, poddaną operacji, która jest reprezentowana przez odpowiedni **operator**. Np. test na równość jest reprezentowany przez operator "**==**". Operatory, które oddziaływują tylko na jeden operand, nazywa się **jednoargumentowymi** (unarnymi). Przykładem może być wyrażenie `*wsk`, którego wynikiem jest wartość, zapisana pod adresem wskazywanym przez zmienną `wsk`. Operatory dwuargumentowe nazywa się **binarnymi**; ich argumenty określa się jako **operand lewy** i **operand prawy**. Niektóre operatory reprezentują zarówno operacje jednoargumentowe, jak i dwuargumentowe; np. operator *****, który wystąpił w wyrażeniu `*wsk`, w innym wyrażeniu, np.

```
zmienna1* zmienna2
```

reprezentuje binarny operator mnożenia.

Najprostszymi postaciami wyrażeń są wyrażenia stałe. W tym przypadku "operand" występuje bez operatora. Przykładami takich wyrażeń mogą być:

3.14159 "abcd"

Wynikiem wartościowania 3.14159 jest 3.14159 typu **double**; wynikiem wartościowania abcd jest adres pamięci pierwszego elementu łańcucha (typu char*).

Wyrażeniem złożonym nazywa się takie wyrażenie, w którym występuje dwa lub więcej operatorów. Wartościowanie wyrażenia przebiega w porządku, określonym pierwszeństwem operatorów i w kierunku, określonym przez kierunek wiązania operatorów.

Instrukcja jest najmniejszą samodzielną, wykonywalną jednostką programową. Każda instrukcja prosta języka C++ kończy się średnikiem, który jest dla niej symbolem terminalnym. Najprostszą jest **instrukcja pusta**, będąca pustym ciągiem znaków, zakończonym średnikiem:

```
; //instrukcja pusta
```

Instrukcja pusta jest użyteczna w przypadkach gdy składnia wymaga obecności instrukcji, natomiast nie jest wymagane żadne działanie. Nadmiarowe instrukcje puste nie są traktowane przez kompilator jako błędy syntaktyczne, np. zapis

```
int i;;
```

składa się z dwóch instrukcji: instrukcji deklaracji int i; oraz instrukcji pustej.

Instrukcją złożoną nazywa się sekwencję instrukcji ujętą w parę nawiasów klamrowych:

```
{
instrukcja-1
instrukcja-2
...
instrukcja-n
}
```

W składni języka taka sekwencja jest traktowana jako jedna instrukcja. Instrukcje złożone mogą być zagnieżdżane, np.

```
{
instrukcja-1
...
instrukcja-i
    {
    instrukcja-j
    ...
    instrukcja-n }
}
```

Jeżeli pomiędzy nawiasami klamrowymi występują instrukcje deklaracji identyfikatorów (nazw), to taką instrukcję złożoną nazywamy **blokiem**. Każda nazwa zadeklarowana w bloku ma zasięg od punktu deklaracji do zamykającego blok nawiasu klamrowego. Jeżeli nadamy identyczną nazwę identyfikatorowi, który był już wcześniej zadeklarowany, to poprzednia deklaracja zostanie przesłonięta na czas wykonania danego bloku; po opuszczeniu bloku jej ważność zostanie przywrócona.

3.1. Instrukcja przypisania

Dowolne wyrażenie zakończone średnikiem jest nazywane **instrukcją wyrażeniową** (ang. expression statement) lub krótko **instrukcją**, ponieważ większość używanych instrukcji stanowią instrukcje-wyrażenia. Jedną z najprostszych jest **instrukcja przypisania** o postaci:

```
zmienna = wyrażenie;
```

gdzie symbol "=" jest **operatorem przypisania**. Następujące napisy są instrukcjami języka C++:

```
int liczba;
liczba = 2 + 3;
cout << liczba;
```

Pierwsza z nich jest instrukcją deklaracji; definiuje ona obszar pamięci związany ze zmienną liczba, w którym mogą być przechowywane wartości całkowite. Drugą jest instrukcja przypisania. Umieszcza ona wynik dodawania (wartość wyrażenia) dwóch liczb w obszarze pamięci przeznaczonym na zmienną liczba. Trzecią jest poznana już instrukcja wyprowadzania zawartości obszaru pamięci, związanego ze zmienną liczba, na terminal użytkownika.

Zauważmy, że jeżeli w instrukcji nie występuje operator przypisania, to jej wykonanie może, ale nie musi, zmienić wartości żadnej zmiennej. Weźmy dla przykładu następujący ciąg instrukcji:

```
int i = 10;
int j = 20;
i + j;
i = i + j;
```

W instrukcjach deklaracji `int i = 10;` , `int j = 20;` znaki "=" nie są operatorami przypisania, lecz operatorami inicjowania zmiennych `i` oraz `j` wartościami początkowymi 10 oraz 20. Zakończone średnikiem wyrażenie `i+j` jest instrukcją, której wykonanie nie zmieni wartości żadnej ze zmiennych. Natomiast w instrukcji `i = i + j;` znak "=" jest operatorem przypisania wartości wyrażenia `i + j` do zmiennej `i`, a więc po wykonaniu tej instrukcji wartość `i` wyniesie 30.
Przy okazji zwróćmy uwagę na tzw. *efekty uboczne* wyrażeń. Termin ten odnosi się do wyrażeń, których wartościowanie zmienia zawartość komórki pamięci (lub pliku). Np. wyrażenie `i + j` nie daje efektów ubocznych, ponieważ nie umieszcza w pamięci wyniku dodawania. Natomiast wyrażenie `i = i + j` daje efekt uboczny, ponieważ zmienia zawartość pamięci, przypisując nową wartość do zmiennej `i`, zaś wynik dodawania `i + j` (tj. 30), już niepotrzebny, zostanie odrzucony.

Z powyższego wynika, że samo **przypisanie** jest wyrażeniem, którego wartość jest równa wartości przypisywanej argumentowi z lewej strony operatora przypisania. Instrukcja przypisania powstaje przez dodanie średnika po zapisie przypisania. Skoro tak, to jest oczywiste, że przypisania można wykorzystywać w wyrażeniach, co pozwala pisać zwięzłe, klarowne i łatwe w czytaniu programy. Ilustracją jest poniższy przykład.

Przykład 3.1.

```
a = b;
a = b + c;
a = (b + c)/d;
a = e > f;
a = (e > f && c < d) + 1;
a = a << 3;
```

Jednak skorzystanie z wymienionej cechy przypisania może również dać efekt odwrotny dla czytelności programu; np. instrukcja

```
a = (b = c + d)/(e = f + g);
```

jest raczej mało czytelna, podczas gdy równoważna sekwencja instrukcji przypisania

```
b = c + d;
e = f + g;
a = b/e;
```

jest bardziej elegancka i łatwo czytelna.

W instrukcji przypisania

```
zmienna = wyrażenie;
```

zmienna jest nazywana **l-wartością** lub **modyfikowalną l-wartością**. Nazwa ta wywodzi się stąd, że zmienna, umieszczona po lewej stronie operatora przypisania, jest wyrażeniem (tutaj po prostu identyfikatorem) reprezentującym w sposób symboliczny pewien obszar (adres) pamięci. Umieszczone w tym obszarze dane mogą być zmieniane przypisaniem wartości wyrażenie; wartość tę nazywa się czasem **r-wartością**. Dodatkową ilustracją wprowadzonej terminologii może być przypisanie:

```
a = a + b
```

Tutaj `a` oraz `b` po prawej stronie są r-wartościami, odczytywanymi pod adresami symbolicznymi a i b; natomiast `a` po lewej stronie jest adresem, pod którym zostaje zapisany wynik dodawania poprzedniej zawartości `a` i zawartości `b`.

☞ *Uwaga. l-wartość jest modyfikowalna, jeżeli nie jest ona nazwą funkcji, nazwą tablicy, bądź const.*

3.2. Operatory

Język C++ oferuje ogromne bogactwo operatorów, zarówno dla argumentów typów podstawowych, jak i typów pochodnych. Jest to jedna z przyczyn, dla której język ten szybko się rozpowszechnia i staje się de facto standardem przemysłowym.

3.2.1. Operatory arytmetyczne

Operatory arytmetyczne służą do tworzenia wyrażeń arytmetycznych. W języku C++ przyjęto jako normę stosowanie tzw. arytmetyki mieszanej, w której wartość argumentu operatora jest automatycznie przekształcana przez kompilator do typu, podanego w deklaracji tego argumentu. W związku z tym nie przewidziano oddzielnych operatorów dla typów całkowitych i typów zmiennopozycyjnych, za wyjątkiem operatora **%**, stosowalnego jedynie dla typów **short int**, **int** i **long int**. W tablicy 3.1 zestawiono dwuargumentowe operatory arytmetyczne języka C++.

Tablica 3.1 Zestawienie operatorów arytmetycznych

Symbol operatora	Funkcja	Zastosowanie
+	dodawanie	wyrażenie + wyrażenie
-	odejmowanie	wyrażenie - wyrażenie
*	mnożenie	wyrażenie * wyrażenie
/	dzielenie	wyrażenie / wyrażenie
%	operator reszty z dzielenia	wyrażenie % wyrażenie

Wszystkie operatory za wyjątkiem operatora **%** można stosować zarówno do argumentów całkowitych, jak i zmiennopozycyjnych. Operatory **+** i **-** można również stosować jako operatory jednoargumentowe. Jeżeli przy dzieleniu liczb całkowitych iloraz zawiera część ułamkową, to jest ona odrzucana; np. wynik dzielenia
18/6 i 18/5
jest w obu przypadkach równy 3. Operator reszty z dzielenia (**%**) można stosować tylko do argumentów całkowitych; np. 18 % 6 daje wynik 0, a 18 % 5 daje wynik 3.
W pewnych przypadkach wartościowanie wyrażenia arytmetycznego daje wynik niepoprawny lub nieokreślony. Są to tzw. *wyjątki*. Mogą one być spowodowane niedopuszczalnymi operacjami matematycznymi (np. dzieleniem przez zero), lub też mogą wynikać z ograniczeń sprzętowych (np. nadmiar przy próbie reprezentacji zbyt dużej liczby). W takich sytuacjach stosuje się własne, lub predefiniowane metody obsługi wyjątków.

3.2.2. Operatory relacji

Wszystkie operatory relacji są dwuargumentowe. Jeżeli relacja jest prawdziwa, to jej wartością jest 1; w przypadku przeciwnym wartością relacji jest 0. Warto zwrócić uwagę na zapis operatora równości ==, który początkujący programiści często mylą z operatorem przypisania =.

Tablica 3.2 Zestawienie operatorów relacji

Symbol operatora	Funkcja	Zastosowanie
<	mniejszy	wyrażenie < wyrażenie
<=	mniejszy lub równy	wyrażenie <= wyrażenie
>	większy	wyrażenie > wyrażenie
>=	większy lub równy	wyrażenie >= wyrażenie
==	równy	wyrażenie == wyrażenie
!=	nierówny	wyrażenie != wyrażenie

☞ *Uwaga. Argumenty operatorów relacji muszą być typu arytmetycznego lub wskaźnikowego.*

3.2.3. Operatory logiczne

Wyrażenia połączone dwuargumentowymi operatorami logicznymi koniunkcji i alternatywy są zawsze wartościowane od strony lewej do prawej. Dla operatora **&&** otrzymujemy wartość 1 (prawda) wtedy

i tylko wtedy, gdy wartościowanie obydwu operandów daje 1. Dla operatora || otrzymujemy wartość 1, gdy co najmniej jeden z operandów ma wartość 1.

Tablica 3.3 Zestawienie operatorów logicznych

Symbol operatora	Funkcja	Składnia
!	negacja	!wyrażenie
&&	koniunkcja	wyrażenie && wyrażenie
||	alternatywa	wyrażenie || wyrażenie

3.2.4. Bitowe operatory logiczne

Język C++ oferuje sześć tzw. **bitowych operatorów logicznych**, które interpretują operand(-y) jako uporządkowany ciąg bitów. Każdy bit może przyjmować wartość 1 lub 0.

Tablica 3.4 Zestawienie bitowych operatorów logicznych

Symbol operatora	Funkcja	Składnia
&	bitowa koniunkcja	wyrażenie & wyrażenie
|	bitowa alternatywa	wyrażenie | wyrażenie
^	bitowa różnica symetryczna	wyrażenie ^ wyrażenie
<<	przesunięcie w lewo	wyrażenie << wyrażenie
>>	przesunięcie w prawo	wyrażenie >> wyrażenie
~	bitowa negacja	~wyrażenie

Argumenty (synonim operandów) tych operatorów muszą być całkowite, a więc typu **char**, **short int**, **int** i **long int**, zarówno bez znaku, jak i ze znakiem. Ze względu na różnice pomiędzy reprezentacjami liczb ze znakiem w różnych implementacjach, zaleca się używanie operandów bez znaku.
Bitowy operator koniunkcji **&** stosuje się często do "zasłaniania" (maskowania) pewnego zbioru bitów; np. instrukcja

```
n = n & 0177; //Liczba oktalna 0177==127 dec
```

zeruje wszystkie oprócz 7 najniższych bitów zmiennej n.
Bitowy operator alternatywy **|** stosuje się do "ustawiania" bitów; np. instrukcja

```
n = n | MASK;
```

ustawia jedynki na tych bitach zmiennej n, które w MASK są równe 1.
Bitowy operator różnicy symetrycznej (ang. exclusive OR) ustawia jedynkę na każdej pozycji, gdzie jego operandy się różnią (np. 0 i 1 lub 1 i 0) i zero tam, gdzie są takie same. Np.

```
0177 ^ 0176
```

daje po obliczeniu wartość 1.

Operatory przesunięcia w lewo **<<** i w prawo **>>** przesuwają reprezentującą daną liczbę sekwencję bitów odpowiednio w lewo lub w prawo, wypełniając zwolnione bity zerami. Np. dla `i = 6;` instrukcja `i = i << 2;` zmieni wartość `i` na 24.
Jednoargumentowy operator bitowej negacji ~ daje uzupełnienie jedynkowe swojego całkowitego argumentu. Np. instrukcja

```
n = n & 077;
```

ustawia ostatnie sześć bitów zmiennej `n` na zero.

3.2.5. Operatory przypisania

Przypadkiem szczególnym instrukcji przypisania jest instrukcja:

```
a = a op b;
```

gdzie `op` może być jednym z dziesięciu operatorów: `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`. Dla bardziej zwięzłego zapisu wprowadzono w języku C++ złożenia znaku przypisania "=" z symbolem odpowiedniego operatora, co pozwala zapisać powyższą instrukcję w postaci:

```
a op= b;
```

Weźmy dla przykładu instrukcję przypisania `a = a << 3;`, której wykonanie przesuwa wartość zmiennej `a` o trzy pozycje w lewo, a następnie przypisuje wynik do `a`. Instrukcję tę można przepisać w postaci: `a <<= 3;`
Operatory powstałe ze złożenia operatora przypisania "=" z każdym z wymienionych 10 operatorów zestawiono w Tablicy 3.5.

Tablica 3.5 Zestawienie wieloznakowych operatorów przypisania

Symbol operatora	Zapis skrócony	Zapis rozwinięty
+=	a += b	a = a + b;
-=	a -= b	a = a - b;
*=	a *= b	a = a * b;
/=	a /= b	a = a / b;
%=	a %= b	a = a % b;
<<=	a <<= b	a = a << b;
>>=	a >>= b	a = a >> b;
&=	a &= b	a = a & b;
|=	a |= b	a = a | b;
^=	a ^= b	a = a ^ b;

☞ *Uwaga.Wszystkie operatory wymagają l-wartości jako ich lewego argumentu, zaś typ wyrażenia przypisania jest typem jego lewego argumentu. Wynikiem przypisania jest wartość, zapisana w lewym argumencie; jest to l-wartość.*

3.2.6. Operator sizeof

Rozmiary dowolnego obiektu (stałej, zmiennej, etc.) języka C++ wyraża się wielokrotnością rozmiaru typu **char**; zatem, z definicji

```
sizeof(char) == 1
```

Operator `sizeof` jest jednoargumentowy. Składnia języka przewiduje dwie postacie wyrażeń z operatorem sizeof:

```
sizeof(nazwa-typu)
sizeof wyrażenie
```

Dla podstawowych typów danych obowiązują następujące relacje:

1 == sizeof(char) <= sizeof(short int) <= sizeof(int) <= sizeof(long int)
sizeof(float) <= sizeof(double) <= sizeof(long double)
sizeof(I) == sizeof(signed I) == sizeof(unsigned I)

gdzie I może być **char**, **short int**, **int**, lub **long int**.

Ponadto – dla dowolnej platformy sprzętowej można być pewnym, że typ **char** ma co najmniej 8 bitów, **short int** co najmniej 16 bitów, a **long int** co najmniej 32 bity.
Wiemy już, że każdej zmiennej typu **char** można przypisać jeden znak. Jeżeli zatem znak jest zapisywany na 8 bitach w maszynowym zbiorze znaków (np. pełny zbiór ASCII), to operator sizeof zastosowany do dowolnego argumentu będzie zwracał liczbę bajtów zajmowanych przez jego argument.

☞ *Uwaga.Operatora sizeof nie można stosować do funkcji (ale można do wskaźnika do funkcji), pola bitowego, niezdefiniowanej klasy, typu **void** oraz tablicy bez podanych wymiarów.*

3.2.7. Operator warunkowy "?:"

Jest to jedyny operator trójargumentowy w języku C++. Wyrażenie warunkowe, utworzone przez zastosowanie operatora "?:" ma postać:

```
wyrażenie1 ? wyrażenie2 : wyrażenie3
```

Wartość tak utworzonego wyrażenia jest obliczana następująco. Najpierw wartościowane jest wyrażenie1. Jeżeli jest to wartość niezerowa (prawda), to wartościowane jest wyrażenie2 i wynikiem obliczeń jest jego wartość. Przy zerowej wartości (fałsz) wyrażenia wyrażenie1 wynikiem obliczeń będzie wartość wyrażenia wyrażenie3.
Przykład 3.2.

```
#include <iostream.h>
int main() {
int a,b,z;
cin >> a >> b;
z = (a > b) ? a : b;  // z==max(a,b)
cout << z;
return 0;
}
```

3.2.8. Operatory zwiększania/zmniejszania

W języku C++ istnieją operatory, służące do zwięzłego zapisu zwiększania o 1 (++) i zmniejszania o 1 (--) wartości zmiennej. Zamiast zapisu

`n=n+1` (lub `n+=1`) `n=n-1` (lub `n-=1`)

piszemy krótko

`++n`, bądź `n++` `--n`, bądź `n--`

przy czym nie jest obojętne, czy dwuznakowy operator "++" lub "--" zapiszemy przed, bądź za nazwą zmiennej. Notacja przedrostkowa (++n) oznacza, że wyrażenie ++n zwiększa n zanim wartość n zostanie użyta, natomiast n++ zwiększa n po użyciu dotychczasowej wartości n. Tak więc wyrażenia ++n oraz n++ (i odpowiednio --n oraz n--) są różne. Ilustruje to poniższy przykład.

Przykład 3.3.

```
#include <iostream.h>
int main()  {
 int i,j = 5;
 i = j++ ; // przypisz 5 do i, po czym przypisz 6 do j
 cout << "i=" << i << ", j=" << j << endl;
 i = ++j; // przypisz 7 do j, po czym przypisz 7 do i
 cout << "Teraz i=" << i << ", j=" << j << endl;
 // j++ = i; zle! j++ nie jest l-wartoscia
 return 0;
}
```

☞ *Uwaga. Argumentami operatorów "++" oraz "--" muszą być modyfikowalne l-wartości typu arytmetycznego lub wskaźnikowego. Np. zapis: n = (n + m)++ jest błędny, ponieważ (n + m) nie jest l-wartością. Typ wyniku jest taki sam, jak typ argumentu. Dla obu postaci: przedrostkowej (np. ++n) i przyrostkowej (np. n++) wynik nie jest l-wartością.*

3.2.9. Operator przecinkowy

Operator przecinkowy ',' pozwala utworzyć wyrażenie, składające się z ciągu wyrażeń składowych, rozdzielonych przecinkami. Wartością takiego wyrażenia jest wartość ostatniego z prawej elementu ciągu, zaś wartościowanie przebiega od elementu skrajnego lewego do skrajnego prawego. Przykładem wyrażenia z operatorem przecinkowym może być:

```
num++, num + 10
```

gdzie `num` jest typu **int**.

Wartościowanie powyższego wyrażenia z operatorem przecinkowym przebiega w następujący sposób:
- Najpierw jest wartościowane wyrażenie num++, w wyniku czego zostaje zmieniona zawartość komórki pamięci o nazwie num (efekt uboczny).
- Następnie jest wartościowane wyrażenie num + 10 i ta wartość jest wartością końcową.

Weźmy inny przykład:

```
double x, y, z;
z = (x = 2.5, y = 3.5, y++);
```

Wynikiem wartościowania wyrażenia z dwoma operatorami przecinkowymi będą wartości: x==2.5, y==4.5 oraz z==3.5 (wartość z nie będzie równa 4.5, ponieważ do y przyłożono przyrostkowy operator '++').

3.2.10. Rzutowanie (konwersja) typów

W języku C++ raczej normą niż wyjątkiem jest "arytmetyka mieszana", tj. sytuacja, gdy w instrukcjach i wyrażeniach argumenty operatorów są różnych typów. Możemy np. dodawać wartości wyrażeń typu **int** do wartości typu **long int**, wartości typu **float** do wartości typu **double**, etc. W takich przypadkach, jeszcze przed wykonaniem żądanej operacji, argumenty każdego operatora muszą zostać przekształcone do jednego, tego samego typu, dopuszczalnego dla danego operatora. Operację taką nazywa się **rzutowaniem** lub **konwersją** (ang. cast). Konwersja typów może być *niejawna*, wykonywana automatycznie przez kompilator bez udziału programisty. Język pozwala programiście dokonywać także konwersji *jawnych*, oferując mu odpowiednie *operatory konwersji*.
Zarówno konwersje jawne, jak i niejawne, muszą być *bezpieczne*, tzn. takie, aby w wyniku konwersji nie była tracona żadna informacja. Jest pewnym, że jeśli liczba bitów w maszynowej reprezentacji wielkości podlegającej konwersji nie ulega zmianie, bądź wzrasta po konwersji, to taka konwersja jest bezpieczna. Bezpieczna konwersja argumentu "węższego" typu do "szerszego" jest nazywana **promocją typu**. Typowym przykładem promocji jest automatyczna konwersja typu **char** do typu **int**. Powód jest oczywisty: wszystkie predefiniowane operacje na literałach i zmiennych typu **char** są faktycznie wykonywane na liczbach porządkowych znaków, pobieranych ze zbioru kodów maszynowych (np. ASCII). Powyższe dotyczy również typu **short int** oraz **enum**.
Podane niżej zestawienie typów od "najwęższego" do "najszerszego" określa, który argument operatora binarnego będzie przekształcany.

int
unsigned int
long int
unsigned long int
float
double
long double

Typ, który występuje jako pierwszy w wykazie, podlega promocji do typu, występującego jako następny. Np. jeśli argumenty operatora są **int** i **long int**, to argument **int** zostanie przekształcony do typu **long int**; jeżeli typy argumentów są **long int** i **long double**, to operand typu **long int** będzie przekształcony do typu **long double**, etc. Weźmy dla przykładu następującą sekwencję instrukcji:

```
int n = 3;
long double z;
z = n + 3.14159;
```

W ostatniej instrukcji najpierw zmienna n zostanie przekształcona do typu **double**; jej wartość stanie się równa 3.0. Wartość ta zostanie dodana do 3.14159, dając wynik 6.14159 typu **double**. Otrzymany wynik zostanie następnie przekształcony do typu **long double**.

Przykładem konwersji zwężającej może być wykonanie następującej sekwencji instrukcji:

```
int n = 10;
n *= 3.1;
```

W drugiej instrukcji mamy dwie konwersje. Najpierw n jest przekształcane do typu **double** i wartości 10.0. Po wymnożeniu przez 3.1 otrzymuje się wynik 31.0, który zostanie następnie zawężony do typu **int**. Ta zawężona wartość (31) zostanie przypisana do zmiennej n.
Konwersja jawna może być wymuszona przez programistę za pomocą jednoargumentowego operatora konwersji '()' o składni

(typ) wyrażenie

lub

typ (wyrażenie)

W obu przypadkach wyrażenie zostaje przekształcone do typu typ zgodnie z regułami konwersji. Przykładowo, obie poniższe instrukcje

```
(double) 25;
double (25);
```

dadzą wartość 25.0 typu **double**.

Konwersja jawna bywa stosowana dla uniknięcia zbędnych konwersji niejawnych. Np. wykonanie instrukcji (n jest typu **int**)

```
n = n + 3.14159;
```

wymaga konwersji n do **double**, a następnie zawężenia sumy do **int**.
Modyfikując tę instrukcję do postaci

```
n = n + int(3.14159);
```

mamy tylko jedną konwersję z typu **double** do **int**.

Wynik konwersji nie jest l-wartością (wyjątek – typ referencyjny, który zostanie omówiony później), a więc można go przypisywać, np

```
float f = float(10);
```

☞ *Uwaga. Gdy konwersja jawna nie jest niezbędnie potrzebna, należy jej unikać, ponieważ programy, w których używa się jawnych konwersji, są trudniejsze do zrozumienia.*

Przykład 3.4.

```
/*      rzutowanie(konwersja typow)       */
#include <iostream.h>

// Deklaracje zmiennych globalnych
char   a;
int    b;

int main()
{
 cout << "int(a)== " << int(a) << '\n';
 cout << "b== " << b << '\n';
 a = 'A';
 cout << "a po przypisaniu== " << a << endl;
 b = int (a);
 cout << " (b = int (a))== " << b << endl;
 a = (char) b;
 cout << " (a = (char) b)== " << a << endl;
 return 0;
}
```

Dyskusja. Zadeklarowanym zmiennym globalnym a i b kompilator nada automatycznie zerowe wartości początkowe (a=='\0', b==0). Wydruk z programu ma postać:

a== 0
b== 0
a po przypisaniu== A
(b = int (a))== 65
(a = (char) b)== A

3.2.11. Hierarchia i łączność operatorów.

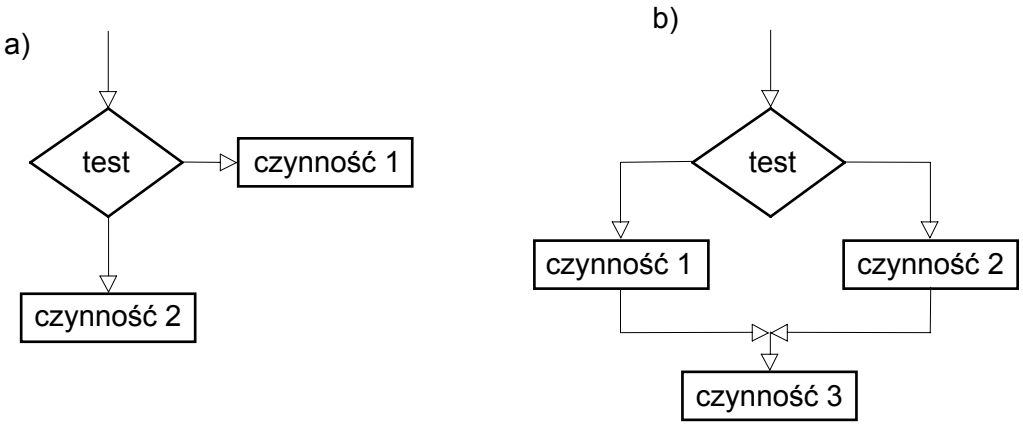
Dla poprawnego posługiwania się operatorami w wyrażeniach istotna jest znajomość ich priorytetów i kierunku wiązania (łączności). Przy wartościowaniu wyrażeń obowiązuje zasada wykonywania jako pierwszej takiej operacji, której operator ma wyższy priorytet i w tym kierunku, w którym operator wiąże swój argument (argumenty). Programista może zmienić kolejność wartościowania, zamykając część wyrażenia (podwyrażenie) w nawiasy okrągłe. Wówczas jako pierwsze będą wartościowane te podwyrażenia, które są zawarte w nawiasach najgłębiej zagnieżdżonych (zanurzonych).
Poniżej zestawiono operatory języka C++ według ich priorytetów; bardziej szczegółowy wykaz zamieszczono w Dodatku B. Hierarchię operatorów należy rozumieć w ten sposób, że wyższe pozycje w podanej niżej tablicy oznaczają wyższy priorytet: wyrażenia, w których argumenty są powiązane operatorami o wyższym priorytecie, są wykonywane jako pierwsze.

Tablica 3.6 Hierarchia operatorów

Operatory	Kierunek wiązania
:: zasięg globalny (unarny)	od prawej do lewej
:: zasięg klasy (binarny)	od lewej do prawej
-> . () przyrostkowy++ przyrostkowy--	od lewej do prawej
przedrostkowy ++ przedrostkowy -- ~ ! unarny + unarny - unarny & (typ) sizeof new delete	od prawej do lewej
->* .*	od lewej do prawej
* / %	od lewej do prawej
+ -	od lewej do prawej
<< >>	od lewej do prawej
< <= > >=	od lewej do prawej
== !=	od lewej do prawej
&	od lewej do prawej
^	od lewej do prawej
|	od lewej do prawej
&&	od lewej do prawej
||	od lewej do prawej
?:	od lewej do prawej
= *= /= %= += -= <<= >>= &= ^= |=	od prawej do lewej
,	od lewej do prawej

3.3. Instrukcje selekcji

Instrukcje selekcji (wyboru) wykorzystuje się przy podejmowaniu decyzji. Konieczność ich stosowania wynika stąd, że kodowane w języku programowania algorytmy tylko w bardzo prostych przypadkach są czysto sekwencyjne. Najczęściej kolejne kroki algorytmu są zależne od spełnienia pewnego warunku lub kilku warunków. Rysunek 3-1 ilustruje dwie takie typowe sytuacje.



Rys. 3-1 Sieć działań warunkowych

W sytuacji z rys. 3-1a) spełnienie testowanego warunku oznacza wykonanie sekwencji działań: czynność 1 - czynność 2; przy warunku niespełnionym wykonywana jest czynność 2 (czynność 1 jest pomijana). W sytuacji z rys. 3-1b) pozytywny wynik testu oznacza wykonanie sekwencji działań: czynność 1 - czynność 3, a wynik negatywny: czynność 2 - czynność 3.

3.3.1. Instrukcja if

Instrukcja **if** jest implementacją schematów podejmowania decyzji, pokazanych na rysunku 6-1. Formalny zapis jest następujący:

if(wyrażenie) instrukcja

lub

if(wyrażenie) instrukcja1 else instrukcja2

gdzie wyrażenie musi wystąpić w nawiasach okrągłych, zaś żadna z instrukcji nie może być instrukcją deklaracji.

☞ *Uwaga. Pamiętajmy, że instrukcje proste języka C++ kończą się średnikami!*

Wykonanie instrukcji **if** zaczyna się od obliczenia wartości wyrażenia. Jeżeli wyrażenie ma wartość różną od zera (prawda), to będzie wykonana instrukcja (lub instrukcja1); jeżeli wyrażenie ma wartość zero (fałsz), to w pierwszym przypadku instrukcja jest pomijana, a w drugim przypadku wykonywana jest instrukcja2. Każda z występujących tutaj instrukcji może być instrukcją prostą lub złożoną, bądź instrukcją pustą.
Ponieważ **if** sprawdza jedynie wartość numeryczną wyrażenia, dopuszcza się pewne skrótowe zapisy. Np. zamiast pisać

```
if(wyrażenie != 0)
```

pisze się często

```
if(wyrażenie)
```

Przykład 3.5.

```
//Instrukcje if
#include <iostream.h>
```

```
int main()  {
 int a, b, c = 0;
 cout << "Wprowadz dwie liczby calkowite: " << endl;
 cin >> a >> b;
 if(a*b > 0)
   if (a > b)
     c = a;
   else   c = b;
 cout << "Pierwszy test daje c= " << c << endl;
 if (a*b > 0)  {
   if (a > b)
   c =a;       }
 else  c = b;
 cout << "Drugi test daje c= " << c << endl;
 return 0;
}
```

Dyskusja. Mamy tutaj przykłady zagnieżdżonych instrukcji **if**. Jeżeli wprowadzimy z klawiatury dwie liczby o różnych znakach, np. a == 5, b == -2, to po sprawdzeniu, że a*b < 0 wbudowana instrukcja **if** razem ze swoją opcją **else** zostanie pominięta i **cout** wydrukuje Pierwszy test daje c== 0, tj. inicjalną wartość c. W drugiej konstrukcji opcja **else** dotyczy zewnętrznej instrukcji **if**, zatem **cout** wydrukuje:
Drugi test daje c== -2
Niektóre proste konstrukcje **if** można z powodzeniem zastąpić wyrażeniem warunkowym, wykorzystującym operator warunkowy "?:" . Np.

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

można zastąpić przez

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

Nawiasy okrągłe w ostatnim wierszu nie są konieczne; dodano je dla lepszej czytelności.

3.3.2. Instrukcja switch

Istrukcja **switch** służy do podejmowania decyzji wielowariantowych, gdy zastosowanie instrukcji **if-else** prowadziłoby do zbyt głębokich zagnieżdżeń i ewentualnych niejednoznaczności. Składnia instrukcji jest następująca:

```
switch (wyrażenie) instrukcja;
```

gdzie instrukcja jest zwykle instrukcją złożoną (blokiem), której instrukcje składowe są poprzedzane słowem kluczowym **case** z etykietą; wyrażenie, które musi przyjmować wartości całkowite, spełnia tutaj rolę *selektora wyboru*. W rozwiniętym zapisie

```
switch (wyrażenie)
{
  case etykieta-1 : instrukcje
          ...
  case etykieta-n : instrukcje
  default         : instrukcje
}
```

Etykiety są całkowitymi wartościami stałymi lub wyrażeniami stałymi. Nie może być dwóch identycznych etykiet. Jeżeli jedna z etykiet ma wartość równą wartości wyrażenia wyrażenie, to wykonanie zaczyna się od tego przypadku (ang. case – przypadek) i przebiega aż do końca bloku. Instrukcje po etykiecie `default` są wykonywane wtedy, gdy żadna z poprzednich etykiet nie ma aktualnej wartości selektora wyboru.
Przypadek z etykietą `default` jest opcją: jeżeli nie występuje i żadna z pozostałych etykiet nie przyjmuje wartości selektora, to nie jest podejmowane żadne działanie i sterowanie przechodzi do następnej po **switch** instrukcji programu.

☞ *Uwaga 1. Etykieta default i pozostałe etykiety mogą występować w dowolnym porządku.*

☞ *Uwaga 2. Każda z instrukcji składowych może być poprzedzona więcej niż jedną sekwencją case etykieta:, np. case et1: case et2: case et3: cout << "Trzy etykiety\n";*

Z powyższego opisu wynika, że – jak na razie – przedstawiona wersja instrukcji **switch** jest co najwyżej dwualternatywna i mówi: wykonuj wszystkie instrukcje od danej etykiety do końca bloku, albo: wykonuj instrukcje po default do końca bloku (bądź żadnej, jeśli default nie występuje). Wersja ta nie wychodzi zatem poza możliwości instrukcji **if**, bądź **if-else**. Dla uzyskania wersji z wieloma wzajemnie wykluczającymi się alternatywami musimy odseparować poszczególne przypadki w taki sposób, aby po wykonaniu instrukcji dla wybranego przypadku sterowanie opuściło blok instrukcji **switch**. Możliwość taką zapewnia instrukcja **break**. Zatem dla selekcji wyłącznie jednego z wielu wariantów składnia instrukcji **switch** będzie miała postać:

```
switch(wyrażenie)
{
  case et-1 : instrukcje
              break;
       ...
  case et-n : instrukcje
              break;
  default   : instrukcje
              break;
}
```

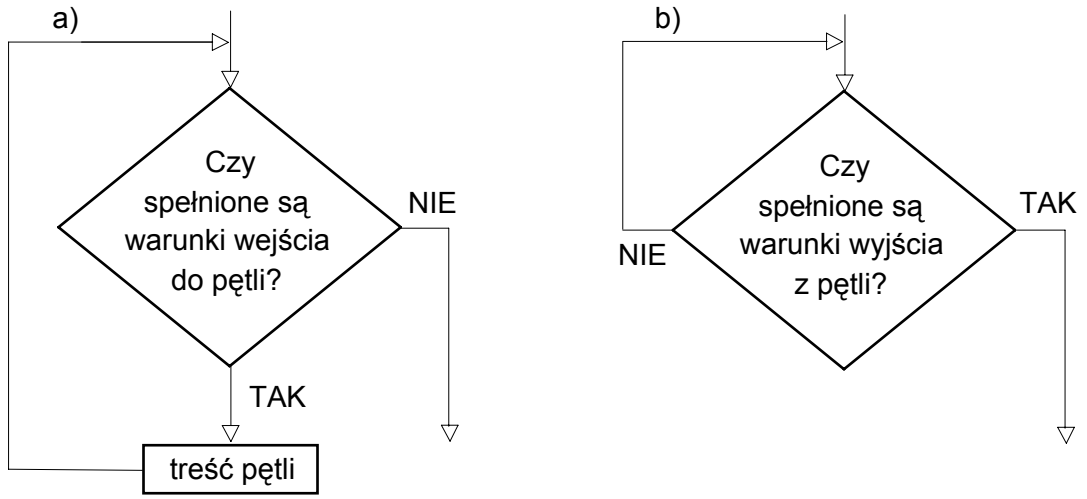
Przykład 3.6.

```
#include <iostream.h>
int main()  {
    char droga;
    int czas;
    cout << "Wprowadz litere A, B, lub C : " ;
    cin >> droga;
    cout << endl;
    if((droga=='A')||(droga=='B')||(droga=='C'))
    switch (droga)  {
     case 'A': case 'B': czas = 3;
                         cout << czas << endl;
                         break;
     case 'C':           czas = 4;
                         cout << czas << endl;
                         break;
     default:            droga = 'D';
                         czas = 5;
                         cout << czas << endl;
     }
    else cout << "Zostan w domu\n";
```

```
    return 0;
}
```

3.4. Instrukcje iteracyjne

Instrukcje powtarzania (pętli) lub iteracji pozwalają wykonywać daną instrukcję, prostą lub złożoną, zero lub więcej razy. W języku C++ mamy do dyspozycji trzy instrukcje iteracyjne (pętli): **while**, **do-while** i **for**. Różnią się one przede wszystkim metodą sterowania pętlą. Dla pętli **while** i **for** sprawdza się warunek wejścia do pętli (rys. 3-2a), natomiast dla pętli **do-while** sprawdza się warunek wyjścia z pętli (rys. 3-2b).



Rys. 3-2 Struktury pętli: a) z testem na wejściu, b) z testem na wyjściu

Instrukcje iteracyjne, podobnie jak instrukcje selekcji, można zagnieżdżać do dowolnego poziomu zagnieżdżenia. Jednak przy wielu poziomach zagnieżdżenia program staje się mało czytelny. W praktyce nie zaleca się stosować więcej niż trzech poziomów zagnieżdżenia.

3.4.1. Instrukcja while

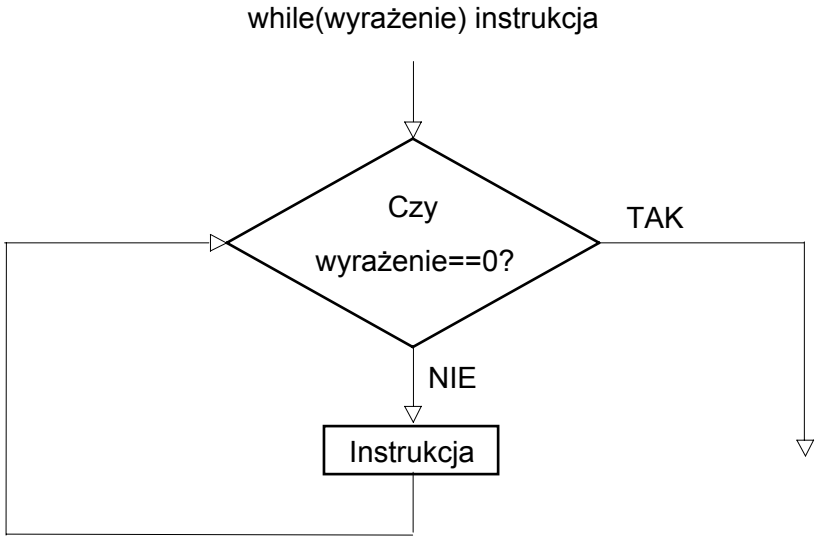
Składnia instrukcji **while** jest następująca:

```
while (wyrażenie) instrukcja
```

gdzie `instrukcja` może być instrukcją pustą, instrukcją prostą, lub instrukcją złożoną.
Sekwencja działań przy wykonywaniu instrukcji **while** jest następująca:

1. Oblicz wartość wyrażenia i sprawdź, czy jest równe zeru (fałsz). Jeżeli tak, to pomiń krok 2; jeżeli nie (prawda), przejdź do kroku 2.
2. Wykonaj instrukcję i przejdź do kroku 1.

Jeżeli pierwsze wartościowanie wyrażenia wykaże, że ma ono wartość zero, to instrukcja nigdy nie zostanie wykonana i sterowanie przejdzie do następnej instrukcji programu.
Rysunek 3-3 ilustruje w sposób poglądowy działanie instrukcji **while**.



Rys. 3-3 Sieć działań instrukcji while

Przykład 3.7.

```
#include <iostream.h>
#include <iomanip.h>
int main() {
    const int WIERSZ = 5;
    const int KOLUMNA = 15;
    int j, i = 1;
    while(i <= WIERSZ)
     {
       cout << setw(KOLUMNA - i) << '*';
       j = 1;
       while( j <= 2*i-2 )
        {
         cout << '*';
         j++;
        }
       cout << endl;
       i++;
     }
    return 0;
}
```

Wydruk z programu będzie miał postać:

```
    *
   ***
  *****
 *******
*********
```

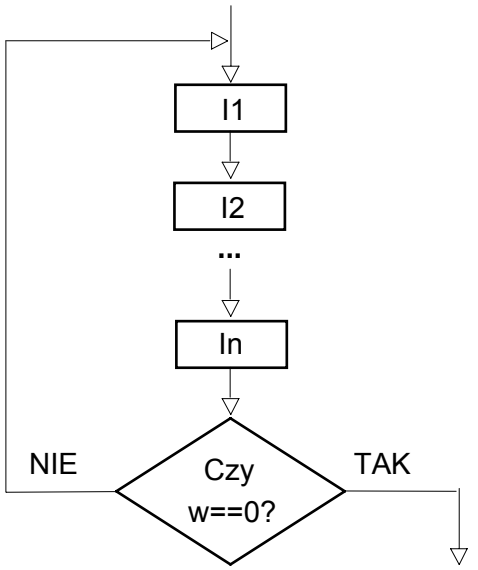
Dyskusja. W programie mamy zagnieżdżoną pętlę **while**. Pierwsze sprawdzenie wyrażenia j <= 2*i-2 w pętli wewnętrznej daje wartość 0 (fałsz), zatem w pierwszym obiegu pętla wewnętrzna zostaje pominięta. W drugim sprawdzeniu pętla wewnętrzna dorysowuje dwie gwiazdki, itd., aż do wyjścia z pętli zewnętrznej. Nowym elementem programu jest włączenie pliku nagłówkowego iomanip.h, w którym znajduje się deklaracja funkcji setw(int w). Funkcja ta służy do ustawiania szerokości pola wydruku na podaną liczbę w znaków.

3.4.2. Instrukcja do-while

Składnia instrukcji **do-while** ma postać:

```
do instrukcja while (wyrażenie)
```

gdzie `instrukcja` może być instrukcją pustą, instrukcją prostą lub złożoną.
Pętla **do-while** funkcjonuje według schematu, pokazanego na rysunku 3-4.




```
do {
      I1
      I2
       ...
      In  }
while(w);
```

I1, I2, ...,In - instrukcje

w - wyrażenie

Rys. 3-4 Sieć działań instrukcji do-while

W pętli *do-while* instrukcja (ciało pętli) zawsze będzie wykonana co najmniej jeden raz, ponieważ test na równość zeru wyrażenia jest przeprowadzany po jej wykonaniu. Pętla kończy się po stwierdzeniu, że wyrażenie jest równe zeru.
Przykład 3.8.

```
#include <iostream.h>
int main() {
    char znak;
    cout << "Wprowadz dowolny znak;\
    * oznacza koniec.\n";
    do {
       cout << ": ";
       cin  >> znak;
      }
   while  (znak != '*');
   return 0;
}
```

3.4.3. Instrukcja for

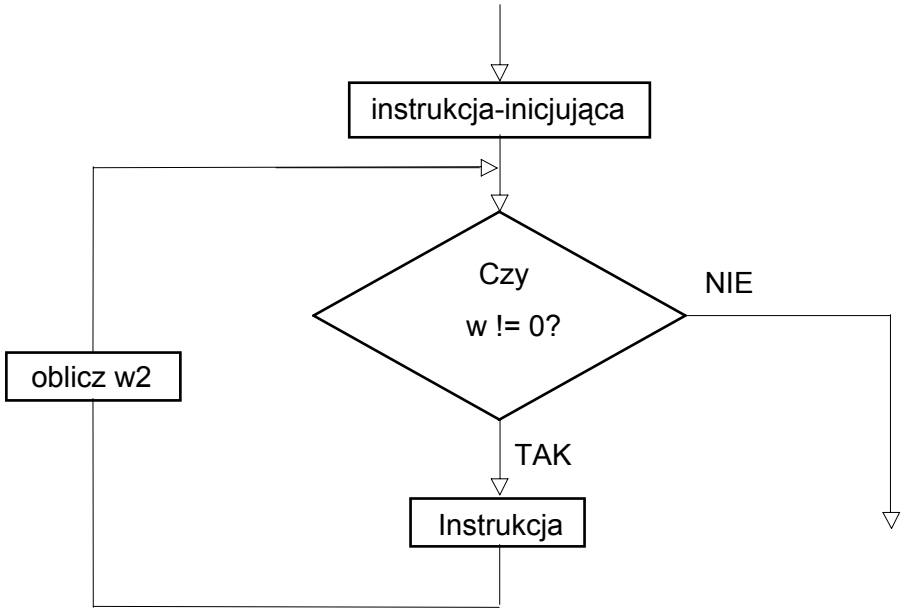
Jest to, podobnie jak instrukcja **while**, instrukcja sterująca powtarzaniem ze sprawdzeniem warunku zatrzymania na początku pętli. Stosuje się ją w tych przypadkach, gdy znana jest liczba obiegów pętli. Składnia instrukcji **for** jest następująca:

for(instrukcja-inicjująca wyrażenie1;wyrażenie2) instrukcja

gdzie instrukcja może być instrukcją pustą, instrukcją prostą lub złożoną.
Algorytm obliczeń dla pętli *for* jest następujący:

1. Wykonaj instrukcję o nazwie *instrukcja-inicjująca*. Zwykle będzie to zainicjowanie jednego lub kilku liczników pętli (zmiennych sterujących), ewentualnie inicjująca instrukcja deklaracji, np.
   ```
   for (i = 0; ...
   for (i =0, j = 1; ...
   for (int i = 0; ...
   ```
   Instrukcja inicjująca może być również instrukcją pustą, jeżeli zmienna sterująca została już wcześniej zadeklarowana i zainicjowana, np.
   ```
   int i = 1; for (; ...
   ```
2. Oblicz wartość wyrażenia *wyrażenie1* i porównaj ją z zerem. Jeżeli *wyrażenie1* ma wartość różną od zera (prawda) przejdź do kroku 3. Jeżeli *wyrażenie1* ma wartość zero, opuść pętlę.
3. Wykonaj instrukcję *instrukcja* i przejdź do kroku 4.
4. Oblicz wartość wyrażenia *wyrażenie2* (zwykle oznacza to zwiększenie licznika pętli) i przejdź do kroku 2.

Ilustracją opisanego algorytmu jest rysunek 3-5.



Rys. 3-5 Sieć działań instrukcji for

☞ *Uwaga 1. Jeżeli instrukcja inicjująca jest instrukcją deklaracji, to zasięg wprowadzonych nazw rozciąga się do końca bloku instrukcji* ***for****.*

☞ *Uwaga 2. Wyrażenie1 musi być typu arytmetycznego lub wskaźnikowego.*

Instrukcja **for** jest równoważna następującej instrukcji **while**:

```
instrukcja-inicjująca
while (wyrażenie1)
{
 instrukcja
 wyrażenie2;  }
```

Ważnym elementem składni instrukcji **for** jest sposób zapisu instrukcji inicjującej oraz wyrażeń składowych, gdy mamy kilka zmiennych sterujących. W takich przypadkach przecinek pomiędzy wyrażeniami pełni rolę operatora. Np. w instrukcji

```
for ( ii = 1, jj = 2; ii < 5; ii++, jj++ )
  cout << "ii = " << ii << " jj = " << jj << "\n";
```

instrukcja inicjująca zawiera dwa wyrażenia: `ii = 1` oraz `jj = 2` połączone przecinkiem. Operator przecinkowy ',' wiąże te dwa wyrażenia w jedno wyrażenie, wymuszając wartościowanie wyrażeń od lewej do prawej. Tak więc najpierw `ii` zostaje zainicjowane do 1, a następnie `jj` zostanie zainicjowane do 2. Podobnie wyrażenie2, które składa się z dwóch wyrażeń `ii++` oraz `jj++`, połączonych operatorem przecinkowym; po każdym wykonaniu instrukcji `cout` najpierw zwiększa się o 1 `ii`, a następnie `jj`.

Przykład 3.9.

```
//Program Piramida
#include <iostream.h>
#include <iomanip.h>
int main()  {
    const int WIERSZ = 5;
    const int KOLUMNA = 15;
    for (int i = 1; i <= WIERSZ; i++)
     {
      cout << setw(KOLUMNA - i) << '*';
      for (int j = 1; j <= 2 * i -2; j++)
      cout << '*';
      cout << endl;
     }
    return 0;
}
```

Dyskusja. W przykładzie, podobnie jak dla instrukcji **while**, wykorzystano funkcję `setw()` z pliku `iomanip.h`. Identyczne są również definicje stałych symbolicznych `WIERSZ` i `KOLUMNA`. Taka sama jest również postać wydruku. Natomiast program jest nieco krótszy; wynika to stąd, że instrukcja **for** jest wygodniejsza od instrukcji **while** dla znanej z góry liczby obiegów pętli. Zauważmy też, że w pętli wewnętrznej umieszczono definicję zmiennej `j` typu **int**. Zasięg tej zmiennej jest ograniczony: można się nią posługiwać tylko od punktu definicji do końca bloku zawierającego wewnętrzną instrukcję **for**.

- *Uwaga 1. Syntaktycznie poprawny jest zapis for (; ;). Jest to zdegenerowana postać instrukcji **for**, równoważna for(;1;) lub while (1), czyli pętli nieskończonej; np. instrukcja for(;;) cout << "wiersz\n"; będzie drukować podany tekst aż do zatrzymania programu, lub wymuszenia instrukcją **break** zatrzymania pętli.*

- *Uwaga 2. W ciele instrukcji iteracyjnych używa się niekiedy instrukcji **continue;**. Wykonanie tej instrukcji przekazuje sterowanie do części testującej wartość wyrażenia w pętlach **while** i **do-while** (krok 1), lub do kroku 4 w instrukcji **for**.*

- *Uwaga 3. W języku C++ istnieje instrukcja skoku bezwarunkowego **goto** o składni goto etykieta:. Instrukcji tej nie omawiano, ponieważ jej używanie nie jest zalecane.*

3.

Typy pochodne

Użytkownik może w swoim programie deklarować i definiować typy pochodne od typów podstawowych: wskaźniki, referencje, tablice, struktury, unie oraz klasy, a także typy pochodne od tych struktur; może również definiować nowe operacje, wykonywane na tych strukturach danych.

Typ wskaźnikowy

Wskaźniki (ang. pointers) są tym mechanizmem, który uczynił język C i jego następcę – język C++ – tak silnym narzędziem programistycznym. W języku C++ dla każdego typu X istnieje skojarzony z nim typ wskaźnikowy X*. Zbiorem wartości typu X* są *wskaźniki* do obiektów typu X. Do zbioru wartości typu X* należy również wskaźnik pusty, oznaczany jako 0 lub NULL.
Wystąpieniem typu wskaźnikowego jest *zmienna wskaźnikowa*. Deklaracja zmiennej wskaźnikowej ma następującą postać:

nazwa-typu-wskazywanego* nazwa-zmiennej-wskaźnikowej;

Zgodnie z powyższymi określeniami, wartościami zmiennej wskaźnikowej mogą być wskaźniki do uprzednio zadeklarowanych obiektów (zmiennych, stałych) typu wskazywanego. Wartością zmiennej wskaźnikowej nie może być stała. Jeżeli żądamy, aby zmienna wskaźnikowa nie wskazywała na żaden obiekt programu, przypisujemy jej wskaźnik 0 (NULL), np.

```
int* wski;//Typ wskazywany:int.Typ wskaźnikowy:int*
wski = 0;
```

W deklaracji zmiennej wskaźnikowej typem wskazywanym może być dowolny typ wbudowany, typ pochodny od typu wbudowanego, lub typ zdefiniowany przez użytkownika. W szczególności może to być typ **void**, np.

```
void* wsk; //Typ wskazywany: void. Typ wsk: void*
```

używany wtedy, gdy typ wskazywanego obiektu nie jest dokładnie znany w chwili deklaracji, lub może się zmieniać w fazie wykonania. Zmiennej wsk typu **void*** możemy przypisać wskaźnik do obiektu dowolnego typu.

Podobnie jak zmienne typów wbudowanych, zmienne wskaźnikowe mogą być deklarowane w zasięgu globalnym lub lokalnym. Globalne zmienne wskaźnikowe są alokowane w pamięci statycznej programu, bez względu na to, czy są poprzedzone słowem kluczowym **static**, czy też nie. Jeżeli globalna zmienna wskaźnikowa nie jest jawnie zainicjowana, to kompilator przydziela jej niejawnie wartość 0 (NULL). Tak samo będzie inicjowana statyczna zmienna lokalna, tj. zmienna wskaźnikowa zadeklarowana w bloku funkcji, przy czym jej deklaracja jest poprzedzona słowem kluczowym **static**.

Wskaźniki i adresy

W ogólności wskaźnik zawiera informację o lokalizacji wskazywanej danej oraz informację o typie tej danej. Typową zatem jest implementacja wskaźników jako adresów pamięci; wartością zmiennej wskaźnikowej może być wtedy adres początkowy obiektu wskazywanego.

Weźmy pod uwagę następujący ciąg deklaracji:

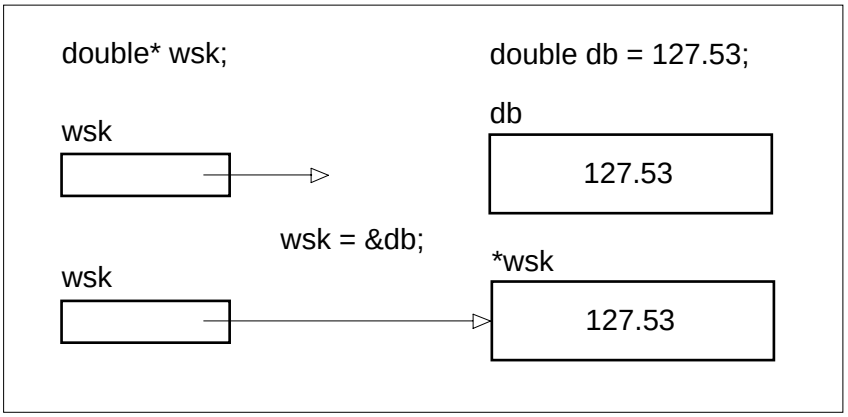
```
int i = 1, j = 10;
int* wski; // deklaracja zmiennej wski typu int*
wski = &i; // Teraz wski wskazuje na i. *wski==1
j = *wski; // Teraz j==1
```

Deklaracje te wprowadzają zainicjowane zmienne `i` oraz `j`, a następnie zmienną wskaźnikową `wski`, której następna instrukcja przypisuje wskaźnik do zmiennej `i`. Znamy *operator adresacji* "&" przyłożony do istniejącego obiektu (np. `&i`) daje wskaźnik do tego obiektu. Wskaźnik ten w instrukcji przypisania `wski= &i;` został przypisany zmiennej wskaźnikowej `wski`. Znamy *operator dostępu pośredniego* "*" (nazywany też operatorem wyłuskania) transformuje wskaźnik w wartość, na którą on wskazuje. Inaczej mówiąc, wyrażenie "*wski" oznacza zawartość zmiennej wskazywanej przez `wski`.
Zmienną wskaźnikową `wski` można też bezpośrednio zainicjować wskaźnikiem do zmiennej `i` w jej deklaracji:
int* wski = &i;

☞*Uwaga 1. Zarówno zmienne wskaźnikowe, jak i przyjmowane przez nie wartości przyjęto nazywać wskaźnikami; konwencję tę będziemy często stosować w dalszych partiach tekstu.*

☞*Uwaga 2. Ze względu na wykorzystywane w tej pracy implementacje języka C++, będziemy często utożsamiać wskaźnik do obiektu z adresem tego obiektu.*

Rysunek 4-1 ilustruje deklarację i przypisanie wskaźnikowi adresu istniejącej zmiennej.



Rys. 4-1 Zmienna wskaźnikowa i zmienna wskazywana

Wskaźnik może być również inicjowany innym wskaźnikiem tego samego typu, np.

```
int ii = 38;
int* wsk1 = &ii;
int* wsk2 = wsk1;
```

Natomiast deklaracja inicjująca o postaci:

```
int* wsk3 = &wsk1;
```

jest błędna, ponieważ `wsk3` jest wskaźnikiem do zmiennej typu **int**, podczas gdy wartością `&wsk1` jest adreswskaźnika do zmiennej typu **int**. Ostatnią deklarację można jednak zmienić na poprawną, pisząc:

```
int** wsk3 = &wsk1;
```

ponieważ `wsk3` jest teraz wskaźnikiem do wskaźnika do zmiennej typu **int**.

Jeżeli `int* wski` wskazuje na zmienną `i`, to `*wski` może wystąpić w każdym kontekście dopuszczalnym dla `i`. Np.

```
*wski = *wski + 3;
```
zwiększa `*wski` (czyli zawartość zmiennej `i`) o 3, a instrukcja przypisania:

```
j = *wski + 5;
```

pobierze zawartość spod adresu wskazywanego przez `wski` (tj. aktualną wartość zmiennej `i`), doda do niej 5 i przypisze wynik do `j` (wartość `*wski` pozostanie bez zmiany).
Powyższe instrukcje działają poprawnie, ponieważ `*wski` jest wyrażeniem, a operatory "*" i "&" wiążą silniej niż operatory arytmetyczne.
Podobnie instrukcja:

```
j = ++*wski;
```

zwiększy o 1 wartość `*wski` i przypisze tę zwiększoną wartość do `j`, zaś instrukcja:

```
j = (*wski)++;
```

przypisze bieżącą wartość `*wski` do `j`, po czym zwiększy o 1 wartość `*wski`. W ostatnim zapisie nawiasy są konieczne, ponieważ operatory jednoargumentowe, jak "*" i "++" wiążą od prawej do lewej. Bez nawiasów mielibyśmy zwiększenie o 1 wartości `wski` zamiast zwiększenia wartości `*wski`.

Zauważmy także, że skoro wartością wskaźnika jest adres obszaru pamięci przeznaczonego na zmienną danego typu, to wskaźniki różnych typów będą mieć taki sam rozmiar. Jest to oczywiste, ponieważ system adresacji komórek pamięci dla określonej platformy sprzętowej i określonego systemu operacyjnego jest zunifikowany i niezależny od interpretacji (wymuszonej typem) ciągu bitów zapisanego pod danym adresem.

Dodajmy na zakończenie kilka uwag dotyczących wskaźników do typu **void**. Zmiennej typu **void*** można przypisać wskaźnik dowolnego typu, ale nie odwrotnie. I tak np. poprawne są deklaracje:

```
void* wskv;
double db;
double* wskd = &db;
wskv = wskd; // konwersja niejawna do void*
```

ale nie można się odwołać do `*wskv`. Błędem byłaby też próba przypisania wskaźnika dowolnego typu do `wskv`, np.

```
wskd = wskv;.
```

Spróbujmy obecnie krótko podsumować nasze rozważania.

Zmienna wskaźnikowa (wskaźnik) jest szczególnym rodzajem zmiennej, przyjmującej wartości ze zbioru, którego elementami są adresy. Przypomnijmy, że "zwykła" zmienna jest obiektem o następujących atrybutach:

- Typ zmiennej;
- Nazwa zmiennej. Zmienna może mieć zero, jedną, lub kilka nazw. Nazwy nie muszą być prostymi identyfikatorami, jak np. a, b, znak; mogą one być również wyrażeniami, oznaczającymi zmienną, np. oznaczenie zmiennej indeksowanej tablicy tab może mieć postać tab[i + 5]. Wyrażenie oznaczające zmienną jest l-wartością, ponieważ tylko takie wyrażenie może się pojawić po lewej stronie operatora przypisania.
- Lokalizacja zmiennej, tj. adres w pamięci, pod którym są przechowywane jej wartości.

- Wartość, zapisana pod adresem, określonym lokalizacją zmiennej, nazywana niekiedy r-wartością.

Deklaracja wskaźnika niezainicjowanego przypisuje mu zadeklarowany typ, natomiast wskazywany przez niego adres i przechowywane pod tym adresem dane są nieokreślone. Jeżeli wskaźnik zainicjujemy w jego deklaracji lub przypiszemy mu adres wcześniej zadeklarowanej "zwykłej" zmiennej, to zostanie wyposażony we wszystkie wymienione wyżej atrybuty zmiennej symbolicznej.

Przykład 4.1.

```
// Wskazniki i adresy
#include <iostream.h>

int   main() {
      int i = 5, j = 10;
      int* wsk;
      wsk = &i;
      *wsk = 3; // ten sam efekt co i = 3;
      j = *wsk + 25; // j==28, *wsk==3
      wsk = &j; // *wsk==28
      i = j;  // i==28, &i bez zmiany
      j = i;  // j==28, &j bez zmiany
      cout << "*wsk= " << *wsk << endl
           << "i= "  << i  << endl;
      return 0;
 }
```

Dynamiczna alokacja pamięci

W środowisku programowym C++ każdy program otrzymuje do dyspozycji pewien obszar pamięci dla alokacji obiektów tworzonych w fazie wykonania. Obszar ten, nazywany *pamięcią swobodną* jest zorganizowany w postaci tzw. *kopca* lub *stogu*. Na kopcu (ang. heap) alokowane są obiekty dynamiczne, tj. takie, które tworzy się i niszczy przez zastosowanie operatorów **new** i **delete**. Operator **new** alokuje (przydziela) pamięć na kopcu, zaś operator **delete** zwalnia pamięć alokowaną wcześniej przez new. Uproszczona składnia instrukcji z operatorem new jest następująca:

```
wsk = new typ;
```

lub

```
wsk = new typ (wartość-inicjalna);
```

gdzie `wsk` jest wcześniej zadeklarowaną zmienną wskaźnikową, np. `int* wsk;`
Zmienną `wsk` można też bezpośrednio zainicjować adresem tworzonego dynamicznie obiektu:

```
typ* wsk = new typ;
```

lub

```
wsk = new typ(wartość inicjalna);
```

np.

```
int* wsk = new int(10);
```

Składnia operatora **delete** ma postać:

```
delete wsk;
```

gdzie `wsk` jest wskaźnikiem do typu o nazwie `typ`.

Zadaniem operatora **new** jest próba (nie zawsze pomyślna) utworzenia "obiektu" typu typ; jeżeli próba się powiedzie i zostanie przydzielona pamięć na nowy obiekt, to **new** zwraca wskaźnik do tego obiektu. Zauważmy, że alokowany dynamicznie obszar pamięci nie ma nazwy, która miałaby charakter l-wartości. Po udanej alokacji zastępczą rolę nazwy zmiennej pełni `*wsk`, zaś adres obszaru jest zawarty w `wsk`.
Operator **delete** niszczy obiekt utworzony przez operator **new** i zwraca zwolnioną pamięć do pamięci swobodnej. Po operacji **delete** wartość zmiennej wskaźnikowej staje się nieokreślona. Tym niemniej zmienna ta może nadal zawierać stary adres, lub, zależnie od implementacji, wskaźnik może zostać ustawiony na zero (lub NULL). Jeżeli przez nieuwagę skorzystamy z takiego wymazanego wskaźnika, który nadal zawiera stary adres, to prawdopodobnie program będzie nadal wykonywany, zanim dojdzie do miejsca, gdzie ujawni się błąd. Tego rodzaju błędy są na ogół trudne do wykrycia.

☞*Uwaga. Ponieważ alokowane dynamicznie zmienne istnieją aż do chwili zakończenia wykonania programu, brak instrukcji z operatorem delete spowoduje zbędną zajętość pamięci swobodnej. Natomiast powtórzenie instrukcji dealokacji do już zwolnionego obszaru pamięci swobodnej może spowodować niekontrolowane zachowanie się programu.*

Przykład 4.2.

```
#include <iostream.h>
int   main() {
      int* wsk; // wsk jest wskaznikiem do int;
      wsk = new int;
      delete wsk;
      wsk = new int (9); // *wsk == 9
      cout << "*wsk= " << *wsk << endl;
      delete wsk;
      return 0;
}
```

Zastosowanie operatora **delete** do pamięci niealokowanej prawie zawsze grozi nieokreślonym zachowaniem się programu podczas wykonania. Można temu zapobiec, uzależniając dealokację od powodzenia alokacji pamięci na stogu.

Podane niżej przykłady ilustrują kilka sposobów zabezpieczenia się przed niepowodzeniem alokacji pamięci za pomocą operatora **new**.

Przykład 4.3.

```
#include <iostream.h>
int   main()
      {
      int* wsk;
      if ((wsk = new int) == 0)
//alternatywny zapis: if (!(wsk = new int))
      { cout << "Nieudana alokacja\n"; return 1; }
      *wsk = 9;
      delete wsk;
      return 0;
}
```

Przykład 4.4.

```
#include <iostream.h>
int   main()
      {
      int* wsk = new int;
      if (wsk == 0)
//alternatywny zapis: if (!wsk)
      {
       cout << "Nieudana alokacja\n";
       return 1;
      }
      *wsk = 9;
      delete wsk;
      return 0;
}
```

Przykład 4.5.

```
#include <iostream.h>
#include <stdlib.h>
int   main()
      {
      int* wsk = new int;
      if (!wsk)
      {
       cout << "Nieudana alokacja\n";
       abort(); // lub exit(-1)
      }
      *wsk = 9;
      delete wsk;
      return 0;
}
```

Przykład 4.6.

```
#include <iostream.h>
#include < assert.h>
int   main() {
      int* wsk = new int (8);
      assert (wsk != 0);
      cout << "*wsk= " << *wsk << endl;
      delete wsk;
      wsk = 0; //lub NULL
      return 0;
}
```

Wskaźniki stałe

W C++ można także deklarować wskaźniki z tzw. *modyfikatorem const*. Tak więc wskaźnik może adresować *stałą symboliczną*:

```
const double cd = 3.50;
const double* wskd = &cd;
```

Przy zapisie jak wyżej, `wskd` jest wskaźnikiem do stałej symbolicznej `cd` typu **double**. Ponieważ stała symboliczna może być traktowana jako zmienna “tylko odczyt” (ang. read-only variable), zatem

jej wartość nie może być zmieniona ani bezpośrednio, ani pośrednio, tj. przez zmianę wartości `*wskd`. Natomiast wskaźnik `wskd` można zmieniać, przypisując mu adres zmiennej symbolicznej, np.

```
double d = 1.5;
wskd = &d;
```

Chociaż `d` nie jest stałą, powyższe przypisanie zapewnia nam, że wartość `d` nie może być modyfikowana poprzez `wskd`. Ilustruje to niżej podany przykład.

Przykład 4.7.

```
#include <iostream.h>
int   main() {
      const double cd = 1.50;
      const double* wskd = &cd;
      cout << "cd = " << cd << endl;
      cout << "*wskd = " << *wskd << endl;
      double d = 2.50;
      wskd = &d;
//    *wskd = 3.50; Niedopuszczalne
      cout << "*wskd = " << *wskd << endl;
      return 0;
}
```

Wydruk z programu ma postać:

cd = 1.5
***wskd = 1.5**
***wskd = 2.5**

Można też zadeklarować *wskaźnik stały*. Np.

```
int i = 10;
int* const wski = &i;
```

Tutaj `wski` jest wskaźnikiem stałym do zmiennej `i` typu **int**. Przy takiej deklaracji można zmieniać wartość zmiennej `i` zmieniając wartość `*wski`, ale nie można zmienić adresu wskazywanego przez `wski`.
Dopuszczalne jest również zadeklarowanie wskaźnika stałego do stałej symbolicznej, np.

```
const int ci = 7;
const int* const wskci = &ci;
```

W tym przypadku błędem syntaktycznym byłaby zarówno próba zmiany wartości `*wskci` jak i adresu wskazywanego przez `wskci`. Omawiane deklaracje ilustruje kolejny przykład.

Przykład 4.8.

```
#include <iostream.h>
int   main() {
      int i = 10;
      int* const wski = &i;
      *wski = 20;
      cout << "*wski = " << *wski << endl;
//    wski++; Niedopuszczalne
      const int ci = 7;
      const int* const wskci = &ci;
      cout << "*wskci = " << *wskci << endl;
      return 0;
}
```

Typ tablicowy

Tablica jest strukturą danych, złożoną z określonej liczby elementów tego samego typu. Elementy tablicy mogą być typu **int**, **double**, **float**, **short int**, **long int**, **char**, **wskaźnikowego**, **struct**, **union**; mogą być również tablicami oraz obiektami klas.
Deklaracja tablicy składa się ze specyfikatora (określnika) typu, identyfikatora i wymiaru. Wymiar tablicy, który określa liczbę elementów zawartych w tablicy, jest ujęty w parę nawiasów prostokątnych "[]" i musi być większy lub równy jedności. Jego wartość musi być wyrażeniem stałym typu całkowitego, możliwym do obliczenia w fazie kompilacji; oznacza to, że nie wolno używać zmiennej dla określenia wymiaru tablicy. Parę nawiasów "[]", poprzedzonych nazwą tablicy, np. "Arr[]", nazywa się *deklaratorem tablicy*.
Elementy tablicy są dostępne poprzez obliczenie ich położenia w tablicy. Taka postać dostępu jest nazywana *indeksowaniem*. Np. zapis:

```
double tabd[4];
```

deklaruje tablicę 4 elementów typu double: tabd[0], tabd[1], tabd[2] i tabd[3], gdzie tabd[i] są nazywane *zmiennymi indeksowanymi*. Inaczej mówiąc, zmienne tabd[0], ..., tabd[3] będą sekwencją czterech kolejnych komórek pamięci, w każdej z których można umieścić wartość typu **double**.

Inicjowanie tablic

Deklarację tablicy można powiązać z nadaniem wartości inicjalnych jej elementom. Tablicę można zainicjować na kilka sposobów:

- Umieszczając deklarację tablicy na zewnątrz wszystkich funkcji programu. Taka tablica globalna (ewentualnie poprzedzona słowem kluczowym **static**) zostanie zainicjowana automatycznie przez kompilator. Np. każdy element tablicy globalnej o deklaracji double tabd[3]; zostanie zainicjowany na wartość 0.0.
- Deklarując tablicę ze słowem kluczowym **static** w bloku funkcji. Jeżeli nie podamy przy tym wartości inicjalnych, to uczyni to, jak w poprzednim przypadku, kompilator.
- Definiując tablicę, tj. umieszczając po jej deklaracji wartości inicjalne, poprzedzone znakiem "=". Wartości inicjalne, oddzielone przecinkami, umieszcza się w nawiasach klamrowych po znaku "=". Np. tablicę tabd[4] można zainicjować instrukcją deklaracji:
  ```
  double tabd[4] = { 1.5, 6.2, 2.8, 3.7 };
  ```
  W deklaracji inicjującej można pominąć wymiar tablicy:
  ```
  double tabd[] = { 1.5, 6.2, 2.8, 3.7 };
  ```
  W tym przypadku kompilator obliczy wymiar tablicy na podstawie liczby elementów inicjalnych.
- Deklarując tablicę, a następnie przypisując jej elementom pewne wartości w kolejnych instrukcjach przypisania, np. umieszczonych w pętli. Zwróćmy w tym miejscu uwagę na różnicę pomiędzy

inicjowaniem tablicy w jej definicji, a przypisaniem, które może mieć miejsce po uprzednim zadeklarowaniu tablicy (chociaż w obu przypadkach używany jest ten sam symbol "=").

Podobnie jak stałe typów wbudowanych, można zdefiniować tablicę o elementach stałych, np.

```
const int tab[4] = { 10, 20, 30, 40 };
```

lub

```
const int tab[] = { 10, 20, 30, 40 };
```

W takim przypadku zmienne indeksowane stają się stałymi symbolicznymi, których wartości nie można zmieniać żadnymi instrukcjami przypisania.

☞*Uwaga. Tablica nie może być inicjowana inną tablicą ani też tablica nie może być przypisana do innej tablicy.*

Jeżeli w definicji tablicy podaje się wartości inicjalne składowych, to kompilator inicjuje tablicę według rosnących adresów jej elementów. W przypadku gdy liczba wartości inicjalnych jest mniejsza od maksymalnego indeksu, podanego w deklaratorze ([]) tablicy, zostaną zainicjowane początkowe składowe, a pozostałe kompilator zainicjuje na 0. Błędem syntaktycznym jest podanie większej od wymiaru tablicy liczby wartości inicjalnych.

Przykład 4.9.

```
// Tablica liczb typu int.
#include <iostream.h>

void main() {
     int i;
     const int WYMIAR = 10;
     int tab[WYMIAR];
/*   tab[WYMIAR] - tablica 10 liczb calkowitych:
     tab[0],tab[1],..., tab[9]  */
     for (i = 0; i < WYMIAR; i++)
     {
      tab[i] = i;
      cout << "tab[" << i << "]= " << tab[i] << endl;
     }
}
```

Dyskusja. Deklaracja `const WYMIAR = 10;` definiuje identyfikator `WYMIAR`, który w programie będzie zastępowany stałą 10. Zmienną `i` typu **int** zadeklarowano na użytek pętli **for**; zmienną tę można było także zadeklarować bezpośrednio w pętli for:

```
     for(int i = 0; ... )
```

Nadawanie elementom tablicy `tab[]` wartości równych ich indeksom można było zastąpić zainicjowaniem tablicy w chwili jej deklaracji:

```
     int tab[WYMIAR] = { 0,1,2,3,4,5,6,7,8,9 };
```

lub

```
     int tab[] = { 0,1,2,3,4,5,6,7,8,9 };
```

Przykład 4.10.

```
// Tablica znakow
#include <iostream.h>
void main() {
      const int BUF = 10;
      char a[BUF] = "ABCDEFGHIJ";
// a[BUF]-tablica 10 znakow: a[0],a[1],...,a[9]
/* Dopuszczalne jest zainicjowanie: char a[BUF] =
   {'A','B','C','D','E','F','G','H','I','J'};
*/
      cout << a<< endl;
}
```

Przykład 4.11.

```
// Tablica znakow, instrukcja for
#include <iostream.h>
void main() {
      const int BUF = 10;
      char znak;
      char a[BUF] =
      {'A','B','C','D','E','F','G','H','I','J'};
  // a[BUF]-tablica 10 znakow: a[0],a[1],...,a[9]
      for (znak = 'A'; znak <= 'J'; znak++)
      cout << a[znak - 65] << endl;
}
```

Wskaźniki i tablice

W języku C++ istnieje ścisła zależność pomiędzy wskaźnikami i tablicami. Zależność ta jest ta silna, że każda operacja na zmiennej indeksowanej może być także wykonana za pomocą wskaźników. Przy tym operacje z użyciem wskaźników są w ogólności wykonywane szybciej.
Podczas kompilacji nazwa tablicy, np. `tab`, jest automatycznie przekształcana na wskaźnik do pierwszego jej elementu, czyli na adres `tab[0]`. We fragmencie programu:

```
int tab[4] = { 10, 20, 30, 40 };
int* wsk;
wsk = tab;
```

instrukcja:

```
wsk = tab;
```

jest równoważna instrukcji:

```
wsk = &tab[0];
```

ponieważ `tab == &a[0]`. Inaczej mówiąc, `wsk` oraz `tab` wskazują teraz na element początkowy `tab[0]` tablicy `tab[]`. Istnieje jednak istotna różnica pomiędzy `wsk` i `tab`. Identyfikator tablicy ("tab") jest inicjowany adresem jej pierwszego elementu ("tab[0]"); adres ten nie może ulec zmianie w programie (nazwa tablicy nie jest modyfikowalną l-wartością). Tak więc identyfikator tablicy jest równoważny *wskaźnikowi stałemu* i nie można go zwiększać czy zmniejszać. natomiast `wsk` jest *zmienną wskaźnikową*; w programie musimy najpierw ustawić wskaźnik na adres wcześniej alokowanego obiektu (np. tablicy `tab`), a następnie możemy zmieniać jego wskazania. Zwróćmy

uwagę na to, że zwiększenie wartości wskaźnika `wsk` np. o 2 zwiększy w tym przypadku wskazywany adres o tyle bajtów, ile zajmują dwa elementy tablicy `tab`. Tak więc instrukcje:

```
tab = tab+1;
tab = wsk;
```

są błędne, natomiast instrukcja:

```
wsk = wsk + 1;
```

jest poprawna (jest ona równoważna instrukcji: `wsk = &tab[1];`, zatem nowa wartość `*wsk == tab[1]`).

Przykład 4.12.

```
// Wskazniki i tablice
#include <iostream.h>
void main() {
     int t1[10] = { 0,1,2,3,4,5,6,7,8,9 };
     int t2[10], *wt1, *wt2;
     wt1 = t1;
// to samo, co wt1 = &t1[0], poniewaz t1==&t1[0]
     wt2 = t2;
// to samo, co wt2 = &t2[0], poniewaz t2==&t2[0]
     for (int i = 0; i < 10; i++)
     cout << "t1[" << i << "]= " << *wt1++ << endl;
}
```

Dyskusja. W bloku funkcji main mamy kolejno:
- Deklarację tablicy `t1[10]` typu **int**, zainicjowanej wartościami 0..9, równymi indeksom tablicy. Przypomnijmy, że podanie wymiaru tablicy nie było w tym przypadku konieczne.
- Deklarację tablicy t2[10] typu int oraz dwóch wskaźników: wt1 i wt2, wskazujących na typ int.
- Dwie kolejne instrukcje przypisania: wt1 = t1; wt2 = t2;. Przedyskutujmy je nieco bardziej szczegółowo.

Bezpośrednio po przypisaniu `wt1 = 1;` zawartość `*wt1` jest równa `t1[0]`. Zawartość pod adresem `wt1+1`, czyli `*(wt1+1)` jest równa `t1[1]`, itd. Wyjaśnia to użycie wyrażenia `*wt1++` w pętli for: zamiast `cout << *wt1++` moglibyśmy napisać `cout << *(wt1+i)` lub `cout << wt1[i]`. Ten ostatni zapis jest poprawny, ponieważ `wt1[i]` jest równoważne `*(wt1+i)`.
Zapis `*wt1++` daje kolejne wartości `t1[i]` dzięki hierarchii operatorów: operator "++" wiąże silniej niż operator "*" i w rezultacie otrzymujemy `*wt1`, `*(wt1+1)`, `*(wt1+2)`, ... , `*(wt1+9)`, czyli zawartości `t1[0]`, `t1[1]`, ... , `t1[9]`.

Przykład 4.13.

```
// Wskazniki i tablice
void main() {
     int t1[10], *wt1, *wt2;
     int t2[10] = {0,1,2,3,4,5,6,7,8,9};
     wt1 = t1;
     wt2 = t2;
     while (wt2 <= &t2[9])
     *wt1++ = *wt2++ ;
}
```

Dyskusja. Przykład ilustruje kopiowanie tablicy `t2` do tablicy `t1`. Zauważmy, że zamiast instrukcji `wt1 = t1;` moglibyśmy napisać: `wt1 = &t1[0];`, ponieważ `t1==&t1[0]`. To samo dotyczy wskaźnika `wt2` i tablicy `t2`. Kopiowanie jest wykonywane w pętli **while** na wskaźnikach. W instrukcji przypisania `*wt1++ = *wt2++;` można byłoby umieścić nawiasy dla lepszego pokazania, iż najpierw jest zwiększany wskaźnik, a następnie stosowany operator dostępu pośredniego "*", tj. napisać tę instrukcję w postaci:

```
*(wt1++) = *(wt2++);
```

Nie jest to jednak konieczne ponieważ przyrostkowy operator "++" wiąże od prawej do lewej i ma wyższy priorytet niż "*". Omawiana instrukcja jest idiomem języka C++. Jest ona równoważna sekwencji instrukcji:

```
*wt1 = *wt2;
wt1 = wt1 + 1;
wt2 = wt2 + 1;
```

☞*Uwaga. Zapamiętajmy: int* wsk[10] deklaruje tablicę 10 wskaźników do typu int, natomiast int (*wsk)[10] deklaruje wskaźnik wsk do tablicy o 10 elementach typu int. Nawiasy są tutaj konieczne, ponieważ deklarator tablicy [] ma wyższy priorytet niż *.*

Wskaźniki i łańcuchy znaków

W języku C++ wszystkie operacje na łańcuchach znaków są wykonywane za pomocą wskaźników do znaków łańcucha. Przypomnijmy, że łańcuch jest ciągiem znaków, zakończonym znakiem `'\0'`. Jeżeli zadeklarujemy tablicę znaków `alfa1[]` z wartością inicjalną "ABCD"

```
char alfa1[] = "ABCD";
```

to kompilator doda do łańcucha "ABCD" terminalny znak zerowy i obliczy, że wymiar tablicy `alfa1[]` wynosi 5. wynika stąd, że deklaracja:

```
char alfa1[4] = "ABCD";
```

jest błędna , ponieważ nie przewiduje miejsca na kończący zapis znak `'\0'`.
Błędną będzie także próba przypisania łańcucha znaków do niezainicjowanej tablicy typu **char**:

```
char alfa2[5];
alfa2 = "ABCD"; //niedopuszczalne!
```

ponieważ w języku nie zdefiniowano operacji przypisania dla tablicy jako całości.
Formalnie dopuszczalnym jest zainicjowanie tablicy znaków w sposób analogiczny do innych typów, np.

```
char alfa3[] = { 1, 2, 3, 4 };
char alfa4[] = { 'A', 'B', 'C', 'D' }
```

Tak zainicjowane tablice `alfa3[]` oraz `alfa4[]` są tablicami 4 a nie 5 znaków i powyższe zapisy są oczywistym błędem programistycznym. Oczywistym dlatego, ponieważ zdecydowana większość funkcji bibliotecznych operujących na łańcuchach znaków zakłada istnienie terminalnego znaku zerowego. Można, rzecz jasna, napisać deklarację

```
char alfa4[] = { 'A', 'B', 'C', 'D', '\0' };
```

równoważną innej poprawnej deklaracji

```
char alfa4[] = "ABCD";
```

ale, przyznajmy, jest to raczej kłopotliwe.

Wyjściem z tych kłopotów jest pokazany już wcześniej mechanizm automatycznej konwersji nazwy tablicy na wskaźnik do jej pierwszego elementu. Zatem wykonanie instrukcji deklaracji

```
char alfa[] = "ABCD";
```

spowoduje przydzielenie przez kompilator wskaźnika do `alfa[0]`, tj do znaku 'A'. Możemy więc zdefiniować wskaźnik do typu **char**, np.

```
char* wsk;
```

i przypisać mu nazwę tablicy `alfa[]`

```
wsk = alfa;
```

co jest równoważne sekwencji instrukcji:

```
char* wsk;
wsk = &alfa[0];
```

lub krótszemu zapisowi:

```
char* wsk = &alfa[0];
```

Z arytmetyki wskaźników wynika, że jeżeli `wsk==&alfa[0]`, to `wsk+i== &alfa[i]`. Pamiętając o przypisaniu `wsk = alfa;` mamy równoważność `wsk + i == &wsk[i]`. Wobec tego znak, zapisany pod adresem `&wsk[i]` (lub `&alfa[i]`), znajdziemy przez odwrócenie ostatniej relacji:

```
*(wsk + i) == wsk[i],
```

np.

```
*(wsk + 2) == 'C'
```

ponieważ `wsk[i] == alfa[i]`.

Konkluzja z tej nieco rozwlekłej dyskusji jest prosta: wszędzie tam, gdzie mamy zamiar wykonywać operacje na znakach, zamiast deklarować tablicę
`char alfa[] = "ABCD";`
wystarczy zadeklarować wskaźnik
`char* alfa = "ABCD";`

Pokazane niżej trzy przykłady ilustrują zastosowanie wskaźników do tablic (łańcuchów) znaków.

Przykład 4.14.

```
// Kopiowanie ze "start" do "cel"
// Wersja ze zmiennymi indeksowanymi
#include <iostream.h>
int  main() {
     char *start = "ABCD";
     char *cel = "EFGH";
     int i = 0;
     while ((cel[i] = start[i]) != '\0')  i++;
     cout << "Lancuch cel: " << cel << endl ;
     return 0;
}
```

Dyskusja. Wskaźnik `start` został ustawiony na adres pierwszego znaku łańcucha "ABCD", tj. `*start == start[0] == 'A'`. Analogicznie `*cel == cel[0] == 'E'`. Kolejne składowe łańcucha "ABCD" są kopiowane do łańcucha "EFGH" w pętli **while**. Ponieważ operacje te są prowadzone na zmiennych indeksowanych, adresy wskazywane przez `start` i `cel` nie ulegają zmianie. Można się o tym przekonać, deklarując je jako wskaźniki stałe:

```
char* const start = "ABCD";
char* const cel   = "EFGH";
```

Przykład 4.15.

```
// Kopiowanie ze "start" do "cel"
// Wersja ze wskaznikami
#include <iostream.h>
int  main() {
     char *start = "ABCD";
     char *cel = "EFGH";
     char *pomoc = cel;
     while ((*cel = *start) != '\0')
     {
      cel++ ;
      start++ ;
     }
     cout << "Lancuch pomoc: " << pomoc << endl ;
     return 0;
}
```

Dyskusja. W bloku instrukcji **while** wskaźniki `start` i `cel` są przesuwane aż do znaku `'\0'` kończącego łańcuch. Składowe łańcuchów są kopiowane przypisaniem `*cel = *start`, co jest równoważne `cel[0]=start[0]`. Po wyjściu z pętli wskaźnik `cel` będzie ustawiony na ostatniej składowej skopiowanego łańcucha, tj. na `'\0'`. Dla wydrukowania zawartości tego łańcucha musielibyśmy najpierw "cofnąć" wskaźnik `cel` o liczbę elementów łańcucha

```
cel = cel - 4;
```

i dopiero potem wydrukować łańcuch, pisząc np.

```
cout << "cel = " << cel << endl;
```

W programie przyjęto inne rozwiązanie. Wskaźnik `pomoc` jest na stałe ustawiony na adres pierwszego elementu łańcucha `cel` i wydruk kopii nie wymaga żadnych dodatkowych operacji. Zatem, podobnie jak w poprzednim przykładzie, wskaźnik `pomoc` można zadeklarować jako wskaźnik stały

```
char* const pomoc = cel;
```

Przykład 4.16.

```
// Kopiowanie ze "start" do "cel"
// Inna wersja ze wskaznikami
#include <iostream.h>
int main() {
      char *start = "ABCD";
      char *cel = "EFGH";
      char *pomoc = cel;
      while (*cel++ = *start++)
      ;
      cout << "Lancuch pomoc: " << pomoc << endl ;
      return 0;
}
```

Dyskusja. W tym przykładzie wykorzystano wzmiankowany już wcześniej idiom języka C++. Porównując ten program z poprzednim zauważymy, że jedyna różnica występuje w konstrukcji pętli **while**. Ponieważ instrukcja przypisania `*cel++ = *start++;` jest równoważna sekwencji instrukcji:

```
*cel = *start;
cel++;
start++;
```

to przypisanie `*cel = *start` można wykorzystać w wyrażeniu testowanym w pętli, co też uczyniono w poprzednim przykładzie.
W stosunku do poprzedniego przykładu występuje tutaj jeszcze jedna różnica. Ze względu na to, że zwiększanie wskaźników o 1 następuje na wejściu do pętli, końcowy adres wskazywany przez wskaźnik `cel` będzie teraz przesunięty w stosunku do początkowego o 5 jednostek (a nie o 4, jak poprzednio). Tak więc tym razem można by wydrukować łańcuch skopiowany, cofając najpierw wskaźnik `cel` o 5 jednostek:
cel = cel - 5;

☞*Dla kopiowania łańcuchów można wykorzystać funkcję biblioteczną strcpy(char* do, const char* z) z pliku nagłówkowego <string.h>, dostarczanego standardowo z każdym kompilatorem języka C++. Należy wówczas w programie umieścić dyrektywę #include <string.h>, a w bloku main() wywołać wspomnianą funkcję, pisząc: strcpy(cel, start);.*

Tablice wielowymiarowe

Ponieważ elementy tablicy mogą być tablicami, możliwe jest deklarowanie tablic wielowymiarowych. Np. zapis:

```
int tab[4][3];
```

deklaruje tablicę o czterech wierszach i trzech kolumnach. W tym przypadku dopuszcza się notacje: `tab` – tablica 12-elementowa, `tab[i]` – tablica 3-elementowa, w której każdy element jest tablicą 4-elementową, `tab[i][j]` – element typu **int**. Inaczej mówiąc, tablica `tab[4][3]` jest tablicą złożoną z czterech elementów `tab[0]`, `tab[1]`, `tab[2]` i `tab[3]`, przy czym każdy z tych czterech elementów jest tablicą trójelementową liczb całkowitych. Podobnie jak dla tablic jednowymiarowych, identyfikator `tab` jest niejawnie przekształcany we wskaźnik do pierwszego elementu tablicy, czyli do pierwszej spośród czterech tablic trójelementowych. Proces dostępu do elementów tablicy przebiega następująco:

Jeżeli mamy wyrażenie `tab[i]`, które jest równoważne `*(tab+i)`, to `tab` jest najpierw przekształcane do wymienionego wskaźnika; następnie `(tab+i)` zostaje przekształcone do typu `tab`, co obejmuje mnożenie i przez długość elementu na który wskazuje wskaźnik, tj. przez trzy elementy typu **int**. Otrzymany wynik jest dodawany do tab, po czym zostaje przyłożony operator dostępu pośredniego "*", dając w wyniku tablicę (trzy liczby całkowite), która z kolei zostaje przekształcona we wskaźnik do jej pierwszego elementu, tj. `tab[i][0]`.
Wynikają stąd dwa wnioski:

1) Tablice wielowymiarowe są w języku C++ zapamiętywane wierszami (ostatni z prawej indeks zmienia się najszybciej).
2) Pierwszy z lewej wymiar w deklaracji tablicy pomaga wyznaczyć wielkość pamięci zajmowanej przez tablicę, ale nie odgrywa żadnej roli w obliczaniu indeksów.

Tablice wielowymiarowe inicjuje się podobnie, jak jednowymiarowe, zamykając wartości inicjalne w nawiasy klamrowe. Jednak ze względu na zapamiętywanie wierszami przewidziano dodatkowe, zagnieżdżone nawiasy klamrowe, w których umieszcza się wartości inicjalne dla kolejnych wierszy.

Przykład 4.17.

```
#include <iostream.h>
int  main() {
     int tab[4][2] = // W [4] mozna opuscic 4
     {
      { 1, 2 }, // inicjuje tab[0],
     // tj. tab[0][0] i tab[0][1]
      { 3, 4 }, // inicjuje tab[1]
      { 5, 6 }, // inicjuje tab[2]
      { 7, 8 }  // inicjuje tab[3]
     };
     for (int i = 0; i < 4; i++)
     {
      cout<<"tab["<<i<<"][0]: "<<tab[i][0]<<'\t';
      cout<<"tab["<<i<<"][1]: "<<tab[i][1]<< '\n';
     }
     return 0;
}
```

tab[w-wiersze] [k-kolumny] => rozmiar (w x k)

Dyskusja. W programie zadeklarowano tablicę (macierz) o 4 wierszach i 2 kolumnach. Ponieważ tablica jest inicjowana jawnie, w jej deklaracji można byłoby opuścić podawanie pierwszego wymiaru. Nie są również konieczne zagnieżdżone nawiasy klamrowe, zawierające wartości inicjalne dla kolejnych wierszy – umieszczono je tutaj dla pokazania, że tablice wielowymiarowe są zapamiętywane wierszami. Widać to wyraźnie na wydruku:

tab[0][0]: 1 tab[0][1]: 2
tab[1][0]: 3 tab[1][1]: 4
tab[2][0]: 5 tab[2][1]: 6
tab[3][0]: 7 tab[3][1]: 8

Wydruk byłby identyczny, gdyby definicja tablicy miała postać:
`int tab[4][2] = { 1, 2, 3, 4, 5, 6, 7, 8 };`
Podobnie jak dla tablic jednowymiarowych można podać mniejszą od stopnia tablicy (tj. iloczynu wszystkich jej wymiarów) liczbę wartości inicjalnych. Wówczas ta część elementów tablicy, dla której zabraknie wartości inicjalnych, zostanie zainicjowana zerami. Tę własność należy również mieć na

uwadze przy opuszczaniu zagnieżdżonych nawiasów klamrowych. Np. wykonanie instrukcji deklaracji

```
int tab[4][2] =  { 1, 2, 3, 4, 5, 6 };
```

lub

```
int tab[4][2] = {  { 1, 2 },  { 3, 4 },  { 5, 6 }  };
```

Nada wartości inicjalne pierwszym trzem wierszom i spowoduje wyzerowanie czwartego wiersza. Natomiast wykonanie instrukcji deklaracji

```
int tab[4][2] = {  { 1 }, { 2 }, { 3 }, { 4 }  };
```

spowoduje wyzerowanie drugiej kolumny macierzy `tab[4][2]`.

Dynamiczna alokacja tablic

Operatory **new** i **delete** można również stosować do dynamicznej alokacji i dealokacji tablic w pamięci swobodnej programu, tj. na kopcu. Składnia instrukcji powołującej do życia tablicę dynamiczną jest następująca:

```
wsk = new typ[wymiar];
```

gdzie `wsk` jest wskaźnikiem do typu o nazwie `typ`, zaś wymiar określa liczbę elementów tablicy. Jeżeli alokacja się powiedzie, to **new** zwraca wskaźnik do pierwszego elementu tablicy. W przeciwieństwie do statycznej alokacji tablic z jednoczesnym inicjowaniem, jak w przykładowej deklaracji

```
int tab[10] = { 0,1,2,3,4,5,6,7,8,9 };
```

tablica alokowana dynamicznie nie może być inicjowana. Przykładowa instrukcja dynamicznej alokacji tablicy może mieć postać:

```
int *wsk = new int[150];
```

Powyższa instrukcja definiuje wskaźnik wsk do typu **int** i ustawia go na pierwszy element tworzonej tablicy 150 elementów typu **int**. W rezultacie `wsk[0]` będzie wartością pierwszego elementu tablicy, `wsk[1]` drugiego, etc.

Składnia operatora **delete** dla uprzednio alokowanej (z sukcesem!) tablicy dynamicznej ma postać:

```
delete [] wsk;
```

☞*Uwaga 1. Niektóre starsze wersje kompilatorów języka C++ wymagają napisania instrukcji usuwającej tablicę z pamięci w postaci delete wsk; lub podania wymiaru tablicy, np. delete [150] wsk;.*

☞*Uwaga 2. Ponieważ alokowane dynamicznie tablice istnieją aż do chwili zakończenia wykonania programu, brak instrukcji z operatorem delete spowoduje zbędną zajętość pamięci swobodnej. Natomiast powtórzenie instrukcji dealokacji do już zwolnionego obszaru pamięci swobodnej może spowodować niekontrolowane zachowanie się programu.*

Przykład 4.18.

```
// Dynamiczna alokacja tablicy
#include <iostream.h>
int   main() {
      int* wsk;
      wsk = new int[5];
      if (!wsk)  {
      cout << "Nieudana alokacja\n";
      return 1;  }
      for (int i = 0; i < 5; i++)
      wsk[i] = i;
      for(int j = 0; j < 5; j++) {
      cout << "wsk[" << j << "]: ";
      cout << wsk[j] << endl;    }
      delete [] wsk;
      return 0;
}
```

Analiza programu. Po wykonaniu instrukcji deklaracji `int* wsk;` wskaźnik `wsk` zostanie zainicjowany przypadkowym adresem; oczywiście wartość `*wsk` będzie również przypadkowa. Natomiast sama zmienna `wsk` otrzyma konkretny adres podczas kompilacji. Po wykonaniu instrukcji `wsk = new int[5];` wartość `wsk` będzie już konkretnym adresem pierwszego elementu tablicy. Oczywiście `&wsk`, tj. miejsce na stosie programu, gdzie ulokowany jest adres samego `wsk`, nie ulegnie zmianie. Ponieważ nie jest dozwolone inicjowanie tablicy w chwili powołania jej do życia, uczyniliśmy to w pierwszej pętli **for**. Po wyjściu z pętli otrzymamy wartości:
`wsk[0]==0`, `wsk[1]==1`, `wsk[2]==2`, `wsk[3]==3` i `wsk[4]==4`. Adres wskazywany przez `wsk`, tj. wartość samego `wsk`, będzie równy `&wsk[0]`. Następne adresy, tj. `&wsk[1]`, `&wsk[2]`, `&wsk[3]` i `&wsk[4]`, będą odległe od siebie o tyle bajtów, ile wynosi `sizeof(int)`.

Referencje

Referencja jest specjalnym, niejawnym wskaźnikiem, który działa jak alternatywna nazwa dla zmiennej. Głównym zastosowaniem referencji jest użycie ich w charakterze argumentów i wartości zwracanych przez funkcje. Tym niemniej można również deklarować i używać w programie referencje niezależne. Tę właśnie możliwość wykorzystamy do omówienia własności referencji.
Zmienną o nazwie podanej po nazwie typu z przyrostkiem **&**, np. `T& nazwa`, gdzie `T` jest nazwą typu, nazywamy *referencją do typu T*.
Referencja musi być zainicjowana do pewnej l-wartości, po czym może być używana jako inna nazwa dla l-wartości zmiennej. Ponieważ referencja spełnia rolę innej nazwy dla zmiennej, zatem jest ona także l-wartością. Przykładowe deklaracje:

```
int n = 1;
int& rn = n;
rn = 2; // Ten sam efekt co n = 2;
int *wsk = &rn;
int& rr = rn;
```

Pierwsza deklaracja, `int n = 1;` definiuje zmienną `n` typu **int** oraz inicjuje jej wartość na 1.
Druga deklaracja, `int& rn = n;` definiuje zmienną referencyjną `rn`, inicjując ją do wartości będącej adresem zmiennej `n`. Tak więc, skoro `rn` oraz `n` mają ten sam adres, to `rn` jest dodatkową, alternatywną nazwą dla tego obszaru pamięci, któremu wcześniej nadano symboliczną nazwę `n`.
Widać to wyraźnie w trzeciej instrukcji `rn = 2;`, która może mieć równoważną postać `n = 2;`. Zauważmy przy tym, że pierwsze dwie instrukcje deklaracji używają znaku równości do zainicjowania deklarowanych wielkości; natomiast w instrukcji `rn = 2;` znak równości jest operatorem przypisania.

Czwarta instrukcja deklaracji
`int *wsk = &rn;`
definiuje wskaźnik `wsk`, inicjując go adresem zmiennej `rn`. Zatem `wsk` będzie wskazywał na adres `n` (ten sam adres ma `rn`), zaś `*wsk` będzie tą samą wartością, którą została zainicjowana zmienna `n`.
Ostatnia instrukcja deklaracji
`int& rr = rn;`
inicjuje zmienną `rr` adresem zmiennej `rn`. Ponieważ adres zmiennej `rn` jest identyczny z adresem zmiennej `n`, zatem `rr` jest kolejną (czwartą) zastępczą nazwą dla tej samej komórki pamięci (`n`, `rn`, `*wsk` i `rr`).
Referencje do zmiennych danego typu `T` mogą być inicjowane jedynie l-wartościami typu `T`. Natomiast referencje do stałych symbolicznych typu `T` mogą być inicjowane l-wartościami typu `T`, l-wartościami sprowadzalnymi do typu `T`, lub nawet r-wartościami, np. stałą 1024. W takich przypadkach kompilator realizuje następujący algorytm:

1. Wykonaj, jeżeli to konieczne, konwersję typu.
2. Uzyskaną wartość wynikową umieść w zmiennej tymczasowej.
3. Wykorzystaj adres zmiennej tymczasowej jako wartość inicjalną.

Dla ilustracji weźmy deklarację

```
const double& refcd = 10;
```

dla której wskaźnikowa interpretacja algorytmu jest następująca:

```
double *refcdp; // referencja jako wskaznik
double temp;
temp = double(10);
refcdp = &temp;
```

gdzie `temp` reprezentuje zmienną tymczasową generowaną przez kompilator dla wykonania konwersji z **int** do **double**. Czas życia utworzonej w ten sposób zmiennej tymczasowej jest określony przez jej zasięg (np. do wyjścia z bloku, pliku lub programu). Implementacja tego typu konwersji musi zapewnić istnienie zmiennej tymczasowej dotąd, dopóki istnieje związana z nią referencja. Kompilator musi także zapewnić usunięcie zmiennej tymczasowej z pamięci, gdy nie jest już potrzebna.
Ze sposobu tworzenia niezależnej referencji wynika, że po zainicjowaniu *nie można zmienić obiektu*, z którym została związana. Ponieważ referencje nie tworzą "prawdziwych" obiektów w sensie używanym w języku C++, nie istnieją tablice referencji. Nie zdefiniowano również dla referencji operatorów, podobnych do używanych dla wskaźników.
Powyższe względy zdecydowały o niewielkiej użyteczności niezależnych referencji. Natomiast zastosowanie referencji jako parametrów formalnych i wartości zwracanych dla funkcji jest w wielu przypadkach wygodne i zalecane.

Struktury

Struktura jest jednostką syntaktyczną grupującą składowe różnych typów, zarówno podstawowych, jak i pochodnych. Ponadto składowa struktury może być tzw. polem bitowym. Struktury są deklarowane ze słowem kluczowym **struct**. Deklaracja referencyjna struktury ma postać:

```
struct nazwa;
```

zaś jej definicja

```
struct nazwa { /*...*/ };
```

gdzie `nazwa` jest nazwą nowo zdefiniowanego typu. Zwróćmy uwagę na średnik kończący definicję: jest to jeden z nielicznych przypadków w języku C++, gdy dajemy średnik po nawiasie klamrowym.

Przykład 4.19.

```
#include <iostream.h>
 struct skrypt  {
 char *tytul;
 char *autor;
 float cena;
 long int naklad;
 char status;
 };
int main() {
      skrypt stary, nowy;
      skrypt *wsk;
      wsk = &nowy;
      stary.autor = "Jan Kowalski";
      stary.cena = 12.55;
      wsk->naklad = 50000;
      wsk->status = 'A';
      cout << "stary.autor = " << stary.autor << '\n';
      cout << "stary.cena = " << stary.cena << '\n';
      cout << "wsk->naklad = " << wsk->naklad << '\n';
      cout << "wsk->status = " << wsk->status << '\n';
      cout << "(*wsk).status = " << (*wsk).status << '\n';
      return 0;
}
```

Dyskusja. Definicja struktury na początku programu wprowadza nowy typ o nazwie `skrypt`. Zmienne stary, nowy tego typu deklaruje się tak samo, jak zmienne typów podstawowych. Dostęp do składowych (pól) struktury uzyskuje się za pomocą operatora **.** (kropka) lub operatora **->** (minus i znak większości). Np. dla zmiennej nowy typu skrypt możemy napisać:

```
nowy.autor = "Jan Nowak";
nowy.cena = 21.30;
```

a dla zmiennej wskaźnikowej wsk:

```
wsk->tytul = "Podstawy Informatyki";
```

Operatory "." i "**->**" są więc operatorami selekcji.
Ponieważ wskaźnik `wsk` został zainicjowany adresem zmiennej `nowy`, zatem `*wsk` jest synonimem dla zmiennej `nowy`. Wobec tego zamiast np.
`s->status` mogliśmy napisać `(*wsk).status`. Użycie nawiasów dla `*wsk` było konieczne, ponieważ operatory "**.**" i "**->**" są lewostronnie łączne (wiążą od prawej do lewej).

Struktury mogą być zagnieżdżane; ilustruje to następny przykład.

Przykład 4.20.

```
#include <iostream.h>
struct A     {
 int i;
 char znak;  };
 struct B    {
  int j;
  A aa;
  double d;  };
int  main() {
     B s1, *s2 = &s1;
     s1.j = 4;
     s1.aa.i = 5;
     s2->d = 1.254;
     (s2->aa).znak = 'A';
     cout << (s2->aa).znak << endl;
     return 0;
}
```

Dyskusja. W definicji struktury `B` umieszczono deklarację zmiennej `aa` wcześniej zdefiniowanego typu `A`. Dostęp do składowych typu `A` dla zmiennych typu `B` uzyskuje się wówczas przez dwukrotne zastosowanie operatorów selekcji, np. `s1.aa.i` lub `(s2->aa).znak` dla wskaźnika (znowu konieczne nawiasy okrągłe).

Każda deklaracja struktury wprowadza nowy, unikatowy typ, np.

```
struct s1 { int i ; };
struct s2 { int j ; };
```

są dwoma różnymi typami; zatem w deklaracjach

```
s1 x, y ;
s2 z ;
```

zmienne `x` oraz `y` sa tego samego typu `s1`, ale `x` oraz `z` są różnych typów. Wobec tego przypisania

```
x = y;
y = x;
```

są poprawne, podczas gdy

```
x = z;
z = y;
```

są błędne. Dopuszczalne są natomiast przypisania składowych o tych samych typach, np.

```
x.i = z.j;
```

Pola bitowe

Obszar pamięci zajmowany przez strukturę jest równy sumie obszarów, alokowanych dla jej składowych. Jeżeli np. struktura ma trzy składowe typu **int**, a implementacja przewiduje 2 bajty na zmienną tego typu, to reprezentacja struktury w pamięci zajmie 6 bajtów. Dla dużych struktur (np. takich, których składowe są dużymi tablicami) obszary te mogą być znacznej wielkości. W takich przypadkach możliwe jest ściślejsze upakowanie pól struktury poprzez zdefiniowanie tzw. *pól bitowych*, które zawierają podaną w deklaracji liczbę bitów. W pamięci komputera pole bitowe jest

zbiorem sąsiadujących ze sobą bitów w obrębie jednej jednostki pamięci zdefiniowanej w implementacji, a nazywanej *słowem*. Rozmieszczenie w pamięci struktury z polami bitowymi jest także zależne od implementacji. Jeżeli dane pole bitowe zajmuje mniej niż jedno słowo, to następne pole może być umieszczone albo w następnym słowie (wtedy nic nie oszczędzamy), albo częściowo w wolnej części pierwszego słowa, a pozostałe bity w następnym słowie. Przy tym, zależnie od implementacji, alokacja pola bitowego może się zaczynać od najmniej znaczącego lub od najbardziej znaczącego bitu słowa.
Deklaracja składowej będącej polem bitowym ma postać:

```
typ nazwa_pola : wyrażenie;
```

gdzie:

`typ` oznacza typ pola i musi być jednym z typów całkowitych, tj. **char**, **short int**, **int**, **long int** ze znakiem (signed) lub bez (unsigned) oraz **enum**; występujące po dwukropku wyrażenie określa liczbę bitów zajmowaną przez dane pole.

Pola bitowe mogą być również deklarowane bez nazwy (tzn. tylko typ, po nim dwukropek, a następnie szerokość pola). Takie pole nie może być inicjowane, ale jest użyteczne, ponieważ przy zadeklarowanej szerokości 0 (zero) wymusza dopełnienie do całego słowa wcześniej zadeklarowanego pola bitowego. Dzięki temu alokacja następnego pola bitowego może się zacząć od początku następnego słowa.

Pola bitowe zachowują się jak małe liczby całkowite i mogą występować w wyrażeniach arytmetycznych, w których przeprowadza się operacje na liczbach całkowitych. Przykładowo, deklarację

```
struct sygnalizatory
{
 unsigned int sg1 : 1;
 unsigned int sg2 : 1;
 unsigned int sg3 : 1;
} s;
```

możemy wykorzystać do włączania lub wyłączania sygnalizatorów

```
s.sg1 = 1;  s.sg2 = 0;  s.sg3 = 1;
```

lub testowania ich stanu

```
if (s.sg1 == 0 && s.sg3 == 0) s.sg2 = 1;
```

Zwróćmy uwagę na składnię dostępu do pola bitowego: jest ona taka sama, jak dla zwykłego pola.

Przykład 4.21.

```
#include <iostream.h>
struct flagi {
 unsigned int flaga1  : 1;
 unsigned int flaga2  : 1;
 unsigned char flaga3 : 6;
};
int  main() {
     flagi bit1;
     cout << sizeof bit1 << endl;
     bit1.flaga1 = 1;
     bit1.flaga2 = bit1.flaga1 ^ 1;
     bit1.flaga3 = '*';
     cout << bit1.flaga1 << ' '
          << bit1.flaga2 << endl;
     cout << bit1.flaga3 << endl;
     return 0;
}
```

Wydruk z programu, dla `sizeof(int)==2`, ma postać:

1
1 0
*****

Przykład 4.22.

```
#include <iostream.h>
struct bity {
 int b1  : 8;
 int b2  : 16;
 int b3 :  16;
};
int  main() {
     bity bit1;
     bit1.b1 = 0;
     bit1.b2 = ~bit1.b1 & 1;
     bit1.b3 = 32767;
     cout << sizeof bit1 << endl;
     cout << bit1.b1 << ' ';
     cout << bit1.b2 << endl;
     cout << bit1.b3 << endl;
     return 0;
}
```

Wydruk z programu, dla `sizeof(int)==2`, ma postać:

5
0 1
32767

Komentarz. W pierwszym przykładzie zaoszczędziliśmy 4 bajty, a w drugim 1 bajt. Wynik wydaje się być niezły, choć już na pierwszy rzut oka widać, że oszczędność opłaciliśmy dłuższym kodem programu (dwukropki i wielkości pól). Tak więc na pewno większy będzie kod wykonalny i nieco dłuższy czas wykonania. I tak jest w większości przypadków, przy czym często jeszcze operowanie na polach bitowych wymaga dodatkowych instrukcji. Wniosek stąd oczywisty: nie warto oszczędzać kilku bajtów pamięci, ale gdy w grę wchodzą dziesiątki czy setki kilobajtów, to stosowanie pól bitowych może się opłacać.

Unie

Unia, podobnie jak struktura, grupuje składowe różnych typów. Jednak – w odróżnieniu od struktury – tylko jedna ze składowych unii może być "aktywna" w danym momencie. Wynika to stąd, że każda ze składowych unii ma ten sam adres początkowy w pamięci, zaś obszar pamięci zajmowany przez unię jest równy rozmiarowi jej największej składowej. Definicja unii ma postać:

```
union nazwa { ... };
```

Przykład 4.23.

```
#include <iostream.h>
 union test  {
  long int i;  double d;   char znak;
};
int   main()  {
      test uu;
      test* uwsk;
      uwsk = &uu;
      uu.d = 14.85;
      cout << uu.d << endl;
      cout << uu.i << endl;
      uu.i = 123456789;
      cout << uu.i << endl;
      cout << uu.d << endl;
      uwsk->i = 79;
      cout << uu.i << endl;
      return 0;
}
```

Dyskusja. W powyższym przykładzie druga instrukcja `cout` jest poprawna, ale `uu.i` odpowiada części danej typu **double** uprzednio przypisanej i może nie mieć sensownej interpretacji, ponieważ w tym momencie "aktywną" składową była `uu.d`.

Unię można zainicjować albo wyrażeniem prostym tego samego typu, albo ujętą w nawiasy klamrowe wartością pierwszej zadeklarowanej składowej. Np. unię uu można zainicjować deklaracją

```
test uu = { 1 };
```

Podobnie jak dla struktur, można stosować instrukcję przypisania dla unii tego samego typu, np.

```
test uu, uu1, uu2;
uu2 = uu1 = uu;
```

W definicji unii można pominąć nazwę po słowie kluczowym **union**, a deklaracje zmiennych umieścić pomiędzy zamykającym nawiasem klamrowym a średnikiem, np.

```
union { int i; char *p; } uu, *uwsk = &uu;
```

Powyższa definicja również tworzy unikatowy typ. Ponieważ typ występuje tutaj bez nazwy, taką definicję stosuje się wtedy, gdy unia ma być wykorzystana tylko jeden raz. Natomiast dostęp do składowych jest taki sam, jak poprzednio.

Przykład 4.24.

```
#include <iostream.h>
int   main() {
      union { int i; char *p; } uu, *uwsk = &uu;
      uu.i = 14;
      cout << uu.i << endl;
      uwsk->p = "abcd";
      cout << uwsk->p << endl;
      return 0;
}
```

Specyficzną dla języka C++ jest unia bez nazwy i bez deklaracji zmiennych, o składni:
`union { wykaz-składowych };`
Taki zapis, nazywany *unią anonimową*, nie tworzy nowego typu, a jedynie deklaruje szereg składowych, które współdzielą ten sam adres w pamięci. Ponieważ unia nie ma nazwy, jej elementy są dostępne bezpośrednio – nie ma potrzeby stosowania operatorów "**.**" i "**->**".

Przykład 4.25.

```
#include <iostream.h>
int   main()  {
      union { int i; char *p; } ;
      i = 14;
      cout << i << endl;
      p = "abcd";
      cout << p << endl;
      return 0;
}
```

5. Funkcje

Przed wprowadzeniem klas i obiektów podprogramy-funkcje (i podprogramy-procedury) były najważniejszymi jednostkami modularyzacji programów.
Funkcję można uważać za operację zdefiniowaną przez programistę i reprezentowaną przez nazwę funkcji. Operandami funkcji są jej argumenty, ujęte w nawiasy okrągłe i oddzielone przecinkami. Typ funkcji określa typ wartości zwracanej przez funkcję.

5.1. Deklaracja, definicja i wywołanie funkcji

5.1.1. Deklaracje funkcji

Deklaracja funkcji ma postać:

```
typ nazwa(deklaracje argumentów);
```

Występujące tutaj trzy elementy: typ zwracany, nazwa funkcji i wykaz argumentów nazywane są łącznie prototypem funkcji. W języku C++ obowiązuje zasada deklarowania prototypu każdej funkcji przed jej użyciem. Prototyp danej funkcji może wystąpić w tym samym programie wiele razy, natomiast brak prototypu funkcji wywoływanej w programie jest błędem syntaktycznym. Argumenty w deklaracji funkcji nazywa się również argumentami formalnymi lub parametrami formalnymi. Wykonanie instrukcji deklaracji funkcji nie alokuje żadnego obszaru pamięci dla parametrów formalnych.

Przykład 5.1.

```
int f1();
void f3();
void f3(void);
int* f5(int);
int (*fp6) (const char*, const char*);
extern double sqrt(double);
extern char *strcpy(char *to, const char *from);
extern int strlen(const char *st);
extern int strcmp(const char *s1, const char *s2);
int printf(const char *format, ...);
```

Dyskusja. Warto w tym miejscu zaznaczyć, że opuszczanie identyfikatora typu **int** jest dopuszczalne dla starszych wersji kompilatorów. Najnowsze ustalenia standardu ANSI/ISO dla języka C++ stanowią, że – podobnie jak dla zmiennych i stałych – także typ zwracany funkcji musi być podawany jawnie (nie ma "niejawnego" **int**).

Deklaracje void `f3();` i `void f3(void);` są równoważne; funkcja o nazwie `f3` ma pustą listę argumentów i nie zwraca żadnej wartości do programu, w którym będzie wywoływana (jest odpowiednikiem procedury bezparametrowej, używanej np. w językach Pascal i Modula-2). Funkcja `f5` przyjmuje jeden argument typu **int** i zwraca wskaźnik do typu **int**. Zapis

```
int  (*fp6) (const char*, const char*);
```

deklaruje wskaźnik `fp6` do funkcji `*fp6`, która przyjmuje dwa wskaźniki do stałych typu **char** i zwraca wartość typu **int**. Nawiasy w `(*fp6)` są konieczne dla poprawnego wiązania części składowych zapisu, ponieważ zapis bez nawiasów
```
int *fp6(const char*, const char*);
```

mówi, że `fp6` jest funkcją (a nie wskaźnikiem) zwracającą wskaźnik do **int**, podobnie jak `f5`.

Omówione dotąd deklaracje stwierdzały niejawnie, że definicje podanych prototypów funkcji są dostępne w plikach wchodzących w skład programu. Inaczej mówiąc, definicje te są zewnętrzne w stosunku do tej funkcji (np. main), z której deklarowane funkcje będą wołane. Ponieważ w języku C++ funkcje nie mogą być zagnieżdżane, zatem każda funkcja jest *zewnętrzna* w stosunku do pozostałych funkcji wchodzących w skład programu. Tę "zewnętrzność" definicji funkcji można wyrazić jawnie, poprzedzając prototyp funkcji słowem kluczowym **extern**. Cztery kolejne deklaracje korzystają z tej właśnie konwencji (poprzednie deklaracje miały specyfikator **extern** nadawany domyślnie).
Zauważmy, że trzy spośród czterech zadeklarowanych ze słowem **extern** funkcji operują na łańcuchach znaków: `strcpy` kopiuje łańcuch `from` do `to`, `strlen` zwraca długość łałcucha, zaś `strcmp` zwraca 0 jeżeli łańcuchy są równe. Zauważmy też, że wymienione trzy funkcje zadeklarowano z nazwami argumentów formalnych (`to`, `from`, `st`, `s1`, `s2`). Nazwy te są opcjonalne, ponieważ kompilator je pomija. Mogą one jednak ułatwić zrozumienie programu, szczególnie gdy funkcje mają wiele argumentów.
Znaczenie modyfikatora **const** w wykazie argumentów jest widoczne z kontekstu: zapobiega on zmianie argumentu wywołania funkcji tam, gdzie byłoby to niepożądane. Ostatni zapis:

```
int printf(const char *format, ...);
```

deklaruje funkcję typu **int**, którą można wywoływać ze zmieniającą się liczbą i typami argumentów, co sygnalizuje kompilatorowi symbol "**...**". Inaczej mówiąc, w wywołaniu musi wystąpić co najmniej jeden argument typu **char***. N.b. funkcja `printf` zadeklarowana w pliku `stdio.h` generuje formatowane wyjście pod kontrolą łańcucha formatującego format, np.

```
printf("Hej, jestem tutaj\n");
printf("Moje nazwisko :  %s %s\n", nazwisko, imie);
printf("Moja pensja : %d\n", pensja);
```

5.1.2. Wywołanie funkcji

Wywołanie funkcji jest poleceniem obliczenia wartości wyrażenia, zwracanej przez nazwę funkcji. Instrukcja wywołania ma składnię:

```
nazwa (argumenty aktualne);
```

gdzie `nazwa` jest zadeklarowaną wcześniej nazwą funkcji, **argumenty aktualne** są wartościami argumentów formalnych, zaś para nawiasów okrągłych **()** jest **operatorem wywołania**. Liczba, kolejność i typy argumentów aktualnych powinny dokładnie odpowiadać zadeklarowanym argumentom formalnym. Przy niezgodności typów argumentów kompilator stara się wykonać konwersje niejawne; jeżeli nie może dokonać sensownej konwersji, sygnalizuje błąd.
Zastosowanie operatora wywołania do nazwy funkcji powoduje alokację pamięci dla argumentów formalnych i przekazanie im wartości argumentów aktualnych. Od tej chwili argumenty formalne stają się zmiennymi lokalnymi o wartościach inicjalnych równych przesłanym do nich wartościom argumentów aktualnych. Zasadniczym sposobem przekazywania argumentów do funkcji jest przekazywanie przez wartość: do każdego argumentu formalnego jest przesyłana kopia argumentu aktualnego. Ponieważ argumenty formalne po wywołaniu stają się zmiennymi lokalnymi funkcji, zatem wszelkie wykonywane na nich operacje nie zmieniają wartości argumentów aktualnych. Wyjątkiem od tej zasady jest przesłanie do funkcji adresów argumentów aktualnych za pomocą wskaźników lub referencji. Sytuacje, w których jest to pożądane, omówimy później.

5.1.3. Definicja funkcji

Składnia definicji funkcji jest następująca:

```
typ  nazwa (deklaracje argumentów)
{
 instrukcje
}
```

Podobnie jak w deklaracji funkcji, definicja podaje typ zwracany przez funkcję, jej nazwę oraz argumenty formalne. Argumentami formalnymi funkcji mogą być zmienne wszystkich typów podstawowych, struktury, unie oraz wskaźniki i referencje do tych typów, a także zmienne typów definiowanych przez użytkownika. Nie mogą być nimi tablice, ale mogą być wskaźniki do tablic. Typ funkcji nie może być typem tablicowym ani funkcyjnym, ale może być wskaźnikiem do jednego z tych typów. Zarówno typ zwracany, jak i typy argumentów muszą być podawane w postaci jednoznacznych identyfikatorów. Identyfikatory mogą być nazwami typów wbudowanych (np. **char**, **int**, **long int**) lub zdefiniowanych wcześniej przez użytkownika. Inaczej mówiąc, w nagłówku funkcji nie mogą występować definicje typów.
Instrukcje w bloku (nazywanym również *ciałem funkcji*) mogą być instrukcjami deklaracji. Ostatnią instrukcją przed nawiasem klamrowym zamykającym blok funkcji musi być instrukcja `return;` jedynie dla funkcji typu **void** instrukcja `return;` jest opcją. Zatem definicja
`int f() { };`
jest błędna, natomiast definicja
`void f() { };`
jest poprawna.
Instrukcja `return;` występuje często w postaci: `return wyrażenie;`, gdzie `wyrażenie` określa wartość zwracaną przez funkcję. Jeżeli typ tego wyrażenia nie jest identyczny z typem funkcji, to kompilator będzie próbował osiągnąć zgodność typów drogą niejawnych konwersji. Jeżeli okaże się to niemożliwe, to kompilacja zostanie zaniechana. Zgodność typu zwracanego z zadeklarowanym typem funkcji można również wymusić drogą konwersji jawnej.
Deklaracje argumentów muszą podawać oddzielnie typ każdego argumentu; nazwy argumentów są opcjonalne. Definicja, która nie zawiera nazw argumentów, a jedynie ich typy, jest syntaktycznie poprawna. Oznacza ona, że argumenty formalne nie są używane w bloku funkcji. Taka sytuacja może wystąpić wtedy, gdy przewidujemy wykorzystanie argumentów formalnych w bloku funkcji w przyszłości, a nie chcemy dopuścić do zmiany postaci wywołania funkcji. W bloku funkcji może wystąpić więcej niż jedna instrukcja `return;`. Ilustruje to poniższy przykład.

Przykład 5.2.

```
#include <iostream.h>
//deklaracja funkcji - prototyp
int dods(int, int);
int  main() {
     int i, j, k;
     cout << "Wprowadz dwie liczby typu int: ";
     cin >> i >> j;
     cout << '\n';
     k = dods (i,j); //wywolanie funkcji
     cout << "i= " << i << "\tj= " << j << '\n';
     cout << "dods(i,j)= " << k << '\n';
     return 0;
}
int dods (int n, int m)
{
 if (n + m > 10) return n + m;
 else return n;
}
```

5.2. Przekazywanie argumentów

Wywołanie funkcji zawiesza wykonanie funkcji wołającej i powoduje zapamiętanie adresu następnej instrukcji do wykonania po powrocie z funkcji wołanej. Adres ten, nazywany *adresem powrotnym*, zostaje umieszczony w pamięci na *stosie programu* (ang. run-time stack). Wywołana funkcja otrzymuje wydzielony obszar pamięci na stosie programu, nazywany *rekordem aktywacji* lub *stosem funkcji*. W rekordzie aktywacji zostają umieszczone argumenty formalne, inicjowane jawnie w deklaracji funkcji, lub niejawnie przez wartości argumentów aktualnych.
Jawne inicjowanie argumentów w deklaracji (nie w definicji!) funkcji można traktować jako przykład *przeciążenia funkcji*; tematem tym zajmiemy się później bardziej szczegółowo. Weźmy następujący przykład: w prototypie funkcji, która symuluje ekran monitora, wprowadźmy inicjalne wartości domyślne dla szerokości, wysokości i tła ekranu

```
char* ekran(int x=80, int y=24, char bg = ' ');
```

Wprowadzone wartości początkowe argumentów `x`, `y` oraz `bg` są domyślne w tym sensie, że jeżeli w wywołaniu funkcji nie podamy argumentu aktualnego, to na stosie funkcji zostanie "położona" wartość domyślna argumentu formalnego.
Jeżeli teraz zadeklarujemy zmienną `char* kursor`, to wywołanie
```
kursor = ekran();
```
jest równoważne wywołaniu
```
kursor = ekran(80, 24, ' ');
```

Jeżeli w wywołaniu podamy inną od domyślnej wartość argumentu, to zastąpi ona wartość domyślną, np. wywołanie
```
kursor = ekran(132);
```
jest równoważne wywołaniu
```
kursor = ekran(132, 24, ' ');
```
zaś wywołanie
```
kursor = ekran(132, 66);
```
jest równoważne wywołaniu
```
kursor = ekran(132, 66, ' ');
```
Składnia ostatniego wywołania pokazuje że nie można podać wartości pierwszego z prawej argumentu nie podając wszystkich wartości po lewej.
Deklaracja funkcji nie musi zawierać wartości domyślnych dla wszystkich argumentów, ale podawane wartości muszą się zaczynać od skrajnego prawego argumentu, np.

```
char* ekran( int x, int y, char bg = ' ' );
char* ekran( int x, int y = 24, bg = ' ' );
```

Wobec tego deklaracje

```
char* ekran (int x = 80, int y, char bg = ' ');
char* ekran ( int x, int y = 24, char bg );
```

są błędne.

5.2.1. Przekazywanie argumentów przez wartość

W języku C++ przekazywanie argumentów przez wartość jest domyślnym mechanizmem językowym. Przy braku takiego mechanizmu każda zmiana wartości argumentu formalnego nie poprzedzonego modyfikatorem **const** wywołałaby taką samą zmianę argumentu aktualnego. Założenie to zostało podyktowane tym, że ewentualne zmiany wartości argumentów aktualnych, będące wynikiem wykonania jakiejś funkcji, są na ogół traktowane jako niepożądane *efekty uboczne*.
Przekazywanie argumentów przez wartość pokazano już w ostatnim przykładzie. Przykład pokazany niżej ilustruje znaczenie prototypu oraz niejawnej konwersji przy wywoływaniu funkcji.

Przykład 5.3.

```
#include <iostream.h>
int imax(int x, int y);
int  main() {
     float zf = 35.7;
     double zd = 11.0;
     int ii;
     ii = imax( zf, zd );
     cout << "ii = " << ii << endl;
     return 0;
}
int imax(int x, int y)
{
 if (x > y) return x;
 else return y;
}
```

Dyskusja. Ponieważ funkcja `imax()` ma prototyp z parametrami typu **int**, to wykonanie operacji wywołania rozpocznie się od dwóch niejawnych konwersji zmiennych `zf` i `zd` do typu **int**. Po przeprowadzeniu konwersji zmienne te zostaną położone na stos funkcji `imax`.

Przekazywanie argumentów przez wartość nie będzie dogodnym mechanizmem w dwóch przypadkach:

1. gdy wartości argumentu aktualnego muszą być przez funkcję zmodyfikowane,
2. gdy przekazywany argument reprezentuje duży obszar pamięci (np. tablica, struktura).

Dla tablic mamy na szczęście mechanizm, który nakazuje kompilatorowi przekazanie argumentów aktualnych w postaci adresu pierwszego elementu tablicy zamiast kopii całej tablicy. Natomiast w pierwszym przypadku należy znaleźć własne rozwiązanie.

Klasycznym przykładem nieprzydatności przekazywania argumentów przez wartość jest operacja zamiany wartości dwóch zmiennych. Funkcja

```
void zamiana(int x, int y)
{
 int pomoc = y;
 y = x;
 x = pomoc;
}
```

zamieni miejscami wartości `x` oraz `y`, które po wywołaniu staną się lokalnymi kopiami wartości argumentów aktualnych. Niestety nie zdołamy tych zamienionych wartości przesłać do funkcji wołającej.

5.2.2. Przekazywanie argumentów przez adres

Podniesione w końcowej części p. 5.2.1 niedostatki przekazywania argumentów przez wartość można przezwyciężyć dwoma sposobami. Pierwszy z nich polega na zastosowaniu wskaźników. Sposób ten ilustruje poniższy przykład.

Przykład 5.4.

```
#include <iostream.h>
void zamiana1 (int*, int* );
int  main()   {
```

```
      int i = 10;
      int j = 20;
      zamiana1( &i, &j );
      cout << "Po zamianie i=" << i << "\tj=" << j << endl;
      return 0;
}
void zamiana1(int* x, int* y)
{
 int pomoc = *y;
 *y = *x;
 *x = pomoc;
}
```

Dyskusja. Powyższy program – ze wskaźnikową wersją argumentów funkcji zamiana1 – daje następujący wydruk:

Po zamianie i=20 j=10

Otrzymujemy zatem wynik, o który nam chodziło, ale program wydaje się być mało czytelny. Alternatywne rozwiązanie tego samego zadania wykorzystuje referencje zamiast wskaźników, dając prostszą i bardziej czytelną notację.

Przykład 5.5.

```
#include <iostream.h>
void zamiana2 (int& i, int& j );
int   main() {
      int i = 10;
      int j = 20;
      zamiana2( i, j );
      cout << "Po zamianie i=" << i << "\tj= " << j << endl;
      return 0;
}
void zamiana2(int& x, int& y)
{
 int pomoc = y;
 y = x;
 x = pomoc;
}
```

5.3. Komunikacja funkcji main z otoczeniem

Każdy program w C++ musi mieć dokładnie jedną funkcję o nazwie main i każdy program zaczyna się wykonywać od wywołania tej funkcji. Funkcja main nie jest predefiniowana przez kompilator, nie może być przeciążana, a jej typ jest zależny od implementacji. W wersji wzorcowej języka zaleca się dwie następujące definicje funkcji main:

```
int main() { /* ... */ }
```
lub
```
int main(int argc, char *argv[]) { /* ... */ }
```

przy czym dopuszcza się możliwość dodawania dalszych argumentów po `argv[]`. W drugiej postaci funkcji argument `argc` oznacza liczbę parametrów przekazywanych do programu z otoczenia, w którym program jest uruchamiany. Parametry te są przekazywane jako zakończone znakiem `'\0'` łańcuchy znaków
`argv[]`. Tak więc do parametru formalnego `argv[0]` jest przekazywany pierwszy łańcuch znaków, a do parametru `argv[argc - 1]` ostatni łańcuch.

Parametr `argv[0]` jest nazwą używaną przy wywołaniu programu z wiersza rozkazowego. Ponieważ `argc` jest liczbą przekazywanych parametrów, to `argv[argc]==0`. Zgodnie z terminologią przyjętą dla łańcuchów znaków, typem argv jest `char*[argc + 1]`.
Funkcja main nie może być wywoływana w obrębie programu; nie jest możliwe odwołanie do adresu main; funkcja main nie może być deklarowana ze słowem kluczowym inline lub static.
Załóżmy, że wiersz rozkazowy wygląda następująco:

```
prog1  17.5  22.9
```

gdzie `prog1` jest nazwą pliku, w którym umieszczony jest kod binarny naszego programu, a ciągi znaków 17.5 i 22.9 są danymi, potrzebnymi dla wykonania programu. Wówczas parametry funkcji main będą następujące:

```
argc                3
argv[0]    "prog1"
argv[1]    "17.5"
argv[2]    "22.9"
argv[3]    0
```

Podawane w wierszu rozkazowym parametry są często nazwami plików, na których ma operować nasz program. Załóżmy, że kod binarny programu kopiującego pliki jest umieszczony w pliku `prog2`. Parametrami, które należy podać, są: nazwa pliku kopiowanego `plik1` i nazwa kopii `plik2`. Wówczas napiszemy następujący wiersz rozkazowy:

```
prog2  plik1  plik2
```

Przykład 5.6.

```
#include <iostream.h>
#include <stdlib.h>
int  main(int argc, char *argv[]) {
     if ( argc != 6 )     {
     cerr << "Niepoprawna liczba parametrow\n";
     exit ( 1);          }
     int i ;
     cout << "Wartosc argc wynosi: " << argc
          << endl << endl;
     cout << "Przekazano " << argc
          << " argumentow do main: " << endl << endl;
     for (i = 0; i < argc; i++)
     cout << "  argv[" << i << "]: " << argv[i] << endl;
     return 0;
/*  Unix, Solaris 2.4, plik p81.cc,
    kompilator CC lub g++ :
    Po wykonaniu polecenia: CC p81.cc -o p81
    wywolaj P81 z linii rozkazowej:
    P81 arg1 arg2 arg3 arg4 arg5
    (6 argumentow, wliczajac nazwe programu) i
     zanotuj wynik    */
/*  MS-DOS, Borlandc, plik P81.CPP, kompilator BCC:
    Po uzyskaniu pliku P81.exe
    wywolaj P81 z linii rozkazowej:
    P81 arg1 arg2 arg3 arg4 arg5
    (6 argumentow, wliczajac nazwe programu) i
     zanotuj wynik    */
}
```

Postać wydruku:

Wartosc argc wynosi: 6

Przekazano 6 argumentow do main:

argv[0]: p82
argv[1]: arg1
argv[2]: arg2
argv[3]: arg3
argv[4]: arg4
argv[5]: arg5

(Pod DOS, zamiast argv[0]: p81, byłoby: argv[0]: C:\BORLANDC\P81.EXE).

Dyskusja. W programie umieszczono dyrektywę włączenia pliku nagłówkowego `stdlib.h`, który zawiera deklarację funkcji `exit()`. Instrukcja **if** zabezpiecza nas przed podaniem niewłaściwej liczby parametrów, przekazywanych do programu. Standardowy strumień cerr jest zadeklarowany, podobnie jak `cin` i `cout`, w pliku nagłówkowym `iostream.h`. Gdybyśmy wywołali program p82 z inną niż 6 liczbą parametrów, to wykonanie programu zostanie zaniechane, a operator wyjścia **<<** wyprowadzi na ekran napis:

Niepoprawna liczba parametrow

Zauważmy ponadto, że podawane w wierszu rozkazowym parametry są niepodzielnymi ciągami znaków (słowami), oddzielonymi jedną lub kilku spacjami. Gdyby program wymagał podania parametru w postaci kilku słów oddzielonych spacjami, to taki parametr należy ująć w podwójne apostrofy, np.

```
prog3 "argument ze spacjami"  arg2 arg3
```

Wspomniana wyżej możliwość dodawania dalszych argumentów po `argv[]` może być wykorzystana do uzyskania informacji o zmiennych środowiska, nazywanych również łańcuchami otoczenia. W takim przypadku nagłówek funkcji main będzie miał postać:

```
int main(int argc, char *argv[], char *env[])
```

a wartości zmiennych środowiska możemy wyświetlić sekwencją instrukcji

```
cout << endl
     <<"Lancuchy otoczenia w tym systemie:\n\n";
for (i = 0; env[i] != NULL; i++)
cout << "  env[" << i << "]: " << env[i] << endl;
```

5.4. Funkcje rozwijalne

Modularyzacja programów prowadzi do definiowania wymaganych operacji w postaci definicji funkcji, często bardzo prostych, zawierających jedną lub kilka instrukcji. Jeżeli takie funkcje są często wywoływane, to narzut czasowy na wywołanie i powrót staje się znaczny i może obniżyć efektywność programu. Narzut ten obejmuje m. in. czas potrzebny na alokację pamięci dla argumentów i zmiennych lokalnych wołanej funkcji i czas na skopiowanie argumentów aktualnych do przydzielonych miejsc pamięci. W przypadku, gdy funkcja wykonuje złożone zadania, czas na wykonanie i powrót będzie mały w porównaniu do czasu obliczeń. Natomiast gdy funkcja wykonuje

jedną lub dwie instrukcje, narzut czasowy na wywołanie i powrót może stanowić znaczny procent w stosunku do czasu obliczeń.
Definicję takich krótkich funkcji możemy poprzedzić słowem kluczowym **inline**. Specyfikator *inline* jest wskazówką (nie poleceniem!) dla kompilatora, aby każde wywołanie takiej funkcji starał się zastąpić instrukcjami, które definiują funkcję. W wyniku eliminacji wielu wywołań i powrotów otrzymuje się program szybszy, ale zajmujący większy obszar pamięci, ponieważ instrukcje, które definiują funkcję, są powielane przez kompilator w każdym punkcie, gdzie funkcja jest wywoływana.
Jednym z typowych przykładów funkcji, definiowanej jako rozwijalna, może być funkcja, której jedyną instrukcją jest wywołanie innej funkcji. Inne przykłady takich funkcji podano niżej.

```
inline int abs(int i) { return i < 0 ? -i : i; }
inline int min( int a, int b)
{ return a < b ? a : b; }
inline int podzielne(int i, int j)
{ return i%j; }
inline double pole(double a, double b)
{return a*b;}
inline void f()
{ char z = '\0';if(cin >> z && z != 'q') f();}
```

Przykład 5.7.

```
#include <iostream.h>
inline int podzielne(int n, int m)
{ return !(n%m); }
int   main() {
      int i,j;
      int k;
      cin >> i >> j ;
      k = podzielne(i,j);
      cout << "k= " << k << '\n';
      return 0;
}
```

Przykład 5.8.

```
#include <iostream.h>
inline void f()
{ char z= '\0'; if(cin >> z && z !='q') f(); }
int   main()  {
      f();
      return 0;
}
```

Dyskusja. Zdefiniowana w tym krótkim programie funkcja `f()` może być wygodną "wstawką" w dłuższych programach, w których chcemy uzależnić wyjście z programu od wprowadzenia określonego znaku z klawiatury (tutaj znak `'q'`).

Przykład 5.9.

```
#include <iostream.h>
inline int parzyste(int n) {return !(n%2); }
int  main() {
     if(parzyste(10))
     cout << "Liczba 10 jest parzysta\n";
     else if(parzyste(11))
     cout << "Liczba 11 jest parzysta\n";
     return 0;
}
```

Dyskusja. W tym przykładzie funkcja `parzyste()`, która zwraca prawdę (wartość różną od zera) jeśli jej argument jest parzysty, została zdefiniowana jako rozwijalna. Oznacza to, że instrukcja

```
if(parzyste(10)) cout << "Liczba 10 jest parzysta\n";
```
jest funkcjonalnie równoważna instrukcji
```
if(!(10%2)) cout << "Liczba 10 jest parzysta\n";
```

Zauważmy, że definicja funkcji rozwijalnej została umieszczona przed funkcją main(). Jest to konieczne, gdyż inaczej kompilator nie mógłby się dowiedzieć, jak ma wyglądać rozwinięcie parzyste(10) w instrukcji *if*.
Zwróżmy jeszcze uwagę na fakt, że jeżeli zmienimy definicję funkcji rozwijalnej, to wszystkie wywołania muszą być powtórnie kompilowane.
Jak już wspomniano, poprzedzenie definicji funkcji rozwijalnej słowem kluczowym **inline** jest jedynie życzeniem programisty, aby kompilator potraktował ją jako funkcję rozwijalną. Kompilator zignoruje to życzenie i wywoła ją dopiero w fazie wykonania programu, jeżeli funkcja zdefiniowana z **inline**:
- zawiera zbyt wiele instrukcji
- jest funkcją rekurencyjną
- jest wywoływana przed jej definicją
- jest wywoływana dwa lub więcej razy w tym samym wyrażeniu
- zawiera pętlę, instrukcję **switch**, lub **goto**
- jest wywoływana przez wskaźnik.

☞ *Uwaga. W języku C++ istnieje możliwość niejawnego (bez inline) definiowania funkcji rozwijalnych, wykorzystywana w szczególności w bibliotekach klas. Gdyby rozwijanie funkcji nie było możliwe, to teksty źródłowe wielu prostych, ale ważnych i często używanych funkcji bibliotecznych byłyby niedostępne dla kompilatora.*

5.5. Funkcje rekurencyjne

Funkcja, która wywołuje sama siebie, bezpośrednio lub pośrednio, jest nazywana *rekurencyjną*. Każde kolejne wywołanie funkcji powoduje utworzenie nowej kopii jej argumentów i zmiennych lokalnych (automatycznych). Funkcje rekurencyjne muszą zawsze zawierać warunek stopu (zatrzymania). W przeciwnym razie funkcja będzie wywoływać sama siebie bez kołca, tworząc pętlę nieskończoną.
Klasycznym przykładem rekurencji jest definicja silni

```
n! = 1 * 2 * 3 * ... * (n - 1) * n
```

którą można zapisać w równoważnej postaci rekurencyjnej

```
       1         dla n = 0
n! =
       n*(n-1)!  dla n > 0
```

Podane niżej dwa przykłady pokazują iteracyjną i rekurencyjną wersję silni, zaś rysunek 5-1 ilustruje mechanizm wywołań dla n równego 3.

Przykład 5.10.

```
//Iteracyjne obliczanie silni
#include <iostream.h>
long int sil(long int n);
int  main() {
     long int j = 3;
     long int k = sil(j);
     cout << "sil(" << j << ") =" << k << endl;
     return 0;    }
     long int sil(long int n)        {
     long int pomoc = 1;
     for ( int i = 1; i <= n; i++)
     pomoc = pomoc * i;
     return n == 1  ? n : pomoc;
}
```

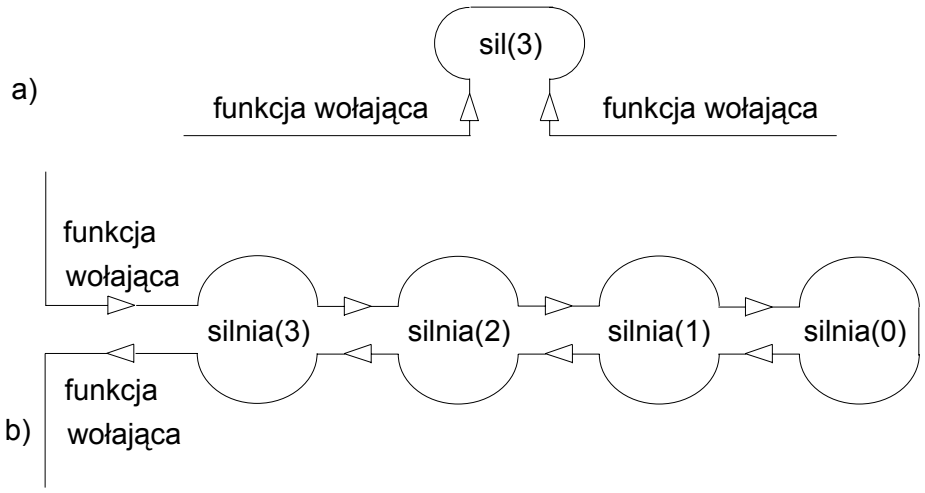
Przykład 5.11.

```
// Rekurencyjne obliczanie silni
#include <iostream.h>
long int silnia(long int n);

int  main() {
     long int j = 3;
     long int k = silnia(j);
     cout << "silnia(" << j << ") =" << k << endl;
     return 0;
}

long int silnia(long int n)
{
  if ( n > 0)
  return n * silnia(n - 1);
  else return 1;
}
```

Dyskusja. Wywołanie funkcji iteracyjnej `sil(long int n)` ma miejsce tylko jeden raz. Na czas wykonania funkcji zostaje zawieszone wykonanie funkcji wołającej. Po powrocie wykonywana jest następna, zapamiętana na stosie, instrukcja funkcji wołającej.
Wykonanie funkcji rekurencyjnej `silnia(long int n)` przebiega następująco:

1. Wywołanie o postaci `long int k = silnia(3);` zawiesza wykonanie funkcji wołającej z jednoczesnym zapamiętaniem adresu następnej do wykonania instrukcji funkcji wołającej. Kopia argumentu aktualnego (tj. wartość 3) zostanie przekazana do funkcji wołanej.
2. Po sprawdzeniu, że `n>0`, wykonanie funkcji `silnia(3)` zostanie zawieszone, ponieważ próba wykonania instrukcji `return 3*silnia(2);` spowoduje wywołanie kopii `silnia(2)`. W rezultacie na stosie funkcji `silnia` (nazywanym stosem rekursji) zostaną umieszczone kolejno: kopia funkcji silnia(3) i adres powrotny do `silnia(3)`.
3. Wykonanie `silnia(2)` również zostanie zawieszone w rezultacie wywołania `silnia(1)` w instrukcji `return`. `silnia(2)` oraz adres powrotny zostaną położone na stos rekursji.
4. `silnia(1)` wywoła `silnia(0)`, tj. wykonalną kopię funkcji. Przed wykonaniem `silnia(0)` funkcja `silnia(1)` oraz adres powrotny zostaną umieszczone na stosie rekursji.

5. W rezultacie wywołania `silnia(0)` zostanie wykonana instrukcja `return 1`.
Teraz możliwy jest powrót, reprezentowany przez dolne łuki na rys. 5-1(b), tj. wykonanie kolejnych instrukcji `return` i przypisanie ich wartości wyrażeniom
`silnia(1)`, `silnia(2)` i `silnia(3)`.



Rys. 5-1 Wywołanie funkcji: (a) nierekurencyjnej, (b) rekurencyjnej

Podobnie konstruuje się funkcję, która znajduje największy wspólny podzielnik dwóch liczb całkowitych. Podane niżej przykłady prezentują iteracyjną `gcd` i rekurencyjną `rgcd` wersję takiej funkcji.

Przykład 5.12.

```
// gcd - greatest common denominator
// wersja iteracyjna
#include <iostream.h>
int gcd ( int, int );
int  main() {
     int i = 24;
     int j = 32;
     int k = gcd ( i, j );
     cout << k << endl;
     return 0;
}

int gcd (int x, int y)
{
 int temp;
 while ( y )  {
  temp = y;
  y = x % y;
  x = temp;   }
 return x;
}
```

Przykład 5.13.

```
// rgcd - greatest common denominator
// wersja rekurencyjna
```

```
#include <iostream.h>
int rgcd ( int, int );

int  main() {
     int i = 24;
     int j = 32;
     int k = rgcd ( i, j );
     cout << k << endl;
     return 0;
}
int rgcd (int x, int y)
{
  if (y == 0) return x;
  return rgcd (y, x % y);
}
```

Kolejny przykład prezentuje funkcję `long int cyfra(long int n, long int k)`, która generuje wartość k-tej cyfry dziesiętnej liczby całkowitej
n, licząc od prawej, tj. od najmniej znaczącej cyfry. Przykładowe wywołanie `cyfra(28491,2);` zwróci wartość 2, a `cyfra(4788,5);` zwróci wartość 0.

Przykład 5.14.

```
#include <iostream.h>
long int cyfra(long int n, long int k);

int  main()  {
     long int i = 1234;
     long int j = 4;
     long int m = cyfra(i, j);
     cout << "cyfra(" << i << "," << j << ") =" << m << endl;
     return 0;
}

long int cyfra(long int n, long int k)
{
 if ( k == 0)
 return 0;
 else if( k == 1)
 return n % 10;
      else return cyfra(n / 10, k - 1);
}
```

Następny przykład ilustruje jedną z możliwości przekształcenia dziesiętnego zapisu liczby całkowitej na zapis binarny. Funkcja `zapisbinarny` jest typową funkcją rozwijalną, zaś funkcja `drukujbity` – rekurencyjną. Obydwie funkcje są typu **void**.

Przykład 5.15.

```
#include <iostream.h>
void drukujbity (int n);
void zapisbinarny ( int m);
int  main() {
     int j = 15;
     zapisbinarny(j);
     cout << " jest zapisem binarnym liczby " << j ;
     cout << '\n';
```

```
      return 0;
}
void drukujbity (int iloraz)
{
 if ((iloraz / 2) != 0)
  drukujbity (iloraz / 2);
  cout << iloraz % 2;
}
void zapisbinarny (int n)
{
 drukujbity (n);
}
```

5.6. Funkcje w programach wieloplikowych

Kody źródłowe większych programów umieszcza się zwykle w kilku plikach. Jest to racjonalny sposób postępowania, ponieważ pliki składowe programu można oddzielnie sprawdzać, oddzielnie kompilować i dopiero po tych operacjach scalić w jeden binarny kod wykonalny.
Ponieważ w tej pracy przyjęto założenie, że wszystkie koncepcje będą ilustrowane raczej krótkimi programami, zatem i konstruowanie, a następnie przetwarzanie programu wieloplikowego pokażemy na prostym przykładzie.
Dla ustalenia uwagi załóżmy, że naszym zadaniem jest posortowanie w kolejności od najmniejszej do największej ciągu liczb typu **double**. Sortowanie przeprowadzimy za pomocą algorytmu zamiany parami. W algorytmie tym bada się (przegląda) pary sąsiadujących ze sobą liczb i ewentualnie zamienia je miejscami. Schemat działania algorytmu ilustruje tablica 5.1.

Tablica 5.1. Schemat algorytmu zamiany parami

		4 zamiany					3 zamiany			1 zamiana		
tab[0]	7.8	5.4				5.4	2.5			2.5	1.6	1.6
tab[1]	5.4	7.8	2.5			2.5	5.4	1.6		1.6	2.5	2.5
tab[2]	2.5		7.8	1.6		1.6		5.4	3.9	3.9		3.9
tab[3]	1.6			7.8	3.9	3.9			5.4	5.4		5.4
tab[4]	3.9				7.8	7.8				7.8		7.8
	↑ stan początkowy					↑ stan po pierwszym przejrzeniu par				↑ stan po drugim przejrzeniu par		↑ stan końcowy

Implementacja algorytmu w postaci kodu źródłowego programu niestrukturalnego, tj. bez wydzielonych funkcji, może wyglądać jak w poniższym przykładzie.

Przykład 5.16.

```
// Sortowanie - jeden plik
// Pod DOS - SORT.CPP, pod Unix - sort.c
#include <iostream.h>
const int rozmiar = 5;
double tab[rozmiar];
int  main() {
```

```
//Czytanie
      cout<<"Podaj "<<rozmiar<<" liczb typu double:\n";
      for ( int i = 0; i < rozmiar; i++) cin >> tab[i];
      cout << "Tablica przed sortowaniem: \n";
//Pisanie
      for (i = 0; i < rozmiar; i++)
      cout<<"tab["<< i << "] : "<<tab[i]<<endl;
//Sortowanie
      int licznik;
      double pomoc;
      do
      {
       licznik = 0;
       for ( i = 0; i < rozmiar - 1; i++)
       {
        if ( tab[i] > tab[i+1])
            {
             pomoc = tab[i];
             tab[i] = tab[i+1];
             tab[i+1] = pomoc;
             ++licznik;
            }
       }
      } while (licznik);
      cout << "Tablica po sortowaniu: \n";
//Ponownie pisanie
      for (i = 0; i < rozmiar; i++)
      cout<<"tab["<<i<<"] : "<<tab[i]<<endl;
      return 0;
}
```

Dyskusja. W programie zadeklarowano zmienną `licznik`, która jest licznikiem zamian w kolejnych przeglądach par; zmienna `pomoc` służy do chwilowego przechowywania wartości `ab[i]` w kolejnych zamianach. Sortowanie jest wykonywane w pętli **do-while**.

Postępując metodycznie, wydzielimy teraz operacje czytania, sortowania i pisania w oddzielne funkcje. Przekształcony w ten sposób program pokazano w kolejnym przykładzie.

Przykład 5.17.

```
// Sortowanie - jeden plik, wydzielone funkcje
// Pod DOS - SORT1.CPP, pod Unix - sort1.c
#include <iostream.h>
void czytaj(double *tc, int rc);
void pisz(double *tp, int rp);
void sortuj(double *ts, int rs);
const int rozmiar = 5;
double tab[rozmiar];
int  main() {
      cout << "Podaj " << rozmiar << " liczb typu double:\n";
      czytaj(tab, rozmiar);
      pisz(tab, rozmiar);
      sortuj(tab,rozmiar);
      pisz(tab, rozmiar);
      return 0;
}
void czytaj(double *tc, int rc)
{
```

```
 for (int i = 0; i < rc; i++)
    cin >> tc[i];
 cout << endl;
 cout << "Tablica przed sortowaniem: " << endl;
}

void pisz(double *tp, int rp)
{
 for (int i = 0; i < rp; i++)
    cout << "tp[" << i << "] = "
         << tp[i]  << endl;
}
void sortuj(double *ts, int rs)
{
 int licznik,i;
 double pomoc;
 do
   {
    licznik = 0;
    for ( i = 0; i < rs - 1; i++)
      {
       if ( ts[i] > ts[i+1])
         {
           pomoc = ts[i];
           ts[i] = ts[i+1];
           ts[i+1] = pomoc;
           ++licznik;
         }
      }
   }
 while (licznik);
 cout << "Tablica po sortowaniu: \n";
}
```

Dalszy tok postępowania zależy od dostępnego środowiska programowego. Dla ustalenia uwagi założymy, że dostępnym środowiskiem programowym jest C++ firmy Borland pod systemem MS-DOS, bądź kompilator-konsolidator CC pod systemem Unix.
Przy tych założeniach kolejnym krokiem może być umieszczenie definicji funkcji `czytaj()`, `sortuj()`, `pisz()` oraz `main()` w oddzielnych plikach. Plikom tym możemy np. nadać nazwy CZYTAJ.CPP, SORTUJ.CPP, PISZ.CPP i MAIN.CPP dla kompilatora Borland C++ pod MS-DOS, lub czytaj.c, sortuj.c, pisz.c i main.c dla kompilatora CC pod systemem Unix. Następnie pliki źródłowe możemy oddzielnie skompilować odpowiednimi poleceniami:
bcc - c nazwa-pliku
pod MS-DOS
lub
CC -c nazwa-pliku
pod systemem Unix.

Podanie opcji "-c" oznacza, że kompilator nie wywoła automatycznie konsolidatora. Zauważmy, że każda z trzech definicji funkcji `czytaj()`, `sortuj()` i `pisz()` wykorzystuje strumień `cout`, a ponadto funkcja `czytaj()` pobiera znaki ze strumienia `cin`. Zatem warunkiem powodzenia kompilacji będzie dopisanie w każdym z plików źródłowych dyrektywy #include <iostream.h>. W wyniku kompilacji otrzymamy nieskonsolidowane pliki tymczasowe: CZYTAJ.OBJ, itd., pod systemem MS-DOS lub czytaj.o, itd., pod systemem Unix.
Kolejnym krokiem będzie scalenie plików tymczasowych. Polecenie konsolidacji może mieć postać:

```
bcc  main.obj  czytaj.obj  sortuj.obj  pisz.obj
```

pod systemem MS-DOS, lub

```
CC  main.o  czytaj.o  sortuj.o  pisz.o
```

pod systemem Unix.

W wyniku konsolidacji otrzymamy binarny kod wykonalny programu (plik MAIN.EXE pod MS-DOS lub a.out pod systemem Unix).
Kod wykonalny możemy również otrzymać za pomocą jednego polecenia, np.

```
bcc  main.cpp  czytaj.cpp  sortuj.cpp  pisz.cpp
```

dla kompilatora bcc, lub

```
CC  main.c  czytaj.c  sortuj.c pisz.c
```

dla kompilatora CC.

Możliwa jest również kompilacja pliku źródłowego, np. main.c (MAIN.CPP) z następującą po niej konsolidacją plików tymczasowych. Przykładowy wiersz rozkazowy może mieć postać:

```
CC  main.c czytaj.o  sortuj.o  pisz.o
```

Programista ma również możliwość utworzenia własnych plików bibliotecznych. Jeżeli mamy już pliki tymczasowe (.o lub .OBJ) dla przykładowych trzech funkcji `czytaj()`, `sortuj()` i `pisz()`, to możemy z nich utworzyć własną bibliotekę. W systemie Borland C++ wykorzystamy do tego celu program TLIB.EXE. Po wykonaniu polecenia:

```
tlib  biblsort +czytaj  + sortuj  +pisz
```

zostanie utworzony plik biblioteczny BIBLSORT.LIB, który możemy konsolidować z dowolnym programem, zawierającym prototypy i wywołania tych trzech funkcji, np.

```
bcc main.cpp biblsort
```

Pod systemem Unix bibliotekę możemy utworzyć poleceniem *ar* (załóż archiwum) z opcją -r:

```
ar  -r  biblsort.a  czytaj.o  sortuj.o  pisz.o
```

Utworzony w ten sposób plik biblioteczny `biblsort.a` konsoliduje się z funkcją main() w analogiczny sposób, jak dla kompilatora Borland C++:

```
CC  main.c  biblsort.a
```

☞ *Uwaga. W niektórych wersjach systemu Unix istnieje polecenie ranlib, które można zastosować do naszego pliku bibliotecznego* `biblsort.a` *pisząc:* `ranlib biblsort.a`*. Wykonanie polecenia indeksuje plik biblioteczny, dzięki czemu uzyskuje się szybszy dostęp do jego składowych. Jeżeli system nie zawiera polecenia* `ranlib`*, to prawdopodobnie nie jest ono potrzebne.*

Można w tym miejscu zapytać: jaką korzyść (poza oczywistym skróceniem wiersza rozkazowego) uzyskuje się z tworzenia bibliotek, zamiast bezpośredniego wyliczenia wszystkich scalanych plików tymczasowych? Nasz przykład jest zbyt prosty, aby ukazać jakieś korzyści. Gdyby jednak funkcja main() wywoływała funkcję sortuj(), a ta z kolei funkcję pisz(), to polecenie konsolidacji `CC main.c sortuj.o` nie zostanie wykonane, ponieważ konsolidator nie znajdzie pliku `pisz.o`. Natomiast gdy posiadamy bibliotekę `biblsort.a`, to konsolidator "wie" jak ze zbioru wszystkich

plików .o wyciągnąć tylko te, które są potrzebne. Tak więc biblioteka może w ogólności zawierać wiele definicji funkcji i zmiennych, używanych przez umieszczone w niej funkcje, a niewidocznych dla użytkownika. Program użytkowy, który odwołuje się do takiej biblioteki, będzie zawierał minimalną liczbę dołączonych definicji.

5.6.1. Pliki nagłówkowe i funkcje

Wydawać by się mogło, że oto mamy receptę na konstruowanie programów wieloplikowych, w których zapewniono integralność danych i niezawodne wywołania potrzebnych funkcji. Tak jednak nie jest, o czym łatwo się przekonać na naszym prostym przykładzie. Zauważmy przede wszystkim, że – po założeniu biblioteki – jedynymi informacjami dotyczącymi sposobu wywołania funkcji bibliotecznych będą prototypy tych funkcji, zadeklarowane przed blokiem main(). Podczas kompilacji wywołania z bloku będą sprawdzane na zgodność z prototypami. Ewentualna pomyłka w którejś z tych deklaracji, bądź przypadkowe jej usunięcie będzie każdorazowo związane z dodatkowymi operacjami "wyciągania" definicji funkcji z biblioteki. Po drugie, przy pracy zespołowej nad dużym systemem może się zdarzyć, że implementacja którejś z naszych funkcji zostanie zmieniona włącznie z nagłówkiem, określającym sposób jej wywołania. Jeżeli plik z tą zmienioną definicją zostanie ponownie skompilowany i dołączony do biblioteki, to prototyp tej funkcji w pliku main.c (MAIN.CPP) przestanie być aktualny...
Nie bez znaczenia jest również fakt, że dla oddzielnego kompilowania naszych funkcji `czytaj()`, `sortuj()` i `pisz()` musieliśmy dopisać w każdym pliku źródłowym dyrektywę #include <iostream.h>, zwiększając objętość tekstu.

Dostatecznie pewnym, a przy tym prostym remedium na wyliczone niedostatki jest uzupełnienie naszego programu o własny plik nagłówkowy, np. o nazwie sorth.h. W pliku tym umieścimy deklaracje każdej nazwy, używanej więcej niż w jednym pliku źródłowym. Jeżeli tak przygotowany plik nagłówkowy włączymy dyrektywą #include do każdego pliku źródłowego, to plik ten będzie spełniał rolę interfejsu pomiędzy wchodzącymi w skład programu jednostkami kompilacji (plikami z kodem źródłowym). Prezentowany niżej przykład jest próbą realizacji tej koncepcji.

Przykład 5.18.

```
// Plik CZYTAJ.CPP pod DOS, czytaj.c pod Unix
#include "sorth.h"
void czytaj(double *tc, int rc)
{
 for (int i = 0; i < rc; i++)
    cin >> tc[i];
 cout << endl;
 cout << "Tablica przed sortowaniem: " << endl;
}

// Plik SORTUJ.CPP pod DOS, sortuj.c pod Unix
#include "sorth.h"
void sortuj(double *ts, int rs)
{
 int licznik,i;
 double pomoc;
 do
   {
    licznik = 0;
    for ( i = 0; i < rs - 1; i++)
      {
       if ( ts[i] > ts[i+1])   {
           pomoc = ts[i];
           ts[i] = ts[i+1];
           ts[i+1] = pomoc;
```

```
            ++licznik;          }
        }
    }
 while (licznik);
 cout << "Tablica po sortowaniu: \n";
}

// Plik PISZ.CPP pod DOS, pisz.c pod Unix
#include "sorth.h"
void pisz(double *tp, int rp)
{
 for (int i = 0; i < rp; i++)
    cout << "tp[" << i << "] = "
         << tp[i]  << endl;
}

// Plik MAIN.CPP pod DOS, main.c pod Unix
#include "sorth.h"
//extern void czytaj(double *tc, int rc);
//extern void pisz(double *tp, int rp);
//extern void sortuj(double *ts, int rs);
const int rozmiar = 5;
double tab[rozmiar];
int  main() {
     cout << "Podaj " << rozmiar << " liczb typu double:\n";
     czytaj(tab, rozmiar);
     pisz(tab, rozmiar);
     sortuj(tab,rozmiar);
     pisz(tab, rozmiar);
     return 0;
}

// Plik SORTH.H pod DOS, sorth.h pod Unix
#include <iostream.h>
extern void czytaj(double *tc, int rc);
extern void pisz(double *tp, int rp);
extern void sortuj(double *ts, int rs);
```

Programy wieloplikowe mogą się składać z więcej niż jednego pliku nagłówkowego. Należy jednak być ostrożnym w mnożeniu plików nagłówkowych, ponieważ w pewnym momencie może się okazać, że już nie konsolidator, lecz programista zacznie mieć trudności z decyzją, które pliki nagłówkowe należy włączyć do poszczególnych plików źródłowych. Nie można tu podać jakiejś ogólnej reguły, ponieważ zarówno korzystanie z plików nagłówkowych, jak też ich liczba, zależą od stylu programowania. Można natomiast podać ogólne wskazówki, odnoszące się do zawartości plików nagłówkowych.

W plikach nagłówkowych nie należy umieszczać:

- Definicji zwykłych funkcji, np. char get(){ return *wsk++; }
- Definicji zmiennych, np. int i;
- Definicji złożonych struktur stałych, np. const int tab[] = { /*...*/ };

Z drugiej strony, plik nagłówkowy może zawierać:

- Dyrektywy inkluzji, np. #include < string.h>
- Deklaracje funkcji, np. extern int strlen(const char*);
- Deklaracje zmiennych, np. extern int i;
- Definicje stałych, np. const double pi = 3.1415926;
- Wyliczenia, np. Bool { false, true };
- Definicje typów, np. struct punkt { int i,j; };

- Deklaracje nazw, np. class wektor;
- Szablony, np. template<class T> class A { /* ... */ };
- Komentarze, np. /* Ten plik jest interfejsem dla ... */

5.7. Wskaźniki do funkcji

Jak nietrudno zauważyć, po zdefiniowaniu funkcji możemy na niej wykonywać zaledwie jedną operację, którą jest wywołanie funkcji. Definiując wskaźniki do funkcji, gwałtownie rozszerzamy repertuar możliwych operacji:

- Wskaźniki do funkcji, podobnie jak wskaźniki do zmiennych, pozwalają uzyskać adres funkcji.
- Wskaźniki do funkcji mogą być argumentami formalnymi funkcji i wartościami zwracanymi przez funkcje; mogą być przekazywane do funkcji jako argumenty aktualne w wywołaniach.
- Wskaźniki do funkcji można używać w instrukcjach przypisania; można je również umieszczać w tablicach. Ostatnia możliwość jest wykorzystywana np. w systemach okienkowych, bazujących na X-Window (Unix) i MS-Windows (DOS). Jeżeli tworzymy system okienkowy z rozwijalnymi menu, to jest naturalne, że kolejne pozycje menu powinny być ułożone w tablicę. Ponieważ pozycje te są odwołaniami do pewnych operacji, zatem będzie to tablica wskaźników do funkcji.

Omawianie wskaźników do funkcji zaczniemy od najprostszych przykładów. Deklaracja

```
void (*wskv)();
```

wprowadza wskaźnik wskv typu `void (*)()` do jeszcze nieokreślonej funkcji typu **void** o pustym wykazie parametrów. Jeżeli zdefiniujemy trzy funkcje z takimi właśnie sygnaturami

```
void f1() { cout  << "To jest pierwsza funkcja.\n"; };
void f2() { cout  << "To jest druga funkcja.\n";  };
void f3() { cout  << "To jest trzecia funkcja.\n";  };
```

to wskaźnikowi wskv możemy przypisać adres dowolnej z tych funkcji, np.

```
wskv  =  &f2;
wskv  =  &f3;
```

Wskaźnik `wskv` można także bezpośrednio zainicjować adresem funkcji w jego deklaracji:

```
void  (*wskv)()  =  &f1;
```

Funkcje `f1()`, `f2()`, `f3()` można w programie wywoływać bezpośrednio, np.

```
f1();
f2();
f3();
```

lub pośrednio poprzez wskaźniki. W tym drugim przypadku istniejące kompilatory dopuszczają składnię wywołań o dwóch postaciach. Jeżeli wskaźnikowi `wskv` przypisano np. adres funkcji `f1()`, to wywołanie może mieć postać:

```
(*wskv)();
```

lub

```
wskv();
```

Zanotujmy wniosek, który można już wysnuć z powyższych przykładów: wskaźniki do funkcji muszą mieć takie same sygnatury (liczbę i typy argumentów) i typy zwracane, jak wskazywane przez nie funkcje.

Prezentowane niżej przykłady ilustrują wymienione wyżej notacje wywołań dla wskaźnika
`void (*wskv)()`
do funkcji
`void ff()`
oraz wskaźnika
`int (*wski)(int)`
do funkcji
`int abs(int).`

Przykład 5.19.

```
#include <iostream.h>
void ff();
void (*wskv)();
void main()  {
      cout << "Wywolanie bezposrednie ff:\n";
      ff();
      cout << "Wywolanie posrednie ff:\n";
      wskv = &ff;
       (*wskv)();
}

void ff()
{
 cout << "To jest funkcja ff" << endl;
}
```

Przykład 5.20.

```
#include <iostream.h>
int abs(int);
int (*wski)(int);
int  main() {
      cout << "Wywolanie bezposrednie abs:\n";
      cout << abs(5) << endl;
      wski = &abs;
      cout << "Wywolanie posrednie abs:\n";
      cout << wski(-10) << endl;
      return 0;
}

int abs(int a)
{
 if (a > 0) return a;
 else return -a;
}
```

W następnym przykładzie zadeklarowano wskaźnik `void (*wskc)(char*)` typu `void(*)(char*)` do funkcji `void ff(char*)`. Zwróćmy uwagę na brak operatora adresu ('**&**') w instrukcji przypisania `wskc = ff;`. Jest to dopuszczalne składniowo i analogiczne, jak w przypadku tablic. Przypomnijmy, że podczas kompilacji nazwa tablicy (bez deklaratora []) jest automatycznie przekształcana na wskaźnik (adres) do jej pierwszego elementu. Podobnie nazwa funkcji – bez operatora wywołania ()– jest przekształcana na wskaźnik do tej funkcji. Np. nazwa funkcji `void ff(char*)`, tj. `ff`, jest przekształcana na wskaźnik bez nazwy, którego typem jest

`void (*)(char*)`. Ponieważ wskaźnik `(*wskc)(char*)` jest tego samego typu, zatem przypisanie

```
wskc = ff
```

jest uzasadnione.

Przykład 5.21.

```
#include <iostream.h>
void ff(char*);
void (*wskc)(char*);
void main()  {
     ff("Hello! ");
     wskc = ff;
     wskc("Goodbye! ");
}

void ff(char* str)
{
 cout << str << endl;
}
```

5.7.1. Synonimy nazw typów

Zdaje się nie ulegać wątpliwości, że mimo sygnalizowanych korzyści z wprowadzenia wskaźników do funkcji, sposób ich deklarowania jest raczej uciążliwy. N.b. podobne kłopoty notacyjne sprawiają wskaźniki, kojarzone z typem tablicowym. Np. deklaracja

```
int* wskt[5];
```

jest czytelna: wprowadza ona tablicę 5 wskaźników `wskt` do typu **int** (`wskt` jest typu int*). Natomiast deklaracja

```
int (*wsktab)[10];
```

jest już mniej czytelna: wprowadza ona wskaźnik `wsktab` do tablicy o 10 elementach typu **int** (`wsktab` jest typu `(*)[]`).

W języku C++ istnieje mechanizm, który pozwala zdefiniować nową nazwę dla istniejącego typu. Mechanizm ten polega na poprzedzeniu deklaracji, w której występuje nazwa typu, słowem kluczowym (specyfikatorem) **typedef**. Np. dla typów podstawowych o długich nazwach możemy wprowadzić nazwy-synonimy

```
typedef  unsigned char  uchar;
typedef  unsigned long int ulong;
```

a następnie deklarować zmienne, posługując się nowymi nazwami

```
uchar  znak1, znak2;
ulong  ul1, ul2;
```

W jednej deklaracji, poprzedzonej przez **typedef**, można umieścić kilka nazw-synonimów, np.

```
typedef  int  droga, dystans,  *rozmiar;
```

a następnie deklarować zmienne (typu **int** oraz **int***) z użyciem nowych nazw:

```
dystans  km, m;
rozmiar  rozx, rozy;
```

Takie deklaracje stosuje się często w celu ukrycia szczegółów implementacji, co jest bardzo istotne w systemach obiektowych.

Podobne deklaracje można stosować w deklaracjach, wprowadzających wskaźniki do funkcji (nie wolno ich używać w definicjach funkcji). Np. deklaracje

```
typedef  int  (*wski)(char*);
typedef  void (*wskc)(double*, int);
```

wprowadzają zastępcze (i o wiele prostsze) nazwy `wski` oraz `wskc` dla typów "wskaźnik do funkcji". Otrzymane w ten sposób nazwy-synonimy można wielokrotnie wykorzystywać dla deklarowania zmiennych tych typów, np.

```
wski  wsk1, wsk2, wsk3;
wskc  wsf1, wsf2;
```

Podobnie jak dla typów podstawowych, nazwy zastępcze mogą służyć do ukrywania informacji, a ponadto "ułatwiają życie" programiście.

Przykład 5.22.

```
#include <iostream.h>
extern int min(int*, int);
typedef int (*wski)(int*, int);
const int rozmiar = 5;
int tab[rozmiar] = { 2, 5, 9, 4, 6 };

int  main() {
     wski wsk1, wsk2;
     cout << "Wywolanie bezposrednie min: "
          << min(tab,rozmiar) << endl;
     wsk1 = min;
     wsk2 = wsk1;
     cout << "Wywolanie posrednie min: "
          << wsk2(tab,rozmiar) << endl;
     return 0;
}
```

Definicja funkcji `min (int*, int)` może mieć postać:

```
int min(int* tab, int r)
{
 int pomoc = tab[0];
 for (int i = 1; i < r; ++i)
   if (pomoc > tab[i])
       pomoc = tab[i];
 return pomoc;
}
```

5.8. Przeciążanie funkcji

W językach proceduralnych każda funkcja musi mieć unikatową nazwę. Jest to zasadne, jeżeli różne funkcje wykonują różne operacje. Jednak w przypadku funkcji, które wykonują podobne operacje na różnych obiektach programu, byłoby korzystnym nadać im tę samą nazwę.
Funkcje można wtedy zróżnicować unikatowymi sygnaturami, tj. typami i/lub liczbą argumentów. Składnia języka C++ dopuszcza taki sposób deklarowania i definiowania funkcji; nazywa się go *przeciążeniem nazwy funkcji* lub krócej *przeciążeniem funkcji*. Obowiązują przy tym następujące zasady:

1. Funkcje przeciążone (o takiej samej nazwie) muszą mieć taki sam zasięg (np. bloku, pliku, programu).
2. Funkcje, które różnią się tylko typem zwracanym, nie mogą mieć takiej samej nazwy.
3. Funkcje o takiej samej nazwie muszą się różnić sygnaturami i mogą się różnić typem zwracanym.
4. Jeżeli argumenty dwóch funkcji różnią się tylko tym, że jedna ma argument typu T, a druga T&, to funkcje te nie mogą mieć takiej samej nazwy. Wynika to stąd, że zarówno T, jak i T& są inicjowane tym samym zbiorem wartości i wywołanie nie potrafi ich rozróżnić.
5. Dwie funkcje, których argumenty różnią się tylko tym, że jedna ma argument typu T, a druga const T, nie mogą mieć takiej samej nazwy. Przyczyna jest analogiczna, jak dla argumentów typu T i T&.

Przykład 5.23.

```
#include <iostream.h>
extern int min(int, int);
extern double min(double, double);
extern int min(int, int, int);
int  main() {
     cout << min( 2, 7 ) << endl;
     cout << min( 2.5, 7.5 ) << endl;
     cout << min( 9, 6, 4 ) << endl;
     return 0;
}
```

Definicje kolejnych funkcji `min()` mogą mieć postać:

```
int min( int a, int b ) { return a < b ? a : b; }

double min(double x, double y)
{ return x < y ? x : y; }

int min(int u, int v, int w)
{
 if (u < v) return u < w  ? u : w;
 else return v < w  ? v : w;
}
```

Dyskusja. W wywołaniu `min(2,7)` liczba i typy argumentów aktualnych są porównywane z liczbą i typami argumentów formalnych kolejnych prototypów funkcji `min()`, zadeklarowanych przed blokiem funkcji `main()`. Po dopasowaniu przez kompilator prototypu `int min(int,int)` konsolidator włączy do pliku z kodem wykonalnym definicję tej funkcji. Analogicznie będą przetwarzane pozostałe dwa wywołania.

Tak oto wprowadziliśmy *funkcje przeciążone*. Jest to realizacja następującej zasady: jeden ogólny interfejs, wiele metod (implementacji). Funkcje przeciążone zalicza się do metod o wiązaniu wczesnym, ponieważ cała informacja adresowa, potrzebna do ich wywołania, jest znana po

zakończeniu kompilacji. Dzięki temu wywołania funkcji o wiązaniu wczesnym (nie tylko przeciążonych) należą do najszybszych, ponieważ narzut czasowy, związany z ich wywołaniem, jest minimalny. Zauważmy też, że przeciążanie funkcji pozwala na rozszerzanie środowiska programowego C++ w miarę potrzeb.

5.8.1. Dopasowanie argumentów

W fazie kompilacji programu, zawierającego wywołania funkcji przeciążonych, uruchamiana jest dość złożona procedura dopasowania argumentów. Procedura ta ma na celu możliwie najlepsze dopasowanie argumentów wywołania do argumentów formalnych i w rezultacie wybranie odpowiedniej funkcji. Szukane dopasowanie ma być najlepsze w tym sensie, że wybrana funkcja musi mieć przynajmniej jeden argument lepiej dopasowany niż każda z pozostałych, możliwych do zaakceptowania funkcji.
Proces dopasowania argumentów jest to najkrótszy ciąg przekształceń (konwersji) typu argumentu (-ów) aktualnego w typ argumentu(-ów) formalnego. W ciągu tym dopuszcza się co najwyżej jedną konwersję jawną, tj. zadaną przez programistę. W procesie dopasowania mają miejsce zestawione niżej reguły.

1. **Dopasowanie dokładne.** Typy argumentów aktualnych i formalnych są te same lub przechodzą w siebie drogą konwersji trywialnych: typ T w T&, typ T& w typ T[] w T*, typ T w const T, typ T*, w const T*, typ T(argumenty) w T(*)(argumenty).
2. **Dopasowanie z promocją.** Jeżeli nie powiedzie się dopasowanie dokładne, to kompilator stosuje wszelkie możliwe promocje. Dla typów całkowitych, argumenty typu **char**, **unsigned char**, **enum** oraz **short int**, są promowane do typu **int**. Jeżeli rozmiar
   `sizeof(short int) <= sizeof(int),`
   to argument typu **unsigned short int** jest promowany do typu **int**. Dla typów zmiennopozycyjnych stosowana jest specjalna promocja z **float** do **double**. Pokazane niżej deklaracje funkcji przeciążonych i ich wywołania ilustrują niektóre promocje.

   Przykład 5.24.

```
#include <iostream.h>
void ff(char);
void ff(int);
void ff(unsigned int);
void ff(float);
void ff(double);

void     main() {
         ff('A');   // dopasowuje ff(char)
         ff(25);    // dopasowuje ff(int)
         ff(25u);   // dopasowuje ff(unsigned int)
         ff(3.14);  // dopasowuje ff(double)
         ff(3.14F); // dopasowuje ff(float)
}

void ff(char c) { cout << "char\n";  }
void ff(int i) { cout << "int\n";  }
void ff(unsigned int u) { cout << "unsigned\n"; }
void ff(double d) { cout << "double\n";  }
void ff(float f) { cout << "float\n";  }
```

3. **Konwersje standardowe.** Jeżeli nie powiedzie się dopasowanie z promocją, to kompilator próbuje zastosować standardowe konwersje. Są to konwersje niejawne typu każdego argumentu aktualnego do typu odpowiedniego argumentu formalnego.

Przykład 5.25.

```
void ff(char*);
void ff(double);
void ff(void*);

void konwersje()
{
 ff("A"); // dopasowuje ff(char*)
 int i = 65;
 ff(&i);  // dopasowuje ff(void*)
 ff('A'); // dopasowuje ff(double)
}
```

4. **Konwersje jawne.** Jeżeli nie powiodą się wszystkie poprzednie próby dopasowania argumentów, to kompilator zastosuje konwersje zdefiniowane przez programistę. Konwersje te mogą być kombinowane z promocjami i standardowymi konwersjami niejawnymi. Każda sekwencja konwersji może zawierać co najwyżej jedną konwersję zdefiniowaną przez programistę.

5. **Niejednoznaczność**. Dla deklaracji, jak w przykładzie poniżej, nie można rozstrzygnąć, do której z funkcji przeciążonych odnoszą się podane wywołania. Kompilator generuje komunikat o błędzie.

Przykład 5.26.

```
void ff(char);
void ff(unsigned char);
void ff(void*);
void niejednoznaczne()
{
 ff(65); /* niejednoznaczne:
            ff(char) lub ff(unsigned char) */
 ff(0); //niejednoznaczne: ff(char) lub ff(void*)
}
```

6. **Brak dopasowania.** Argument aktualny nie da się dopasować do argumentu formalnego. Kompilator generuje komunikat o błędzie.

Przykład 5.27.

```
extern void zamiana(int*, int*);

void brak()
{
  int i, j;
  zamiana(i, j); // brak dopasowania
}
```

5.8.2. Adresy funkcji przeciążonych

Podobnie jak dla funkcji bez przeciążania nazwy, możemy deklarować wskaźniki do funkcji i przypisywać im adresy funkcji przeciążonych. W pokazanym niżej przykładzie zadeklarowano dwie funkcje o nazwie `znaki`; pierwsza z nich, `znaki(int)`, wyprowadza na ekran zadaną liczbę spacji, zaś druga, `znaki(int, char)` – zadaną liczbę znaków różnych od spacji (tutaj 10 znaków 'x').

Przykład 5.28.

```
#include <iostream.h>
void znaki(int licznik)
{ for( ; licznik; licznik-- ) cout << ' ';  }
void znaki(int licznik, char znak)
{ for( ; licznik; licznik-- ) cout << znak; }
int  main() {
     void (*wsk1) ( int );
     void (*wsk2) ( int, char );
     wsk1 = znaki;
     wsk2 = znaki;
     wsk1(5);
     cout << "|\n";
     wsk2(10, 'x');
     cout << "|\n";
     return 0;
}
```

Wydruk z programu ma postać:

```
  |
xxxxxxxxxx|
```

Dyskusja. W programie zadeklarowano dwa wskaźniki, `wsk1` i `wsk2`, do funkcji typu **void**. Wykonanie pierwszej instrukcji przypisania

```
wsk1 = znaki;
```

dopasowuje do nazwy `znaki` funkcję `znaki(int)`, zaś wykonanie instrukcji `wsk2=znaki;` dopasowuje do nazwy `znaki` funkcję `znaki(int,char)`. W instrukcjach **for** obu funkcji opuszczono instrukcję inicjującą, ponieważ zmienna sterująca licznik jest inicjowana w wywołaniach funkcji.

7. Klasy

Klasa jest kluczową koncepcją języka C++, realizującą abstrakcję danych na bardzo wysokim poziomie. Odpowiednio zdefiniowane klasy stawiają do dyspozycji użytkownika wszystkie istotne mechanizmy programowania obiektowego: ukrywanie informacji, dziedziczenie, polimorfizm z wiązaniem późnym, a także szablony klas i funkcji. Klasa jest deklarowana i definiowana z jednym z trzech słów kluczowych: **class**, **struct** i **union**. Chociaż na pierwszy rzut oka może to wyglądać na rozwlekłość, to jednak, jak pokażemy później, jest ona zamierzona i celowa.
Elementami składowymi klasy mogą być struktury danych różnych typów, zarówno podstawowych, jak i zdefiniowanych przez użytkownika, a także funkcje dla operowania na tych strukturach. Dostęp do elementów klasy określa zbiór reguł dostępu.

7.1. Deklaracja i definicja klasy

Klasa jest typem definiowanym przez użytkownika. Deklaracja klasy składa się z nagłówka, po którym następuje ciało klasy, ujęte w parę nawiasów klamrowych; po zamykającym nawiasie klamrowym musi wystąpić średnik, ewentualnie poprzedzony listą zmiennych. W nagłówku klasy umieszcza się słowo kluczowe **class** (lub **struct** albo **union**), a po nim nazwę klasy, która od tej chwili staje się nazwą nowego typu.
Klasa może być deklarowana:
- Na zewnątrz wszystkich funkcji programu. Zakres widzialności takiej klasy rozciąga się na wszystkie pliki programu.
- Wewnątrz definicji funkcji. Klasę taką nazywa się *lokalną*, ponieważ jej zakres widzialności nie wykracza poza zasięg funkcji.
- Wewnątrz innej klasy. Klasę taką nazywa się *zagnieżdżoną*, ponieważ jej zakres widzialności nie wykracza poza zasięg klasy zewnętrznej.

Deklaracji klas nie wolno poprzedzać słowem kluczowym **static**.
Przykładowe deklaracje klas mogą mieć postać:

```
class Pusta {};
class Komentarz { /* Komentarz  */};
class Niewielka { int n; };
```

Wystąpienia klasy deklaruje się tak samo, jak zmienne innych typów, np.

```
Pusta pusta1, pusta2;
Niewielka nw1, nw2;
Niewielka* wsk = &nw1;
```

Zmienne `pusta1`, `pusta2`, `nw1`, `nw2` nazywają się *obiektami*, zaś `wsk` jest wskaźnikiem (zmienną typu `Niewielka*`) do typu `Niewielka`, zainicjowanym adresem obiektu `nw1`. N.b. łatwo można się przekonać, że obiekty klasy `Pusta` mają niezerowy rozmiar.
Klasy w rodzaju `Pusta` i `Komentarz` używa się często jako klasy-makiety podczas opracowywania programu.
Jak już wspomniano, dopuszczalna jest również deklaracja obiektów bezpośrednio po deklaracji klasy, np.

```
class Gotowa { double db; char znak; } ob1, ob2;
```

Dostęp do składowej obiektu danej klasy uzyskuje się za pomocą operatora dostępu ".", np. `nw1.n`; dla zmiennej wskaźnikowej używa się operatora "->", np. `wsk->n`, przy czym ostatni zapis jest równoważny `(*wsk).n`.
Przy deklarowaniu składowych obowiązują następujące ograniczenia:
- deklarowana składowa nie może być inicjowana w deklaracji klasy;

- nazwy składowych nie mogą się powtarzać;
- deklaracje składowych nie mogą zawierać słów kluczowych **auto**, **extern** i **register**; natomiast mogą być poprzedzone słowem kluczowym **static**;

Deklaracje elementów składowych klasy można poprzedzić etykietą **public:**, **protected:**, lub **private:**. Jeżeli sekwencja deklaracji elementów składowych nie jest poprzedzona żadną z tych etykiet, to kompilator przyjmuje domyślną etykietę **private:**. Oznacza to, że dana składowa może być dostępna jedynie dla funkcji składowych i tzw. funkcji zaprzyjaźnionych klasy, w której jest zadeklarowana. Wystąpienie etykiety **public:** oznacza, że występujące po niej nazwy deklarowanych składowych mogą być używane przez dowolne funkcje, a więc również takie, które nie są związane z deklaracją danej klasy. Znaczenie etykiety **protected:** przedstawimy przy omawianiu dziedziczenia.
Ze względu na sterowanie dostępem do składowych klasy, słowa kluczowe **public**, **protected** i **private** nazywa się *specyfikatorami dostępu*.

Przykład 6.1.

```
#include <iostream.h>
class Niewielka  {
 public:
  int n;
};
void main() {
      Niewielka nw1, nw2, *wsk;
      nw1.n = 5;
      nw2.n = 10;
      wsk = &nw1;
      cout << nw1.n << endl;
      cout << wsk ->n << endl;
      wsk = &nw2;
      cout << (*wsk).n << endl;
}
```

Dyskusja. W klasie Niewielka zadeklarowano tylko jedną zmienną składową `n` typu **int**. Ponieważ jest to składowa publiczna, można w obiektach `nw1` i `nw2` bezpośrednio przypisywać jej wartości 5 i 10. Zwróćmy jeszcze uwagę na instrukcję deklaracji wskaźnika `wsk` do klasy `Niewielka`: wskaźnikiem jest identyfikator `wsk`, a nie `*wsk`. Jak widać z przykładu, wskaźnikowi do danej klasy możemy przypisywać kolejno dowolne obiekty tej klasy. Operator dostępu do zmiennej składowej `n` dla wskaźnika `wsk` jest w programie zapisywany w obu równoważnych postaciach: jako `wsk->n` i jako `(*wsk).n`. W drugiej postaci konieczne były nawiasy, ponieważ operator dostępu do składowej "." ma wyższy priorytet niż operator dostępu pośredniego "*".

Zakres widzialności zmiennych składowych obejmuje cały blok deklaracji klasy, bez względu na to, w którym miejscu bloku znajduje się ich punkt deklaracji.
Jeżeli klasa zawiera funkcje składowe, to ich deklaracje muszą wystąpić w deklaracji klasy. Funkcje składowe mogą być jawnie deklarowane ze słowami kluczowymi **inline**, **static** i **virtual**; nie mogą być deklarowane ze słowem kluczowym **extern**. W deklaracji klasy można również umieszczać definicje krótkich (1-2 instrukcje) funkcji składowych; są one wówczas traktowane przez kompilator jako funkcje rozwijalne, tj. tak, jakby były poprzedzone słowem kluczowym **inline**. W programach jednoplikowych dłuższe funkcje składowe deklaruje się w nawiasach klamrowych zawierających ciało klasy, zaś definiuje się bezpośrednio po zamykającym nawiasie klamrowym.
Deklaracja klasy wraz z definicjami funkcji składowych (i ewentualnie innymi wymaganymi definicjami) stanowi *definicję klasy*.

Funkcje składowe klasy mogą operować na wszystkich zmiennych składowych, także prywatnych i chronionych. Mogą one również operować na zmiennych zewnętrznych w stosunku do definicji klasy.

Przykład 6.2.

```
#include <iostream.h>
class Punkt  {
 public:
  int x,y;
  void init(int a, int b) {x = a; y = b; }
  void ustaw(int, int);
};
void Punkt::ustaw(int a, int b)
  { x = x + a; y = y + b; }
int  main() {
      Punkt punkt1;
      Punkt* wsk = &punkt1;
      punkt1.init(1,1);
      cout << punkt1.x << endl;
      cout << wsk -> y << endl;
      punkt1.ustaw(10,15);
      cout << punkt1.x << endl;
      cout << (*wsk).y << endl;
      return 0;
}
```

Dyskusja. W klasie `Punkt` zarówno atrybuty `x`, `y`, jak i funkcje `init()` oraz `ustaw()` są publiczne, a więc dostępne na zewnątrz klasy. Funkcja `init()` służy do zainicjowania obiektu `punkt1` klasy `Punkt`. Jej definicja wewnątrz klasy jest równoważna definicji poza ciałem klasy o postaci:

```
inline void Punkt::init(int a, int b) {x = a; y = b; }
```

Dwuargumentowy (binarny) operator zasięgu "::" poprzedzony nazwą klasy ustala zasięg funkcji składowych `ustaw()` i `init()`. Napis `Punkt::` informuje kompilator, że następująca po nim nazwa jest nazwą funkcji składowej klasy `Punkt`; wywołanie takiej funkcji w programie może mieć miejsce tylko tam, gdzie jest dostępna deklaracja klasy `Punkt`.
Zwróćmy również uwagę na sposób zapisu wywołania funkcji składowej: zapis ten ma taką samą postać, jak odwołanie do atrybutu `x`, bądź `y`.

7.1.1. Autoreferencja: wskaźnik this

Funkcje składowe są związane z definicją klasy, a nie z deklaracjami obiektów tej klasy. Pociąga to za sobą następujące konsekwencje:

- Istnieje tylko jeden "egzemplarz" kodu definicji danej funkcji składowej, będącej "własnością" klasy.
- Kod ten nie wchodzi w skład żadnego obiektu.
- W każdym wywołaniu funkcji składowej klasy musi być wskazany obiekt wołający.

Wskazanie obiektu wołającego jest realizowane przez przekazanie do funkcji składowej ukrytego argumentu aktualnego, którym jest wskaźnik do tego obiektu. Wskaźnikiem tym jest inicjowany niejawny argument – wskaźnik – w definicji funkcji składowej. Do wskaźnika tego można się również odwoływać jawnie, używając słowa kluczowego **this**. Tak więc mamy tutaj autorekursję: poprzez wskaźnik **this** odwołujemy się do obiektu, dla którego wywoływana jest pewna operacja (wywołanie funkcji). W podanym niżej przykładzie pokazano dwie równoważne deklaracje, przy czym druga z nich ilustruje możliwość jawnego użycia wskaźnika **this**.

Przykład 6.3.

```
class Niejawna  {
int m;
int funkcja() { return m; }
};

class Jawna  {
int m;
int funkcja() { return this->m; }
};
```

Ponieważ **this** jest słowem kluczowym, nie może być jawnie deklarowany jako zmienna składowa. Można natomiast wydedukować, że w funkcji `funkcja()` klasy `Niejawna` wskaźnik **this** jest niejawnie zadeklarowany jako

```
Niejawna* const this;
```

a inicjowany adresem obiektu, dla którego wołana jest funkcja składowa `funkcja()`. Ponieważ wskaźnik **this** jest deklarowany jako `*const`, czyli jako wskaźnik stały, nie może on być zmieniany. Można natomiast wprowadzać zmiany we wskazywanym przez niego obiekcie.

Zobaczmy jeszcze, jak wygląda zastosowanie wskaźnika **this** w nieco zmodyfikowanym programie, zawierającym definicję klasy `Punkt`.

Przykład 6.4.

```
#include <iostream.h>
class Punkt  {
public:
    int init(int a, int b)
    { this->x = a; this->y = b; return x; }
    int ustaw(int, int);
private:
    int x, y;
};
int Punkt::ustaw(int a, int b)  {
 this->x = this->x + a; this->y = y + b; return x ;
}
int  main() {
     Punkt punkt1;
     cout << punkt1.init(0,0) << endl;
     cout << punkt1.ustaw(10,15) << endl;
     return 0;
}
```

Z pokazanych wyżej przykładów widać, że jawnych odwołań do wskaźnika **this** nie opłaca się używać, skoro poprawny syntaktycznie jest krótszy zapis. Wyjątkami od tego zalecenia są sytuacje, gdy jawne odwołanie do wskaźnika **this** jest nie tylko opłacalne, ale i konieczne; są to definicje funkcji, które operują na elementach tzw. list jedno- i dwukierunkowych.

7.1.2. Statyczne elementy klasy

Ciało klasy może zawierać deklaracje struktur danych i funkcji poprzedzone słowem kluczowym **static**, będącym, podobnie jak **auto**, **register** i **extern**, specyfikatorem klasy pamięci. Słowo **static** (a także **extern**) może poprzedzać jedynie deklaracje zmiennych i funkcji oraz unii anonimowych; nie może poprzedzać deklaracji funkcji wewnątrz bloku ani deklaracji argumentów formalnych funkcji. W ogólności specyfikator **static** ma dwa znaczenia, które są często mylone. Pierwsze z nich mówi kompilatorowi: przydziel danej wielkości ustalony adres w pamięci i zachowuj go przez cały czas

trwania programu. Drugie mówi o zasięgu: ustal zakres widzialności danej wielkości począwszy od miejsca, w którym została zadeklarowana. Inaczej mówiąc: wielkość zadeklarowana jako **static** istnieje przez cały czas wykonania programu, ale jej zasięg zależy od miejsca deklaracji. Obydwa znaczenia odnoszą się do statycznych zmiennych składowych, natomiast tylko drugie z nich odnosi się do statycznych funkcji składowych.

Przykład 6.5.

```
static int nn;
int  main() {
     nn = 10;
     return 0;
}
```

Dyskusja. W powyższym przykładzie zmienna globalna `nn` istnieje przez cały czas wykonania programu, ale jej zakres widzialności jest ograniczony do pliku, w którym została zadeklarowana (tutaj plik z funkcją `main()`). Oznacza to, że jest dopuszczalne używanie nazwy `nn` w innych plikach programu i w innym znaczeniu. Zmienna `nn` ma zasięg od punktu deklaracji do końca pliku. Gdybyśmy umieścili deklarację zmiennej `nn` np. po bloku main, to każda próba wykonania na niej operacji wewnątrz bloku byłaby błędem syntaktycznym. Zmienną `nn` można inicjować w jej deklaracji; ponieważ tutaj tego nie uczyniono, kompilator nada jej domyślną wartość początkową 0.
Zmienne składowe klasy są – używając terminologii języka Smalltalk – *zmiennymi wystąpienia* (ang. instance variables); dla każdego obiektu danej klasy tworzone są oddzielne ich kopie. Deklaracje takich zmiennych są jednocześnie ich definicjami; są to zmienne lokalne o zasięgu klasy i przypadkowych wartościach inicjalnych (deklarowane w bloku klasy zmienne nie mogą być tam inicjowane).
Jeżeli deklarację zmiennej składowej poprzedzimy specyfikatorem **static**, to taka deklaracja staje się deklaracją referencyjną, a wprowadzona nią zmienna Ä *zmienną klasy*. Konsekwencją tego jest fakt, że definicję takiej zmiennej musimy umieścić na zewnątrz deklaracji klasy. Statyczna zmienna klasy podlega tym samym regułom dostępu (sterowanym etykietami **public:**, **protected:** i **private:**) co zmienna wystąpienia. Przy definiowaniu zmiennej statycznej można jej nadać wartość początkową różną od zera; na czas tej operacji ograniczenia dostępu zostają wyłączone. W rezultacie otrzymamy zmienną o własnościach obiektu globalnego i zasięgu pliku, skojarzoną z samą klasą, a nie z jakimkolwiek jej wystąpieniem, tj. obiektem. Ponieważ zmienna klasy jest związana z klasą, a nie z jej obiektami, można na niej wykonywać operacje w ogóle bez tworzenia obiektów. W tym sensie jest ona samodzielnym obiektem. Jeżeli jednak zadeklarujemy obiekty, to każdy z nich będzie miał dostęp do tego samego adresu w pamięci, który został przydzielony przez kompilator statycznej zmiennej składowej. Tak więc mamy tylko jedną kopię zmiennej składowej, dostępną bez istnienia obiektów, bądź współdzieloną przez zadeklarowane obiekty.
Reasumując: statyczna zmienna klasy jest to zmienna o zasięgu ograniczonym do klasy, w której jest zadeklarowana. Wprowadzenie statycznych zmiennych składowych pozwala uniknąć używania zewnętrznych zmiennych globalnych wewnątrz klasy, co często daje niepożądane efekty uboczne i narusza zasadę ukrywania informacji.
Składnię deklaracji, definicji i operacji na zmiennej statycznej ilustruje poniższy przykład.

Przykład 6.6.

```
#include <iostream.h>
class Test  {
public:
  int m;
  static int n; // Tylko deklaracja
};
// Definicja obiektu statycznego n
int Test::n = 10;
void main() {
      Test::n = 25;
      cout << "Test::n= " << Test::n << endl;
      Test t1,t2;
      t1.m = 5; t2.m = 0;
      cout << "t1.m= " << t1.m << endl;
      cout << "t2.m= " << t2.m << endl;
      cout << "t2.n= " << t2.n << endl;
}
```

Wydruk z programu ma postać:

Test::n= 25
t1.m= 5
t2.m= 0
t2.n= 25

Dyskusja. W deklaracji klasy `Test` mamy deklarację zmiennej statycznej `static int n;`. Definicję tej zmiennej, o postaci `int Test::n = 10;` umieszczono bezpośrednio po deklaracji klasy. Pierwszą instrukcją w bloku funkcji `main()` jest instrukcja przypisania `Test::n = 25;`. Przypisaną do zmiennej statycznej wartość 25 wyprowadzają na ekran instrukcje:
`cout << "Test::n= " << Test::n << endl;`
oraz `cout << "t2.n= " << t2.n << endl;`
Obiekty `t1` i `t2` mają swoje prywatne kopie zmiennej `m` oraz dostęp do jednej kopii zmiennej statycznej `n`.

☞*Uwaga 1. Typ statycznej zmiennej składowej nie obejmuje nazwy jej klasy; tak więc typem zmiennej Test::n jest int.*

☞*Uwaga 2. Statyczne składowe klasy lokalnej (zadeklarowanej wewnątrz bloku funkcji) nie mają określonego zakresu widoczności i nie mogą być definiowane na zewnątrz deklaracji klasy. Wynika stąd, że klasa lokalna nie może mieć zmiennych statycznych.*

☞*Uwaga 3. Zauważmy, że słowo static nie jest ani potrzebne, ani dozwolone w definicji statycznej składowej klasy. Gdyby było dopuszczalne, to mogłaby powstać kolizja pomiędzy znaczeniem static stosowanym do składowych klasy, a znaczeniem, stosowanym do globalnych obiektów i funkcji.*

Słowo kluczowe **static** może również poprzedzać deklarację (nie definicję!) funkcji składowej klasy. Taka statyczna funkcja składowa może operować jedynie na zmiennych statycznych – nie ma dostępu do zmiennych wystąpienia (zwykłe, niestatyczne funkcje składowe mogą operować zarówno na zmiennych niestatycznych, jak i statycznych). Wynika to stąd, że statyczna funkcja składowa nie ma wskaźnika **this**, tak że może ona mieć dostęp do zmiennych niestatycznych swojej klasy jedynie przez zastosowanie operatorów selekcji "." lub "->" do obiektów klasy.

Przykład 6.7.

```
#include <iostream.h>
class Punkt  {
public:
  static int init( int ); // Deklaracja init()
private:
  static int x; // Deklaracja x
};
int Punkt::x; // Definicja x
// Definicja init()
int Punkt::init( int a)  { x = a; return x; }
int  main() {
     cout << Punkt::init(10) << endl;
     return 0;
}
```

Dyskusja. Przykład pokazuje dowodnie, że dostęp do zmiennej statycznej i wywołanie funkcji statycznej danej klasy nie wymagają istnienia obiektów tej klasy.

7.2. Konstruktory i destruktory

Konstruktory i destruktory należą do grupy specjalnych funkcji składowych. Grupa ta obejmuje: konstruktory i destruktory inicjujące, konstruktor kopiujący oraz tzw. funkcje operatorowe.
Konstruktor jest funkcją składową o takiej samej nazwie, jak nazwa klasy. Nazwą destruktora jest nazwa klasy, poprzedzona znakiem tyldy (~).
Każda klasa zawiera konstruktor i destruktor, nawet gdy nie są one jawnie zadeklarowane. Zadaniem konstruktora jest konstruowanie obiektów swojej klasy; jego wywołanie w programie pociąga za sobą:
- alokację pamięci dla obiektu;
- przypisanie wartości do niestatycznych zmiennych składowych;
- wykonanie pewnych operacji porządkujących (np. konwersji typów).

Jeżeli w klasie nie zadeklarowano konstruktora i destruktora, to zostaną one wygenerowane przez kompilator i automatycznie wywołane podczas tworzenia i destrukcji obiektu.

Przykład 6.8.

```
#include <iostream.h>
class Punkt  {
public:
 Punkt(int, int);
 void ustaw(int, int);
private:
 int x,y;
};

Punkt::Punkt(int a, int b)
{  x = a; y = b;  }
void Punkt::ustaw( int c, int d)
{  x = x + c; y = y + d;  }
int  main()  {
     Punkt punkt1(3,4);
     return 0;
}
```

Dyskusja. Ponieważ klasa `Punkt` zawiera dwie prywatne zmienne składowe: `x` oraz `y`, celowym jest zdefiniowanie konstruktora, który nadaje im wartości początkowe `a` oraz `b`. Wartości te (3 i 4) są przekazywane w wywołaniu konstruktora w bloku funkcji `main()`. Korzyści z wprowadzenia

konstruktora są oczywiste; w poprzednich przykładach, w których występowała klasa `Punkt`, musieliśmy zapisywać w bloku funkcji `main()` dwie instrukcje: instrukcję deklaracji obiektu, a następnie instrukcję wywołania funkcji `init()` dla przypisania zmiennym żądanych wartości. Teraz te dwie operacje wykonuje jedna instrukcja deklaracji

```
Punkt punkt1(3,4);
```

Instrukcja ta jest równoważna instrukcji

```
Punkt punkt1 = Punkt(3,4);
```

która woła konstruktor `Punkt::Punkt(int, int)` dla zainicjowania atrybutów obiektu `punkt1` wartościami 3 i 4.
Ponieważ klasa `Punkt` nie zawiera destruktora, obiekt `punkt1` zostanie zniszczony przez destruktor wywoływany w chwili zakończenia programu, a wygenerowany w fazie kompilacji.

W następnych definicjach konstruktorów będziemy najczęściej wykorzystywać notację, zapożyczoną przez C++ z języka Simula. Dla zainicjowania zmiennej jednego z typów podstawowych wartość inicjalną możemy umieścić po znaku równości, bądź w nawiasach okrągłych, np.

```
int i = 7;
int i(7);
```

Wykorzystując drugą z równoważnych notacji możemy zapisać definicję konstruktora klasy Punkt w postaci:

```
Punkt::Punkt(int a, int b):x(a),y(b)  {}
```

W powyższej definicji po nagłówku funkcji mamy listę zmiennych składowych z wartościami inicjalnymi w nawiasach okrągłych, poprzedzoną pojedynczym dwukropkiem. Ponieważ ten prosty konstruktor nie wykonuje niczego więcej poza inicjowaniem zmiennych składowych, jego blok jest pusty.

7.2.1. Własności konstruktorów i destruktorów

Konstruktory i destruktory posiadają szereg charakterystycznych własności i podlegają pewnym ograniczeniom. Zacznijmy od ograniczeń.

1. Deklaracja konstruktora i destruktora nie może zawierać typu zwracanego (nawet **void**). W przypadku konstruktora jawna wartość zwracana nie jest wskazana, ponieważ jest on przeznaczony do tworzenia obiektów, czyli przekształcania pewnego obszaru pamięci w zorganizowane struktury danych. Gdyby dopuścić do zwracania jakiejś wartości, byłby to fragment informacji o implementacji obiektu, która powinna być niewidoczna dla użytkownika. Podobne argumenty dotyczą destruktora.
2. Nie jest możliwy dostęp do adresu konstruktora i destruktora. Zasadniczym powodem jest ukrycie szczegółów fizycznej alokacji pamięci, które powinny być niewidoczne dla użytkownika.
3. Deklaracje konstruktora i destruktora nie mogą być poprzedzone słowami kluczowymi **const**, **volatile** i **static**. (Modyfikator **volatile** wskazuje, że obiekt może być w każdej chwili zmodyfikowany i to nie tylko instrukcją programu użytkownika, lecz także przez zdarzenia zewnętrzne, np. przez procedurę obsługi przerwania). Deklaracja konstruktora nie może być poprzedzona słowem kluczowym **virtual**.
4. Konstruktory i destruktory nie są dziedziczone.
5. Destruktor nie może mieć parametrów formalnych. Parametrami formalnymi konstruktora nie mogą być zmienne składowe własnej klasy.

Wyliczymy teraz kolejno główne własności konstruktorów i destruktorów.

1. Konstruktory i destruktory generowane przez kompilator są **public**. Konstruktory definiowane przez użytkownika również powinny być **public**, aby istniała możliwość tworzenia obiektów na zewnątrz deklaracji klasy. W pewnych przypadkach szczególnych konstruktory mogą wystąpić po etykiecie **protected:**, a nawet **private:**.
2. Konstruktor jest wołany automatycznie, gdy definiuje się obiekt; destruktor – gdy obiekt jest niszczony.
3. Konstruktory i destruktory mogą być wywoływane dla obiektów **const** i **volatile**.
4. Z bloku konstruktora i destruktora mogą być wywoływane funkcje składowe ich klasy.
5. Deklaracja destruktora może być poprzedzona słowem kluczowym **virtual**.
6. Konstruktor może zawierać referencję do własnej klasy jako argument formalny. W takim przypadku jest on nazywany *konstruktorem kopiującym*. Deklaracja konstruktora kopiującego dla klasy `X` może mieć postać `X(const X&);` lub `X(const X&, int=0);`. Jest on wywoływany zwykle wtedy, gdy definiuje się obiekt, inicjowany przez wcześniej utworzony obiekt tej samej klasy, np.
   `X ob2 = ob1;`.

Konstruktor jest wołany wtedy, gdy ma być utworzony obiekt jego klasy. Obiekt taki może być tworzony na wiele sposobów:
- jako zmienna globalna;
- jako zmienna lokalna;
- przez jawne użycie operatora **new**;
- jako obiekt tymczasowy;
- jako zmienna składowa, zagnieżdżona w innej klasie.

Wprawdzie konstruktory i destruktory nie mogą być funkcjami statycznymi, ale mogą operować na statycznych zmiennych składowych swojej klasy. W podanym niżej przykładzie wykorzystano zmienną statyczną `licznik` do rejestrowania istniejących w danej chwili obiektów. Klasę `Status` można uważać za fragment podsystemu zarządzania pamięcią programu.

Przykład 6.9.

```
#include <iostream.h>
class Status  {
public:
  Status() { licznik++; }
  ~Status() { licznik--}
  int odczyt() { return licznik; }
private:
 static licznik; //deklaracja
};
int Status::licznik = 0; // Definicja
int  main() {
      Status ob1,ob2, ob3;
      cout << "Mamy " << ob1.odczyt() << " obiekty\n";
      Status *wsk;
      wsk = new Status; //Alokuj dynamicznie
      if (!wsk)    {
      cout << "Nieudana alokacja\n";
      return 1;   }
      cout << "Mamy " << ob1.odczyt()
           << " obiekty po alokacji\n";
// skasuj obiekt
      delete wsk;
      cout << "Mamy " << ob1.odczyt()
           << " obiekty po destrukcji\n";
```

```
      return 0;
}
```

Dyskusja. Wygląd ekranu po wykonaniu programu będzie następujący:

Mamy 3 obiekty
Mamy 4 obiekty po alokacji
Mamy 3 obiekty po destrukcji

Obiekty programu są tworzone przez kolejne wywołania konstruktora `Status(){licznik++;}`. Najpierw są tworzone obiekty `ob1`, `ob2` i `ob3`, a następnie obiekt dynamiczny `*wsk`. Po destrukcji wskaźnika `wsk` mamy trzeci wiersz wydruku. Oczywiście pierwsze trzy obiekty zostaną zniszczone przy wyjściu z programu przez trzykrotne wywołanie tego samego co dla `wsk` destruktora `~Status() { licznik--}`.

7.2.2. Przeciążanie konstruktorów

Konstruktory, podobnie jak "zwykłe" funkcje (także funkcje składowe klasy), mogą być przeciążane. Najczęściej ma to na celu tworzenie obiektów inicjowanych zadanymi wartościami zmiennych składowych, lub też obiektów bez podanych wartości inicjalnych. Dzięki temu można zadeklarować więcej niż jeden konstruktor dla danej klasy. Wywołanie konstruktora w deklaracji obiektu pozwala kompilatorowi ustalić, w zależności od liczby i typów podanych argumentów, którą wersję konstruktora należy wywołać.
W podanym niżej przykładzie obiektowi `punkt1` nadano wartości początkowe (3 i 4), zaś obiektowi `punkt2` – nie. Gdyby usunąć z programu konstruktor domyślny `Punkt()` o pustym wykazie argumentów, program nie zostałby skompilowany, ponieważ brakłoby konstruktora, który mogłby być dopasowany do wywołania `Punkt punkt2`.

Przykład 6.10.

```
#include <iostream.h>
class Punkt {
public:
 Punkt()  {}
 Punkt(int a, int b): x(a), y(b) {}
 int x,y;
};
int  main() {
      Punkt punkt1(3,4);
      cout << punkt1.x << endl;
      Punkt punkt2;
      cout << punkt2.x << endl;
      return 0;
}
```

7.2.3. Konstruktory z argumentami domyślnymi

Alternatywą dla przeciążania konstruktora jest wyposażenie go w argumenty domyślne, podobnie jak czyniliśmy to w stosunku do zwykłych funkcji. Przypomnijmy, że argument domyślny jest przyjmowany przez kompilator wtedy, gdy w wywołaniu funkcji nie podano odpowiadającego mu argumentu aktualnego.
Wartości domyślne argumentów formalnych w deklaracji (nie w definicji!) zapisujemy w następujący sposób:

– Jeżeli w deklaracji konstruktora umieszczono nazwy argumentów formalnych, to po nazwie argumentu piszemy znak równości, a po nim odpowiednią wartość, np. `Punkt ( int a = 0, int b = 0 );`

– Jeżeli deklaracja nie zawiera nazw argumentów, a jedynie ich typy, to znak równości i następującą po nim wartość piszemy po identyfikatorze typu, np. `Punkt ( int = 0, int = 0 );`

Przykład 6.11.

```
#include <iostream.h>
class Punkt  {
public:
   int x,y;
   Punkt(int = 0, int = 0);
};

Punkt::Punkt(int a, int b): x(a), y(b)
{
 if (a==0 && b==0)
 cout << "Obiekt bez inicjowania...\n";
 else cout << "Obiekt z inicjowaniem...\n";
}

int  main() {
      Punkt punkt1(3,4);
      cout << punkt1.x << endl;
      Punkt punkt2;
      cout << punkt2.x << endl;
      return 0;
}
```

Dyskusja. Powyższy program zawiera wywołania, identyczne jak w przypadku konstruktorów przeciążonych, ale tylko jedną definicję konstruktora. W definicji konstruktora `Punkt()` z zerowymi wartościami domyślnymi umieszczono instrukcję **if** jedynie w tym celu, aby pokazać, że instrukcja deklaracji `Punkt punkt1(3,4);` wywołuje konstruktor z inicjowaniem, a instrukcja deklaracji `Punkt punkt2;` wywołuje konstruktor bez inicjowania. Wydruk z programu ma postać:

Obiekt z inicjowaniem...
3
Obiekt bez inicjowania...
0

Zauważmy że konstruktor `Punkt(int = 0, int = 0)` nie jest równoważny konstruktorowi bez argumentów, np. `Punkt()`. Prezentowana niżej wersja klasy `Punkt` zawiera taki właśnie konstruktor bezargumentowy. Przykład nawiązuje także do przeprowadzonej w p. 5.2 dyskusji, w której zwrócono uwagę na związki pomiędzy przeciążaniem funkcji, a używaniem argumentów domyślnych.

Przykład 6.12.

```
#include <iostream.h>
class Punkt  {
public:
  Punkt(): x(0), y(0)    {}
  Punkt(int a, int b): x(a), y(a) {}
  int fx() { return x; }
  int fy() { return y; }
private:
  int x,y;
};
int   main() {
      Punkt punkt1;
      cout << "punkt1.x= " << punkt1.fx() << "\n";
      Punkt punkt2(3,2);
      cout << "punkt2.x= " << punkt2.fx() << "\n";
      Punkt *wsk = new Punkt(2,2);
      if (!wsk) return 1;
      cout << "wsk->fx()=" << wsk->fx() << "\n";
      if (wsk)
      {
       delete wsk;
       wsk = NULL; }
      return 0;
}
```

Dyskusja. W klasie `Punkt` zmienne składowe `x`, `y` są teraz prywatne, w związku z czym zadeklarowano publiczny interfejs w postaci dwóch funkcji składowych `fx()` oraz `fy()`, które dają dostęp do tych zmiennych. Obiekty `punkt1` i `punkt2` są alokowane na stosie funkcji `main()`, natomiast obiekt wskazywany przez `wsk` – na kopcu (w pamięci swobodnej). Zwróćmy uwagę na instrukcję **if** w bloku `main()`, w której występuje symbol `NULL`, oznaczający wskaźnik pusty. Zapis `if(wsk)`, równoważny `if(wsk == NULL)` jest testem, który zapobiega próbie niszczenia tego samego obiektu dwa razy (jeżeli `wsk == NULL`, to żadna z dwóch instrukcji w bloku **if** nie zostanie wykonana).

7.3. Przeciążanie operatorów

Wśród zdefiniowanych w języku C++ operatorów występują operatory polimorficzne. Możemy tu wyodrębnić dwa rodzaje polimorfizmu:

- **Koercję**, gdy dopuszcza się, że argumenty operatora mogą być mieszanych typów. Ten rodzaj polimorfizmu jest charakterystyczny dla operatorów arytmetycznych: +, -, *, /; np. operator "+" może służyć do dodania dwóch liczb całkowitych, liczby całkowitej do zmiennopozycyjnej, etc.
- **Przeciążenie operatora**, gdy ten sam symbol operatora stosuje się w operacjach nie związanych semantycznie. Typowymi przykładami mogą być operatory inicjowania i przypisania, oznaczane tym samym symbolem "=", lub symbole ">>" oraz "<<", które, zależnie od kontekstu, są bitowymi operatorami przesunięcia lub operatorami wprowadzania/wyprowadzania.

Wymienione rodzaje wbudowanego polimorfizmu występują w wielu nowoczesnych językach programowania. Wspólną dla nich cechą jest to, że są one predefiniowane, a ich definicje stanowią integralną i niezmienną część definicji języka.

Użytkownikowi języka C++ dano znacznie większe możliwości, spotykane w nielicznych współczesnych językach programowania. Ma on do dyspozycji mechanizm, który pozwala tworzyć nowe definicje dla istniejących operatorów. Dzięki temu programista może tak zmodyfikować działanie operatora, aby wykonywał on operacje w narzucony przez niego sposób. Jest to szczególnie istotne w programach czysto obiektowych. Np. obiekty wielokrotnie redefiniowanej klasy Punkt możemy uważać za wektory w prostokątnym układzie współrzędnych. W geometrii i fizyce dla takich

obiektów są np. określone operacje dodawania i odejmowania. Byłoby wskazane zdefiniować podobne operacje dla deklarowanych przez nas klas. Wykorzystamy do tego celu następującą ogólną postać definicji tzw. funkcji operatorowej, tj. funkcji, która definiuje pewien operator "@":

```
typ  klasa::operator@(argumenty)
{  // wykonywane operacje }
```

gdzie słowo `typ` oznacza typ zwracany przez funkcję operatorową, słowo `klasa` jest nazwą klasy, w której funkcja definiująca operator jest funkcją składową, dwa dwukropki oznaczają operator zasięgu, zaś symbol "@" będzie zastępowany w konkretnej definicji przez symbol operatora (np. =, ==, +, ++, **new**). Nazwa funkcji definiującej operator składa się ze słowa kluczowego **operator** i następującego po nim symbolu operatora; np., jeżeli jest przeciążany operator "+", to nazwą funkcji będzie operator+.

Przeciążenie operatora przypomina przeciążenie funkcji. I faktycznie jest to przypadek szczególny przeciążenia funkcji. Poniżej zestawiono ważniejsze reguły przeciążania i własności operatorów przeciążanych.

1. Operator `@` jest zawsze przeciążany względem klasy, w której jest zadeklarowana jego funkcja `operator@`. Zatem w innych kontekstach operator nie traci żadnego ze swych oryginalnych znaczeń, ustalonych w definicji języka, natomiast zyskuje znaczenia dodatkowe.
2. Funkcja definiująca operator musi być albo funkcją składową klasy, albo mieć co najmniej jeden argument będący obiektem lub referencją do obiektu klasy (wyjątkiem są funkcje, które redefiniują znaczenie operatorów **new** i **delete**). Ograniczenie to gwarantuje, że wystąpienie operatora z argumentami typów nie będących klasami, jest jednoznaczne z wykorzystaniem jego standardowej, wbudowanej w język definicji. Jego intencją jest rozszerzanie języka, a nie zmiana wbudowanych definicji.
3. Nie mogą być przeciążane operatory "`.`", "`.*`", "`::`", "`?:`", "`sizeof`" oraz symbole "`#`" i "`##`".
4. Nie jest możliwa zmiana priorytetu, reguł łączności, ani liczby argumentów operatora.
5. Funkcje definiujące operatory, za wyjątkiem funkcji `operator=()`, są dziedziczone.
6. Przeciążany operator nie może mieć argumentów domyślnych.
7. Funkcje: `operator=()`, `operator[]()`, `operator()` i `operator->()` nie mogą być statycznymi funkcjami składowymi.
8. Funkcja definiująca operator nie może posługiwać się wyłącznie wskaźnikami.
9. Operatory: "=" (przypisania), "`&`" (pobrania adresu) i "`,`" (przecinkowy) mają predefiniowane znaczenia w odniesieniu do obiektów klas (o ile nie zostały na nowo zdefiniowane).

Za wyjątkiem operatorów wymienionych wyżej, możemy przeciążać wszystkie operatory wbudowane w język – zarówno operatory jednoargumentowe (unarne), jak i dwuargumentowe (binarne). Podane niżej proste przykłady ilustrują przeciążanie kilku takich operatorów.

Przykład 6.13.

```
// Operator dodawania +
#include <iostream.h>
class Punkt  {
public:
  Punkt(): x(0),y(0){}
  Punkt(int a, int b): x(a),y(a) {}
  int fx()  { return x; }
  int fy()  { return y; }
  Punkt operator+(Punkt);
private:
  int x,y;
};
Punkt Punkt::operator+(Punkt p)
{
 return Punkt(x + p.x, y + p.y);
}
int  main() {
      Punkt punkt1, punkt2, punkt3;
      punkt1 = Punkt(2,2);
      punkt2 = Punkt(3,1);
      punkt3 = punkt1 + punkt2;
      cout << "punkt1.x= " << punkt1.fx() << endl;
      cout << "punkt3.x= " << punkt3.fx() << endl;
      int i = 10, j;
      j = i + 15; // Predefiniowany '+'
      cout << "j= " <<  j  << endl;
      return 0;
}
```

Wydruk z programu ma postać:

punkt1.x= 2
punkt3.x= 5
j= 25

Dyskusja. Instrukcja `Punkt punkt1, punkt2, punkt3;` wywołuje trzykrotnie konstruktor `Punkt()`. Instrukcja przypisania
`punkt1 = Punkt(2,2);`
wywołuje konstruktor `Punkt(int,int)`. Następnie kompilator generuje niejawny operator przypisania, który przypisuje obiektowi `punkt1` obiekt, świeżo utworzony przez konstruktor `Punkt(int,int)`. Tak samo jest wykonywana druga instrukcja przypisania. Instrukcja dodawania
`punkt3 = punkt1 + punkt2;`
woła najpierw generowany przez kompilator konstruktor kopiujący, który tworzy kopię obiektu `punkt2` i przekazuje ją jako parametr aktualny do funkcji operatorowej `Punkt Punkt::operator+(Punkt p)`. Konieczność wykonania tej operacji byłaby bardziej oczywista, gdybyśmy instrukcję dodawania obiektów zapisali w drugiej dopuszczalnej postaci
`punkt3 = punkt1.operator+(punkt2);`
Następnie jest wywoływana funkcja operatorowa "+", która z kolei woła konstruktor `Punkt(int,int)`, aby utworzyć obiekt, będący "sumą" obiektów `punkt1` i `punkt2`. Końcową operacją w tej instrukcji jest wywołanie generowanego przez kompilator operatora przypisania "=", aby przypisać wynik do obiektu `punkt3`.
Omawiane niejawne operacje przypisania i kopiowania moglibyśmy prześledzić, uzupełniając definicję klasy `Punkt` o własne konstrukcje programowe. I tak, operator przypisania dla klasy `Punkt` mógłby mieć postać:

```
Punkt& Punkt::operator=(const Punkt& p)
{ this->x = p.x; this->y = p.y; return *this  }
```
zaś konstruktor kopiujący:
```
Punkt::Punkt(const Punkt& p) { x = p.x; y = p.y; }
```
Dalsze instrukcje są wykonywane w sposób standardowy i nie wymagają komentarzy. Zauważmy jedynie, że predefiniowany operator "+" nie stracił swojego znaczenia, co pokazano na operacjach ze zmiennymi `i` oraz `j`.

Przykład 6.14.

```
// Operator relacyjny ==
#include <iostream.h>
class Punkt  {
public:
  Punkt(): x(0),y(0) {}
  Punkt(int a, int b): x(a),y(a) {}
  int fx()  { return x; }
  int fy()  { return y; }
  int operator==(Punkt);
private:
  int x,y;
};
int Punkt::operator==(Punkt p)
{
 if( p.fx() == fx() && p.fy() == fy() )
  return (-1);
 else return (0);
}

int  main() {
      Punkt punkt1, punkt2;
      punkt1 = Punkt(2,2);
      punkt2 = Punkt(3,1);
      if ( punkt1 == punkt2 ) cout << "Rowne\n";
      else cout << "Nierowne\n";
      int i = 5, j = 6, k;
      k = i == j; // predefiniowany '=='
      cout << "k= " << k << endl;
      return 0;
}
```

Dyskusja. Instrukcja deklaracji `Punkt punkt1, punkt2;` wywołuje dwukrotnie bezargumentowy konstruktor `Punkt()`.
Instrukcja `punkt1 = Punkt(2,2);` najpierw woła konstruktor `Punkt::Punkt(int,int)`, a następnie generowany przez kompilator operator przypisania (porównaj poprzedni przykład). Podobnie przebiega wykonanie drugiej instrukcji przypisania. Ponieważ instrukcję **if** można zapisać w postaci:
```
if(punkt1.operator==(punkt2))...
```
zatem, podobnie jak w poprzednim przykładzie, wołany jest niejawny konstruktor kopiujący dla utworzenia kopii obiektu `punkt2`, a następnie funkcja operatorowa "==", która wywołuje funkcję składową `fx()`. Ponieważ składowe `x` obu obiektów są różne (2 i 3), test na równość na tym się kończy, ponieważ pierwszy argument koniunkcji ma wartość zero. Podobnie jak w poprzednim przykładzie pokazano, że operator "==" nie stracił swojego znaczenia, wbudowanego w język. Wydruk z programu ma postać:

Nierowne
k= 0

Przytoczone przykłady ilustrowały przeciążanie operatorów binarnych. Dla operatorów unarnych składnia deklaracji i definicji funkcji operatorowej jest taka sama z tym, że funkcja przeciążająca musi być bezargumentowa. Ilustracją tego jest poniższy przykład.

Przykład 6.15.

```
// Unarne operatory + i -
#include <iostream.h>
class Firma  {
public:
  Firma(char* n, char s): nazwa(n),stan(s) {}
  void operator+() { stan = '1'; }
  void operator-() { stan = '0'; }
  void podaj() { cout << "stan: " << stan << endl;}
private:
  char* nazwa;
  char stan;
};
int  main() {
     Firma frm1("firma1", '1');
     -frm1;
     +frm1;
     frm1.podaj();
     return 0;
}
```

☞*Uwaga. Poprawne syntaktycznie stosowanie operatorów przeciążonych może nie mieć sensu nawet w stosunku do obiektów tej samej klasy, jeżeli obiekt ma być dobrą abstrakcją rzeczywistości fizycznej. Weźmy dla przykładu dwa obiekty klasy* `Potrawa`*; niech każdy z nich ma składową* `char* smak`*. Jeżeli wartość tej składowej w pierwszym obiekcie jest "słony", a w drugim "słodki", to ma sens porównanie (==), ale nie ma sensu np. dodawanie tych dwóch obiektów do siebie. Warto o tym pamiętać przy tworzeniu systemów obiektowych.*

7.3.1. Przeciążanie operatorów new i delete

Przypomnijmy (nigdy za wiele przypomnień), że obiekty języka C++ mogą być alokowane na trzy sposoby:
- na stosie (jest to tzw. pamięć automatyczna, a zmienne w niej lokowane są nazywane zmiennymi automatycznymi lub lokalnymi),
- w pamięci statycznej pod ustalonym adresem,
- w pamięci swobodnej (dynamicznej), zorganizowanej w postaci listy komórek, nazywanej kopcem, stogiem, lub stertą.

Obiektom lokalnym jest przydzielana pamięć na stosie w chwili, gdy wywoływana jest funkcja z niepustym wykazem argumentów, lub w chwili, gdy obiekt jest tworzony w bloku funkcji. Pamięć dla obiektów statycznych jest przydzielana w fazie konsolidacji (po kompilacji, a przed fazą wykonania). Obiekty dynamiczne są alokowane w pamięci przez wywołanie operatora new. Przy tworzeniu obiektów, w każdym z tych przypadków najpierw jest przydzielany odpowiedni obszar pamięci, a następnie jest wołany konstruktor, który inicjuje w tym obszarze obiekt o żądanych własnościach. Wyraźne rozdzielenie alokacji pamięci od inicjowania jest najlepiej widoczne przy alokacji dynamicznej. Np. deklaracja

```
Test* wsk = new Test(10);
```

oznacza, że operator `new` wywołuje pewną (niejawną) procedurę alokacji dla uzyskania pamięci, a następnie woła konstruktor klasy `Test` z parametrem 10, który inicjuje tę pamięć. Ta sekwencja operacji jest nazywana tworzeniem obiektu, czyli wystąpienia klasy.

W większości przypadków użytkownik nie musi się interesować wspomnianą procedurą alokacji. Można jednak wymienić pewne szczególne sytuacje, gdy użytkownik powinien sam decydować o sposobie przydziału pamięci:

- Program tworzy i niszczy bardzo wiele małych obiektów (np. węzły drzewa, powiązania listy jednokierunkowej, punkty, linie, komunikaty). Alokacja i dealokacja takich licznych obiektów może łatwo zdominować czas wykonania programu, a także szybko wyczerpać zasoby pamięci. Duży narzut czasowy jest pochodną niskiej efektywności standardowego alokatora. Narzut pamięciowy jest powodowany fragmentacją pamięci swobodnej przy alokacji mieszaniny obiektów o różnych rozmiarach.
- Program, który musi działać nieprzerwanie przez długi czas przy bardzo ograniczonych zasobach. Jest to sytuacja typowa dla systemów czasu rzeczywistego, które wymagają gwarantowanego kwantum pamięci z minimalnym narzutem.
- Istniejące środowisko nie zapewnia procedur zarządzania pamięcią (np. piszemy program, który ma obsługiwać mikroprocesor bez systemu operacyjnego). W takiej sytuacji programista musi opracować procedury niskiego poziomu, odwołujące się nawet do adresów fizycznych komórek pamięci.

W pierwszych dwóch przypadkach zastosowanie własnych mechanizmów przydziału pamięci może – jak pokazuje praktyka – przynieść poprawę efektywności od dwóch do nawet dziesięciu razy.

Operatory `new` i `delete` (podobnie jak pozostałe operatory standardowe) można przeciążać względem danej klasy. Ich prototypy mają wtedy postać:

```
class X  {
//...
  void* operator new(size_t rozmiar);
  void operator delete(void* wsk);

  void* operator new [] (size_t rozmiar);
  void operator delete [] (void* wsk);
};
```

gdzie typ `size_t` jest zależnym od implementacji typem całkowitym, używanym do utrzymywania rozmiarów obiektów; jego deklaracja znajduje się w pliku nagłówkowym `<stddef.h>`

Pierwsza para operatorów odnosi się do alokacji pojedynczych obiektów, zaś druga do tablic. Ponieważ funkcja `X::operator new()` jest wywoływana przed konstruktorem, jej typ zwracany musi być `void*`, a nie `X*` (jeszcze nie ma obiektu `X`). Natomiast destruktor, wywoływany przed funkcją operatorową `X::operator delete()`, "dekonstruuje" obiekt, pozostawiając do zwolnienia "niezorganizowaną" pamięć. Dlatego argumentem funkcji `X::operator delete()` nie jest `X*`, lecz `void*`. Ponadto definiowane w klasie `X::operator new()` i `X::operator delete()` są statycznymi funkcjami składowymi, bez względu na to, czy są jawnie zadeklarowane ze słowem kluczowym **static**, czy też nie. Własność ta jest konieczna z tych samych powodów, co wymienione wyżej: wywołanie statycznej funkcji składowej klasy nie wymaga istnienia obiektu tej klasy.

W trzecim z wyżej przytoczonych przypadków programista podaje adres fizyczny lub adres symboliczny alokowanego obiektu. Można się wtedy posłużyć prototypem funkcji operatorowej new() z dwoma argumentami i z pokazanym niżej przykładowym blokiem:

```
void* operator new(size_t, void* wsk) { return wsk; }
```

gdzie wskaźnik `wsk` podaje adres, pod którym jest alokowany dany obiekt.

Ponieważ jest to konstrukcja programowa, która może się odwołać bezpośrednio do sprzętu, ustalono dla niej specjalną składnię wywołania. Jeżeli np. adres pamięci jest podany w postaci:

```
void* bufor = (void*) 0xF00F;
```

to wywołanie alokatora `X::operator new()` może mieć postać:

```
X* wskb = new(bufor)X;
```

Zauważmy, że każdy operator `new()` przyjmuje `size_t` jako swój pierwszy argument; zatem w ostatnim wywołaniu rozmiar alokowanego obiektu jest dostarczany niejawnie.

Podany niżej przykład ilustruje składnię deklaracji, definicji i wywołań przeciążonych operatorów `new()` i `delete()`. Jest to jedynie ilustracja, jako że ani nie tworzy się w programie wielu obiektów, ani też nie jest to nawet makieta systemu czasu rzeczywistego. W klasie `Nowa` zadeklarowano zmienną statyczną `licznik`, która służy do zliczania obiektów po konstrukcji i destrukcji. Doliczanie i odliczanie odbywa się odpowiednio w konstruktorze i destruktorze klasy. W bloku `main()` wyjście z pętli **do-while** następuje po wprowadzeniu z klawiatury znaku “q” albo “Q”; niezależność od małej lub dużej litery zapewnia funkcja `toupper(char)`, która jest zadeklarowana w pliku nagłówkowym `ctype.h`>.

Przykład 6.16.

```
# include <iostream.h>
# include < stddef.h>
# include < ctype.h>
class Nowa  {
public:
  char znak_zliczany;
  int liczba_znakow;
  Nowa(char znak);
  ~Nowa() { cout << "Destruktor...\n"; licznik--; }
  void dodaj_znak() { liczba_znakow++; }
  void* operator new(size_t rozmiar);
  void operator delete(void* wsk);
  static licznik;
};
int Nowa::licznik = 0;
Nowa::Nowa(char z)
{
 cout << "Konstruktor...\n";
 znak_zliczany = z;
 liczba_znakow = 0;
 licznik++;
}
void* Nowa::operator new(size_t rozmiar)
{
 cout << "Alokacja: new...\n";
 void* wsk = new char[rozmiar];
 return wsk;
}
void Nowa::operator delete(void* wsk)
```

```
{
 cout << "Dealokacja: delete... ";
 delete (void*) wsk;
 }

int   main() {
      char we;
      Nowa* wskNowa = new Nowa('x');
      cout << "Napisz kilka liter 'x'; 'q'-koniec:\n";
      do    { //wczytujemy iksy
      cin >> we;
      if(we == wskNowa->znak_zliczany)
      wskNowa->dodaj_znak();
            } while(toupper(we) != 'Q');
      cout << "\nLiczba znakow "
            << wskNowa->znak_zliczany << ": ";
      cout << wskNowa->liczba_znakow << endl;
      cout << "Liczba obiektow: " << Nowa::licznik << endl;
      delete wskNowa;
      return 0;
}
```

Przykładowy wydruk może mieć postać:

Alokacja: new...
Konstruktor...
Napisz kilka liter 'x'; 'q' - koniec:
x
x
q

Liczba znakow x: 2
Liczba obiektow: 1
Destruktor...
Dealokacja: delete...

7.4. Funkcje i klasy zaprzyjaźnione

Funkcje składowe klasy można uważać za implementację koncepcji, nazywanej w językach obiektowych *przesyłaniem komunikatów*. Przy wywołaniu takiej funkcji obiekt, dla którego jest wywoływana, pełni rolę odbiorcy komunikatu; wartości zmiennych, adresy, czy też obiekty, przekazywane jako argumenty aktualne funkcji, stanowią treść komunikatu. Np. wywołanie

```
punkt1.ustaw(c,d);
```

można uważać za adresowany do obiektu `punkt1` komunikat o nazwie `ustaw`, którego treścią są wartości dwóch zmiennych: `c` oraz `d`.
Taki styl programowania jest charakterystyczny dla języków czysto obiektowych, jak np. Smalltalk. Styl ten zapewnia pełną hermetyczność (ukrywanie informacji) klas, których obiekty są dostępne wyłącznie za pośrednictwem funkcji składowych, stanowiących ich publiczny interfejs.
Dość często jednak taki sztywny "gorset", pod którym ukrywa się informacje prywatne, okazuje się zbyt ciasny i niewygodny.
Język hybrydowy, jakim jest C++, daje możliwość rozluźnienia tego gorsetu. Pozwala on zwykłym funkcjom i operatorom wykonywać operacje na obiekcie w podobny sposób, jak na "obiektach" typów podstawowych. Przy takim podejściu moglibyśmy naszą funkcję składową ustaw() uczynić

zwykłą funkcją, ze zmienną punkt1 jako jednym z argumentów formalnych. Wtedy jej wywołanie miałoby postać:

```
ustaw(punkt1,c,d);
```

Tutaj `punkt1` jest traktowany na równi z pozostałymi argumentami. Zauważmy jednak, że teraz funkcja `ustaw()` będzie operować na kopii argumentu `punkt1`, a więc nie będzie mogła zmienić wartości zmiennych składowych obiektu `punkt1`. Można temu łatwo zaradzić, przesyłając parametr `punkt1` przez referencję, a nie przez wartość. Co więcej, funkcję `ustaw()` można przeciążać, podając różne definicje, np. dla ustawienia punktu na jednej z osi układu współrzędnych, na płaszczyźnie, czy w przestrzeni trójwymiarowej. Można również pomyśleć o rozszerzeniu definicji funkcji `ustaw()` tak, aby mogła oddziaływać na stan kilku obiektów jednocześnie; np. wywołanie

```
ustaw(punkt1, punkt2, c, d);
```

mogłoby przesyłać wartości `c` oraz `d` z obiektu `punkt1` do obiektu `punkt2`.
W podobnych do opisanego wyżej przypadkach, gdy decydujemy się odsłonić część informacji ukrytych w deklaracji klasy, możemy wykorzystać mechanizm tzw. *funkcji zaprzyjaźnionych* języka C++. Jak sugeruje nazwa, funkcje zaprzyjaźnione mają te same przywileje, co funkcje składowe, chociaż same nie są funkcjami składowymi klasy, w której zadeklarowano je jako zaprzyjaźnione. W szczególności mają one dostęp do tych elementów klasy, których deklaracje poprzedzają etykiety `private:` i `protected:`.
Deklarację funkcji zaprzyjaźnionej (a także klasy zaprzyjaźnionej) poprzedza słowo kluczowe **friend**. Przy tym jest obojętne, czy taką deklarację umieszcza się w publicznej, czy w prywatnej części deklaracji klasy.
Ponieważ funkcja zaprzyjaźniona nie jest funkcją składową klasy, zatem jej lista argumentów nie zawiera ukrytego wskaźnika **this**. Zasięg funkcji zaprzyjaźnionej jest inny niż zasięg funkcji składowych klasy: funkcja zaprzyjaźniona jest widoczna w zasięgu zewnętrznym, tj. takim samym zasięgu, jak klasa, w której została zadeklarowana.

Przykład 6.17.

```
// Funkcja zaprzyjazniona ustaw()
#include <iostream.h>
class Punkt  {
public:
  Punkt(int, int);
  int fx()  { return x; }
  int fy()  { return y; }
  friend void ustaw(Punkt&, int,int);
private:
  int x,y;
};
 Punkt::Punkt(int a, int b): x(a),y(b) {}
 void ustaw(Punkt& p, int c, int d)
 {  p.x += c; p.y += d;  }
int  main() {
     Punkt punkt1(3,4);
     cout << "punkt1.x przed ustaw():" << punkt1.fx()
          << endl;
     ustaw(punkt1, 5, 5);
     cout << "punkt1.x po ustaw():" <<  punkt1.fx()
          << endl;
     return 0;
}
```

Wydruk z programu ma postać:

punkt1.x przed ustaw(): 3
punkt1.x po ustaw(): 8

Dyskusja. Ponieważ funkcja `ustaw()` nie jest funkcją składową, jej definicja nie jest poprzedzona nazwą klasy i operatorem zasięgu. Zauważmy, że w definicji nie występuje specyfikator **friend**. Zwróćmy także uwagę na fakt, że ze względu na brak niejawnego wskaźnika **this**, funkcja zaprzyjaźniona nie może się odwoływać do zmiennych składowych bezpośrednio, lecz poprzez obiekt `p` klasy `Punkt`. Argument aktualny `punkt1` jest przekazywany przez referencję; dzięki temu stan obiektu `punkt1` (tj. wartości jego zmiennych składowych) może być modyfikowany przez funkcję `ustaw()`.

Funkcja składowa jednej klasy może być zaprzyjaźniona z inną klasą; ilustracją tej możliwości jest poniższy ciąg deklaracji:

```
class Pierwsza {
// ...
  void f();
};
class Druga {
// ...
friend void Pierwsza::f();
};
```

Deklaracje funkcji, poprzedzone specyfikatorem **friend** pozwalają także deklarować klasy, które odwołują się do siebie nawzajem. Charakterystycznym przykładem może być deklaracja operatora mnożenia wektora przez macierz, jak pokazano niżej na przykładowym ciągu deklaracji. Zauważmy, że w tym przypadku konieczne jest użycie deklaracji referencyjnej klasy `Wektor` przed właściwymi deklaracjami klas.

```
class Wektor;
class Macierz {
// ...
friend Wektor operator*(Macierz&, Wektor&);
};
class Wektor  {
// ...
friend Wektor operator*(Macierz&, Wektor&);
};
```

7.4.1. Zaprzyjaźniony operator '<<'

Dotychczas nie pomyśleliśmy o tym, jak ułatwić sobie wyprowadzanie stanu obiektu. Wprawdzie dla nasze przykładowej klasy `Punkt` zdefiniowaliśmy funkcje składowe `fx()` i `fy()` dla uzyskania dostępu do zmiennych prywatnych, ale każde wyprowadzenie wartości `x` oraz `y` do strumienia `cout` wymagało pisania oddzielnych instrukcji z odpowiednimi argumentami dla operatora wstawiania "<<". Obecnie wykorzystamy możliwość przeciążenia operatora "<<" dla wyprowadzenia pełnego stanu obiektu jedną instrukcją.
W pliku nagłówkowym `<iostream.h>` znajduje się deklaracja klasy strumieni wyjściowych ostream oraz szereg definicji przeciążających operator "<<", w których pierwszym jego argumentem jest obiekt klasy ostream. Definicje te pozwalały nam używać operatora "<<" do wyprowadzania wartości różnych typów, np. **char**, **int**, **long int**, **double**, czy **char*** (łańcuchów). Projektując własne klasy, użytkownik może wprowadzać własne definicje operatora "<<" (w razie potrzeby także ">>"). Mają one następującą postać ogólną:

```
ostream&  operator<<(ostream&  os, nazwa-klasy&  ob)
{
 // Ciało funkcji operator<<()
 return  os;
}
```

Pierwszym argumentem funkcji `operator<<()` jest referencja do obiektu typu `ostream`. Oznacza to, że `os` musi być strumieniem wyjściowym. Do drugiego argumentu, `ob`, przesyła się w wywołaniu obiekt (adres) typu `nazwa-klasy`, który będzie wyprowadzany na standardowe wyjście. Zauważmy, że strumień wyjściowy `os` musi być przekazywany przez referencję, ponieważ jego wewnętrzny stan będzie modyfikowany przez operację wyprowadzania. Funkcja `operator<<()` zawsze zwraca referencję do swojego pierwszego argumentu, tj. strumienia wyjściowego `os`; ta własność oraz fakt, że operator "<<" wiąże od lewej do prawej, pozwala używać go wielokrotnie w tej samej instrukcji wyprowadzania. Np. w instrukcji

```
cout << ob1 << "\n";
```

wyrażenie jest wartościowane tak, jak gdyby było zapisane w postaci:

```
( cout << ob1 ) << "\n"
```

Wartościowanie wyrażenia w nawiasach okrągłych wstawia `ob1` do `cout` i zwraca referencję do `cout`; ta referencja staje się pierwszym argumentem dla drugiego operatora "<<". Tak więc drugi operator "<<" jest przykładany tak, jak gdyby napisano:

```
cout << "\n";
```

Wyrażenie `cout << "\n"` również zwraca referencję do cout, a więc może po nim wystąpić następny operator" "<<", i.t.d.

Funkcja `operator<<()` nie powinna być funkcją składową klasy, na której obiektach ma operować. Wynika to stąd, że gdyby była funkcją składową, to jej pierwszym z lewej argumentem, przekazywanym niejawnie poprzez wskaźnik **this**, byłby obiekt, który generuje wywołanie tej funkcji. Tymczasem w naszej definicji pierwszym z lewej argumentem musi być strumień klasy `ostream`, natomiast prawy operand jest obiektem, który chcemy wyprowadzić na standardowe wyjście (kolejności tej nie można zmienić, ponieważ taką kolejność argumentów narzucają definicje w `<iostream.h>`). Wobec tego funkcję `operator<<()` musimy zadeklarować jako funkcję zaprzyjaźnioną klasy, na której obiektach ma operować. Ilustruje to pokazany niżej prosty przykład.

Przykład 6.18.

```
// Zaprzyjazniony operator <<
#include <iostream.h>
class Punkt {
 public:
 Punkt(int, int);
 friend ostream& operator<<(ostream&, Punkt&);
 private:
 int x,y;
};
 Punkt::Punkt(int a, int b): x(a),y(b) {}
 ostream& operator<<(ostream& os, Punkt& ob)
 {
  os << ob.x << ", " << ob.y << "\n";
  return os;
 }

int  main() {
     Punkt punkt1(3,4), punkt2(10,15);
     cout << punkt1 << punkt2;
     return 0;
}
```

Wydruk z programu będzie miał postać:

3, 4
10, 15

Dyskusja. Może się wydawać dziwnym, że w definicji operatora << używa się tego samego symbolu. Zauważmy jednak, że operatory << używane w definicji wyprowadzają liczby całkowite i łańcuchy; zatem wywołują one już istniejące w pliku `<iostream.h>` definicje, w których drugim operandem jest liczba typu **int** lub łańcuch (typu **char***). Drugim godnym uwagi faktem jest to, że instrukcja wyprowadzania w definicji operatora << wysyła wartości `x` oraz `y` do zupełnie dowolnego strumienia klasy `ostream`, przekazywanego do funkcji `operator<<()` jako parametr aktualny. W wywołaniu operatora << w bloku `main()` użyliśmy `cout` jako parametru aktualnego. Równie dobrze moglibyśmy jednak skierować wyjście naszego programu do pliku, zamiast na konsolę dołączoną do `cout`. W takim przypadku należałoby wykorzystać definicje, zawarte w pliku nagłówkowym `<fstream.h>`.

7.4.2. Klasy zaprzyjaźnione

Każda klasa może mieć wiele funkcji zaprzyjaźnionych; jest zatem naturalne, aby całą klasę uczynić zaprzyjaźnioną z inną klasą. Jeżeli klasa A ma być zaprzyjaźniona z klasą B, to deklaracja klasy A musi poprzedzać deklarację klasy B. Schemat deklaracji ilustruje poniższy przykład: każda funkcja składowa klasy `Pierwsza` staje się funkcją zaprzyjaźnioną klasy `Druga`. W rezultacie wszystkie składowe prywatne, publiczne, i chronione klasy `Druga` stają się dostępne dla klasy zaprzyjaźnionej `Pierwsza`.

```
class Pierwsza {
// ...
};
class Druga {
public:
 Druga(int i = 0, int j = 0): x(i),y(j) {}
 friend class Pierwsza;
private:
int x, y;
};
```

7.5. Obiekty i funkcje

Prawie wszystkie prezentowane dotąd przykłady klas zawierały dwa rodzaje funkcji składowych:
- Funkcje, które mogły zmieniać wartości zmiennych składowych, tj. stan obiektu. Funkcje takie w językach obiektowych nazywa się operacjami mutacji (ang. mutator operations).
- Funkcje, które jedynie podawały aktualne wartości zmiennych składowych, tj. bieżący stan obiektu. Funkcje takie w językach obiektowych nazywa się operacjami dostępu (ang. accessor operations, lub field accessors); w języku C++ nazywamy je funkcjami stałymi.

Obecnie zajmiemy się bardziej szczegółowo zagadnieniem zmian i zachowania stanu obiektu w aspekcie obu rodzajów funkcji.

7.5.1. Obiekty i funkcje stałe

Obiekty klas, podobnie jak obiekty typów wbudowanych, można deklarować jako stałe symboliczne, poprzedzając deklarację słowem kluczowym **const**.
Np. dla klasy `Punkt` z konstruktorem domyślnym `Punkt()` możemy utworzyć obiekt stały `punkt1` instrukcją deklaracji:
`const Punkt punkt1;`
Do tego samego celu można wykorzystać konstruktor z parametrami:
`const Punkt punkt2(3,2);`
W obu przypadkach kompilator C++ zaakceptuje definicje zmiennych, które zostaną odpowiednio zainicjowane przez konstruktory. Natomiast kompilator odrzuci każdą próbę zmiany stanu obiektów `punkt1` i `punkt2` zarówno przez bezpośrednie przypisanie, jak i przez przypisanie za pomocą wskaźników, np.

```
Punkt p1;
punkt1 = p1;
Punkt* wsk = &punkt1;
```

Gdyby jednak zadeklarowano wskaźnik stały `const Punkt* wsk;` to oczywiście można mu przypisać adres obiektu stałego `punkt1`
`wsk = &punkt1;`

Pozostaje jednak otwarte pytanie: czy wolno dla obiektu stałego wywoływać funkcje (np. funkcję `ustaw()`), które mogą zmienić jego stan? Logika mówi, że takie operacje nie powinny być dopuszczalne.
Dla tak prostej klasy jak `Punkt`, kompilator mógłby prawdopodobnie odróżnić funkcje, które zmieniają wartości zmiennych składowych `x` oraz `y` od funkcji, które pozostawiają je bez zmian. W ogólności jednak nie jest to możliwe. Tak więc w praktyce programista musi pomóc kompilatorowi przez odpowiednie zadeklarowanie i zdefiniowanie tych funkcji, które zachowują stan obiektu. Wyróżnienie takich funkcji jest możliwe przez dodanie słowa kluczowego **const** do ich deklaracji i definicji. Zasięg tych funkcji będzie się pokrywał z zasięgiem klasy, a ich deklaracje mają postać:

```
typ-zwracany  nazwa-funkcji(parametry)  const;
```

Słowo kluczowe **const** musi również pojawić się po nagłówku w definicji funkcji. Np. definicja funkcji stałej `fx()`, umieszczona wewnątrz deklaracji klasy `Punkt` będzie mieć postać:
```
int fx() const { return x; }
```
Podany niżej przykład ilustruje przeprowadzoną dyskusję.

Przykład 6.19.

```
#include <iostream.h>
class Punkt {
public:
 Punkt(): x(0),y(0) {}
 Punkt( int a, int b ): x(a),y(a) {}
 void ustaw(int c, int d)
 { x = x + c; y = y + d; }
 int fx() const { return x; }
 int fy() const { return y; }
private:
 int x,y;
};
int  main() {
     const Punkt p0;
     cout << "p0.x= " << p0.fx() << "\n";
     cout << "p0.y= " << p0.fy() << "\n";
     Punkt p1;
//   p0 = p1; Niedopuszczalne
//   Punkt* wsk = &p0; Niedopuszczalne
//   p0.ustaw(3,4); Niedopuszczalne
     return 0;
}
```

Analiza programu. Niedopuszczalność przypisania `p0 = p1` oraz wywołania `p0.ustaw(3,4);` jest oczywista, ponieważ obiekt `p0` jest obiektem stałym i jego składowe nie mogą być zmieniane w programie przez żadną operację. W drugiej błędnej instrukcji, `Punkt* wsk = &p0;` próbuje się wskaźnikowi przypisać adres obiektu stałego, co, podobnie jak dla stałych typów wbudowanych, jest niedopuszczalne.

☞*Uwaga 1. Niektóre kompilatory (np. Borland C++,v.3.1) akceptują kod źródłowy, w którym funkcje, nie wyróżnione słowem kluczowym const, mogą być wywoływane dla obiektu stałego. Kompilatory te wprawdzie ostrzegają użytkownika, ale i tak produkują kod wykonalny, który zmienia stan obiektów stałych!*

Przypomnijmy jeszcze, że każda niestatyczna funkcja składowa zawiera niejawny argument **this**, który (domyślnie) występuje jako pierwszy z lewej w wykazie argumentów. Domyślna deklaracja tego wskaźnika dla każdej funkcji składowej pewnej klasy `X` ma postać: `X *const this;`. Dla stałych funkcji składowych domyślna deklaracja będzie: `const X *const this;`.

7.5.2. Kopiowanie obiektów

W zasadzie istnieją dwa przypadki, w których występuje potrzeba kopiowania obiektów:
- gdy deklarowany obiekt pewnej klasy inicjuje się innym, wcześniej utworzonym obiektem;
- gdy obiektowi przypisuje się inny obiekt w instrukcji przypisania.

Obiekt jest również kopiowany wtedy, gdy jest przekazywany przez wartość jako parametr aktualny funkcji oraz gdy jest wynikiem zwracanym przez funkcję.
Kopiowanie wystąpień ("obiektów") typów wbudowanych, np. **char**, **int**, etc. oznacza po prostu kopiowanie ich wartości. Przykładowo możemy napisać:
```
int i(10); // to samo co int i = 10;
int j = i;
```
W obu przypadkach operator "=" jest operatorem inicjowania, a nie przypisania.

Dla obiektów klasy, w której nie zdefiniowano specyficznych dla niej operacji, kopiowanie i przypisywanie obiektów tej klasy będzie wykonywane za pomocą generowanego przez kompilator konstruktora kopiującego i generowanego operatora przypisania. Np. dla klasy `X` będą generowane automatycznie: konstruktor kopiujący `X::X(const X&)` oraz operator przypisania
`X& operator=(const X&)`.
Funkcje te wykonują kopiowanie obiektów składowa po składowej. Np. dla klasy:
```
class Punkt { int x,y; public: Punkt(int, int); };
```
możemy zadeklarować obiekt:
```
Punkt p1(3,4);
```
a następnie obiekt `p2`, którego pola `x`, `y` będą inicjowane wartościami pól obiektu `p1`:
```
Punkt p2 = p1;
```
Zwróćmy uwagę na następujący fakt: gdyby deklaracja obiektu `p2` miała postać: `Punkt p2;`, to deklarację klasy Punkt musielibyśmy rozszerzyć o konstruktor domyślny, np. `Punkt() { x = 0; y = 0; }`. Tymczasem poprawna deklaracja `Punkt p2 = p1;` nie wymaga istnienia konstruktora domyślnego. Wniosek stąd taki, że obiekt `p2`, inicjowany w deklaracji obiektem `p1`, nie jest tworzony przez żaden zadeklarowany konstruktor klasy! I tak jest istotnie: obiekt `p2` jest tworzony, składowa po składowej, przez generowany konstruktor kopiujący `Punkt::Punkt(const Punkt& p1)`. Konstruktor ten jest wywoływany niejawnie przez kompilator, przy czym obiekt `p1` przechodzi przez referencję jako parametr aktualny. Zauważmy też, że wprawdzie konstruktor kopiujący operuje bezpośrednio na obiekcie `p1`, a nie na jego kopii, to jednak obiekt `p1` nie ulegnie żadnym zmianom, ponieważ argument formalny konstruktora kopiującego jest poprzedzony słowem kluczowym **const**.

Kopiowanie obiektów przez niejawne wywołanie automatycznie generowanego konstruktora kopiującego podlega, niestety, bardzo istotnemu ograniczeniu. Operacja ta sprawdza się jedynie dla obiektów, które nie zawierają wskazań na inne obiekty. Jeżeli obiekt jest wystąpieniem klasy, w której zadeklarowano wskaźnik do jej własnego obiektu lub obiektu innej klasy, to przy wyżej opisanej procedurze zostaną wprawdzie skopiowane wskaźniki, ale nie będą skopiowane obiekty wskazywane. Dlatego też kopiowanie z niejawnym wywołaniem generowanego przez kompilator konstruktora kopiującego określa się jako *kopiowanie płytkie* (ang. shallow copy). Płytkie kopiowanie ma jeszcze jedną wadę: jeżeli w klasie zdefiniowano destruktor, to po każdej operacji kopiowania zmiennych wskaźnikowych będziemy mieć po dwa wskazania na ten sam adres. Wówczas wywoływany przed zakończeniem programu (lub funkcji) destruktor będzie dwukrotnie niszczył ten sam obiekt! Pokazany niżej przykład ilustruje taki właśnie przypadek.

Przykład 6.20.

```
#include <iostream.h>
class Niepoprawna  {
 public:
  Niepoprawna() { wsk = new int(10); }
  ~Niepoprawna() { delete wsk; }
 private:
  int* wsk;
};
int  main() {
     Niepoprawna z1;
     Niepoprawna z2 = z1;
     return 0;
}
```

Dyskusja. Druga instrukcja deklaracji w bloku `main()` wywołuje generowany konstruktor kopiujący, inicjując obiekt `z2` obiektem `z1`. Wskaźnik `wsk` typu **int*** obiektu `z2` zostaje zainicjowany kopią wskaźnika `wsk` obiektu `z1`. W rezultacie `wsk` obu obiektów wskazują na ten sam

obiekt typu **int** o wartości 10. Przed zakończeniem programu wykonywany jest dwukrotnie destruktor, odpowiednio dla obiektów `z1` i `z2`. Za każdym razem niszczony jest ten sam obiekt typu **int***, co może przynieść niepożądane konsekwencje. Podobna sytuacja powstaje w przypadku kopiowania obiektów przez przypisanie.

Dla większości klas rozwiązaniem ukazanego problemu jest zdefiniowanie własnego konstruktora kopiującego i operatora przypisania. Właściwe zdefiniowanie tych funkcji pozwoli nam na tzw. *kopiowanie głębokie* (ang. deep copy), przy którym będą kopiowane nie tylko wskaźniki, ale i obiekty przez nie wskazywane. Prawidłowo będzie też przebiegać destrukcja obiektów.

Przykład 6.21.

```
#include <iostream.h>
class Poprawna  {
public:
  Poprawna(): wsk(new int(10)) {}
  ~Poprawna() { delete wsk; }
  Poprawna(const Poprawna& nz)
  {  wsk = new int(*nz.wsk);  }
  Poprawna& operator=(const Poprawna& nz)
  {
   if(this != &nz)
    {
     delete wsk;
     wsk = new int(*nz.wsk);
    }
   return *this;
  }
private:
  int* wsk;
};

int  main() {
     Poprawna z1;
     Poprawna z2 = z1;
     return 0;
}
```

Dyskusja. W programie zdefiniowano własny konstruktor kopiujący i własny operator przypisania (nie używany). Pierwsza instrukcja w bloku `main()` wywołuje konstruktor `Poprawna() { wsk = new int(10); }`, który tworzy obiekt `z1`, a w nim podobiekt typu **int**, na który wskazuje `wsk`. Druga instrukcja wywołuje konstruktor kopiujący `Poprawna(const Poprawna&)` z parametrem aktualnym `z1`; konstruktor ten tworzy nowy podobiekt typu **int** i umieszcza go pod innym adresem niż pierwszy (oczywiście oba podobiekty mają tę samą wartość 10). Dzięki temu wywoływany dwukrotnie przed zakończeniem programu destruktor niszczy za każdym razem inny obiekt.

☞*Uwaga. Instrukcję Poprawna z2 = z1; można także zapisać w postaci Poprawna z2(z1);, pokazującej wyraźnie, że obiekt z2 jest inicjowany obiektem z1.*

7.5.3. Przekazywanie obiektów do/z funkcji

Syntaktyka i semantyka przekazywania obiektów do funkcji jest identyczna dla obiektów typów predefiniowanych (np. int, double), jak i obiektów klas definiowanych przez użytkownika. Dla typu zdefiniowanego przez użytkownika parametr formalny funkcji będzie klasą, wskaźnikiem, lub referencją do klasy. Funkcję taką wywołujemy z parametrem aktualnym, będącym odpowiednio obiektem danej klasy, adresem obiektu, lub zmienną referencyjną. Podobnie jak dla typów

wbudowanych, obiekty klas są domyślnie przekazywane przez wartość, a semantyka przekazywania jest taka sama, jak semantyka inicjowania.
Obiekty przekazywane do funkcji tworzone są jako automatyczne, tzn. takie, które tworzy się za każdym razem gdy jest wykonywana ich instrukcja deklaracji, i niszczy za każdym razem, gdy sterowanie opuszcza blok zawierający deklarację. Jeżeli funkcja jest wywoływana wiele razy (co często się zdarza), to tyle samo razy jest wykonywane tworzenie i niszczenie obiektów, a więc za każdym razem mamy nowy obiekt, z nowymi inicjalnymi wartościami zmiennych składowych. Pokazany niżej przykład ilustruje przekazywanie obiektu do funkcji przez wartość.

Przykład 6.22.

```
#include <iostream.h>
class Test  {
public:
 Test(int a): x(a) { cout << "Konstrukcja...\n"; }
 Test(const Test& t)
{ this->x = t.x; cout << "Konstrukcja kopii...\n"; }
 ~Test() { cout << "Destrukcja...\n"; }
 int podaj() { return x; }
 void ustaw(int i) { x = i; }
private:
 int x;
};
void f(Test arg) {
 arg.ustaw(50);
 cout << "Funkcja f: ";
 cout << "t1.x == " << arg.podaj() << '\n';
}
int  main() {
     Test t1(10);
     cout << "t1.x == " << t1.podaj() << '\n';
     f(t1);
     cout << "t1.x == " << t1.podaj() <<'\n';
     return 0;
}
```

Wydruk z programu ma postać:

Konstrukcja...
t1.x == 10
Konstrukcja kopii...
Funkcja f: t1.x == 50
Destrukcja...
t1.x == 10
Destrukcja...

Analiza programu. Wykonanie instrukcji deklaracji `Test t1(10);` tworzy obiekt `t1` za pomocą konstruktora `Test(int)`, od którego pochodzi pierwszy wiersz wydruku. W instrukcji `cout` wywołuje się funkcję składową `podaj()`, która wyświetla drugi wiersz wydruku. Instrukcja wywołania funkcji `f(t1)`, z argumentem przekazywanym przez wartość, wywołuje konstruktor kopiujący
`Test(const Test&)`, który generuje trzeci wiersz wydruku. Czwarty wiersz jest generowany przez funkcję `f()`; wypisuje ona wartość pola `x` lokalnej kopii argumentu, tj. obiektu utworzonego przez konstruktor kopiujący. Zauważmy, że wartość `x` w kopii obiektu została zmieniona na 50 funkcją składową `ustaw()`, wywołaną z bloku funkcji `f()`, co pokazuje czwarty wiersz wydruku.

Po wydrukowaniu czwartego wiersza sterowanie opuszcza blok funkcji `f()`, wywołując destruktor `~Test()`, który niszczy obiekt, utworzony przez konstruktor kopiujący (piąty wiersz z tekstem `Destrukcja...`). Po opróżnieniu stosu funkcji `f()` sterowanie wraca do bloku `main()`, wywołując w instrukcji `cout` funkcję składową `podaj()` obiektu `t1`. Wykonanie tej instrukcji pokazuje, że wartość pola `x` pozostała bez zmiany, jako że zmiana `x`
na 50 była wykonywana przez funkcję `f()` na kopii obiektu `t1`, a nie na samym obiekcie. Ostatni wiersz wydruku sygnalizuje destrukcję obiektu `t1`, gdy sterowanie opuszcza blok funkcji `main()`.
W powyższym przykładzie warto zwrócić uwagę na dwa momenty. Gdy jest tworzona kopia obiektu przekazywanego do argumentu formalnego funkcji, nie wywołuje się konstruktora obiektu, lecz konstruktor kopiujący. Powód jest oczywisty: ponieważ konstruktor jest w ogólności używany do inicjowania obiektu (np. nadania wartości początkowych zmiennym składowym), to nie może być wołany gdy wykonuje się kopię już istniejącego obiektu. Jeżeli kopia ma zostać przesłana do funkcji, to zależy nam przecież na aktualnym stanie obiektu, a nie na jego stanie początkowym. Natomiast jest celowe i konieczne wywołanie destruktora, gdy funkcja kończy działanie. Jest to konieczne, ponieważ obiekt w bloku funkcji mógłby być użyty do jakiejś operacji, która musi zostać unieważniona, gdy obiekt wychodzi poza swój zasięg. Przykładem może być alokacja pamięci dla kopii; pamięć ta musi być zwolniona po wyjściu z bloku funkcji.

Destrukcja kopii obiektu używanej do wywołania funkcji może być źródłem pewnych kłopotów, szczególnie w przypadku dynamicznej alokacji pamięci. Jeżeli np. obiekt użyty jako argument alokuje pamięć na kopcu (ang. heap) i zwalnia ją po destrukcji, to i jego kopia będzie zwalniać tę samą pamięć, gdy zostanie wywołany jej destruktor. Jednym ze sposobów na uniknięcie tego rodzaju "niespodzianek" jest przekazywanie do funkcji adresu obiektu zamiast samego obiektu. Wtedy, co jest oczywiste, nie będzie tworzony żaden nowy obiekt, i nie będzie wołany żaden destruktor przy wyjściu z bloku funkcji. Adresy obiektów można przesyłać do funkcji albo za pomocą wskaźników, albo referencji. Przy tym, jeżeli chcemy uniknąć zmiany argumentu aktualnego przez operacje wykonywane w bloku funkcji, to wystarczy w definicji funkcji zadeklarować argument formalny ze słowem kluczowym **const**. Podany niżej przykład ilustruje wariant ze wskaźnikiem i z referencją.

Przykład 6.23.

```
#include <iostream.h>
class Test  {
public:
 Test(int a): x(a){ cout << "Konstrukcja...\n"; }
 Test(const Test& t)
 { this->x=t.x; cout<<"Konstrukcja kopii...\n"; }
 ~Test() { cout << "Destrukcja...\n"; }
 int podaj() { return x; }
 void ustaw(int i) { x = i; }
private:
 int x;
};

void fwsk(Test* arg)
{ arg->ustaw(50); cout << "Funkcja fwsk: ";
 cout << "arg.x == " << arg->podaj() << '\n';
}

void fref(Test& arg)
{ arg.ustaw(60); cout << "Funkcja fref: ";
 cout << "arg.x == " << arg.podaj() << '\n';
}
int  main() {
     Test t1(10);
```

```
      cout << "t1.x == " << t1.podaj() << '\n';
      fwsk(&t1);
//    fref(t1);
      cout << "t1.x == " << t1.podaj() << '\n';
      return 0;
}
```

Wydruk z programu ma postać:

Konstrukcja...
t1.x == 10
Funkcja fwsk: arg.x == 50
t1.x == 50
Destrukcja...

Dyskusja. Jak pokazuje wydruk, tworzony i niszczony jest tylko jeden obiekt. Argumentem funkcji `fwsk()` jest wskaźnik do obiektu klasy `Test`; wobec tego jej argument aktualny w wywołaniu musi być adresem obiektu tej klasy. Ponieważ w bloku funkcji `fwsk()` jest wywoływana funkcja składowa `ustaw()` zmieniająca wartość `x`, to po wykonaniu funkcji `fwsk()` wartość ta (50) została wyświetlona przez bezpośrednie wywołanie funkcji `podaj()` dla obiektu `t1`. Gdyby w funkcjach `fwsk()` i `fref()` wyeliminować wywołanie funkcji, zmieniającej stan obiektu, to ich prototypy mogłyby mieć postać:

```
void fwsk(const Test* arg);
```

i

```
void fref(const Test& arg);
```

☞*Uwaga 1. W podanych wyżej przykładach konstruktory kopiujące i destruktory wprowadzono dla lepszego zobrazowania wydruków (obiekty nie zawierają wskaźników do innych obiektów). Gdyby ich nie zadeklarowano, kopiowanie i destrukcję wykonałyby funkcje, generowane przez kompilator.*

☞*Uwaga 2. W dobrze skonstruowanym programie małe obiekty mogą być przesyłane przez wartość, a duże przez wskaźniki lub referencje.*

Podobnie jak dla typów wbudowanych, wynikiem prowadzonych w bloku funkcji obliczeń może być obiekt klasy zdefiniowanej przez użytkownika. Typowa definicja takiej funkcji może mieć jedną z postaci:

```
klasa  nazwa-funkcji(klasa obiekt)
{
 //...
 return obiekt;
}
```

lub

```
klasa&  nazwa-funkcji(klasa& obiekt)
{
 //...
 return obiekt;
}
```

W pierwszym przypadku, zarówno przy wywołaniu funkcji, jak i powrocie z funkcji, będzie wywoływany konstruktor kopiujący (domyślny lub zdefiniowany w klasie). W drugim przypadku obiekt będzie przesłany do funkcji przez referencję (lub stałą referencję), i w taki sam sposób przekazany z funkcji. Stosując technikę referencji należy pamiętać o niebezpieczeństwie przekazania referencji do obiektu, który przestaje istnieć po wyjściu z bloku funkcji, np.

```
Test& g() { Test ts; return ts; }
```

Tutaj zmienna lokalna `ts` przestaje istnieć po wykonaniu funkcji, a więc funkcja zwraca "wiszącą referencję" – referencję do nieistniejącego obiektu.

Przykład 6.24.

```
#include <iostream.h>
class Test  {
public:
 Test(int a): x(a) { cout << "Konstrukcja...\n"; }
 Test(const Test&)
 { this->x = t.x; cout << "Konstrukcja kopii...\n"; }
 ~Test() { cout << "DESTRUKCJA...\n"; }
 Test& operator=(const Test& t)
  { this->x = t.x;
    cout << "Przypisanie...\n";
    return *this;
  }
  int podaj() { return x; }
private:
  int x;
};

Test g(Test arg)
{
 cout << "Funkcja g: ";
 cout << "arg.x == " << arg.podaj() << '\n';
 return arg;
}
int  main() {
     Test t1(10), t2(20);
     t1 = g(t2);
     cout << "t1.x == " << t1.podaj() << '\n';
     return 0;
}
```

Wydruk z programu ma postać:

Konstrukcja...
Konstrukcja...
Konstrukcja kopii...
Funkcja g: arg.x == 20
Konstrukcja kopii...
DESTRUKCJA...
Przypisanie...
DESTRUKCJA...
t1.x == 20
DESTRUKCJA...
DESTRUKCJA...

Analiza programu. Pierwsze dwa wiersze wydruku to (jedyne) dwa wywołania konstruktora `Test(int)`. Wykonanie instrukcji `t1 = g(t2);` pociąga za sobą następującą sekwencję czynności:

– wywołanie konstruktora kopiującego (`Konstrukcja kopii...`**),** który tworzy obiekt tymczasowy dla parametru `arg`
– wywołanie funkcji `g()` z argumentem utworzonym przez konstruktor kopiujący

– wyświetlenie napisu: `Funkcja g: arg.x == 20`, w którym liczba 20 jest wynikiem wywołania funkcji składowej `podaj()`
– wywołanie konstruktora kopiującego dla utworzenia obiektu tymczasowego (wyrażenia, przekazywanego przez instrukcję return)
– destrukcję pierwszego obiektu tymczasowego
– przypisanie wartości funkcji `g()` do `t1` wykonywane przez wywołanie przeciążonego operatora przypisania
– destrukcję drugiego obiektu tymczasowego
– wyświetlenie odpowiedzi z wywołania funkcji składowej `podaj()`
– destrukcję obiektów `t1` i `t2`.

7.5.4. Konwersje obiektów

Kompilacja i wykonanie programów w języku C++ prawie zawsze wymaga wykonania wielu konwersji typów. Zdecydowana większość tych konwersji wykonywana jest niejawnie, bez udziału programisty. Typowym przykładem może być proces dopasowania argumentów funkcji, w szczególności argumentów funkcji przeciążonych. Oczywiście programista może dokonywać konwersji jawnych za pomocą operatora konwersji "()", ale należy to czynić raczej oszczędnie i tylko w przypadkach naprawdę koniecznych.
Natomiast w dotychczasowej dyskusji o klasach i obiektach klas w zasadzie nie zwracaliśmy uwagi na fakt, że konwersja towarzyszy kreowaniu obiektów. Weźmy najbliższy przykład z p. 6.5.3. Utworzenie obiektu klasy `Test` wymagało użycia konstruktora `Test::Test(int y)`, który zmienną `y` typu **int** przekształcał w obiekt typu `Test`. Przykład ten można uogólnić: *jednoargumentowy konstruktor klasy X można traktować jako przepis, który z argumentu konstruktora tworzy obiekt klasy X*. Zauważmy, że argument konstruktora nie musi być typu wbudowanego; może nim być zmienna innej klasy, o ile tylko potrafimy zdefiniować metodę przekształcenia obiektu danej klasy w obiekt innej klasy.
Wykorzystanie konstruktora do konwersji nie jest możliwe, gdy chcemy dokonać konwersji w drugą stronę, tj. przekształcić obiekt klasy do typu wbudowanego. W takich przypadkach definiujemy specjalną funkcję składową konwersji. Ogólna postać *funkcji konwersji* dla klasy `X` jest następująca:

```
X::operator T() { return w; }
```

gdzie `T` jest typem, do którego dokonujemy konwersji, zaś `w` – wyrażeniem, którego argumentami muszą być składowe klasy, ponieważ funkcja konwersji operuje na obiekcie, dla którego jest wołana.
Zauważmy, że w definicji funkcji konwersji nie podaje się ani typu zwracanego, ani argumentów.

Przykład 6.25.

```
#include <iostream.h>
class Test  {
public:
 Test(int i): x(i) {}
 operator int() { return x*x; };
private:
 int x;
};
int  main() {
     int num1 = 2, num2 = 3;
     Test t1(num1), t2(num2);
     int ii;
     ii = t1;
     cout << ii << endl;
     ii = 10 + t2;
     cout << ii << endl;
     return 0;
}
```

Wydruk z programu ma postać:

4
19

Dyskusja. W przykładzie mamy konwersje w obie strony: jednoparametrowy konstruktor `Test(int i)` przekształca argument `i` typu **int** w obiekt klasy `Test`, a funkcja konwersji `operator int() { return x*x; };` pozwala używać obiekty typu `Test` w taki sam sposób, jak obiekty typu **int**.

Przykład 6.26.

```
#include <iostream.h>
class Boolean  {
public:
 enum logika { false = 0, true = 1 };
 //konstruktory
 Boolean(): z(true) {}
 Boolean(int num): z(num != 0) { }
 Boolean(double d) { z = (d != 0); }
 Boolean (void* wsk): z(wsk != 0) { }
 //Konwersja
 operator int() const { return z; }
 //Negacja
 Boolean operator!() const { return !z; }
 private:
 char z;
};
Boolean pierwiastki(double a, double b, double c)
{
 int pr = b*b >= 4*a*c;
 cout << pr << endl;
 return pr;
}
int   main() {
      Boolean b1(Boolean::true);
      Boolean b2(5);
      int ii = !b1 || b2;
      cout << "ii = " << ii << endl;
      int* wsk = new int(10);
      Boolean b3(wsk);
      Boolean b4(3.5);
      double a = 1, b = 4, c = 3;
      if(pierwiastki(a,b,c))
      cout << "Dwa" << endl;
      return 0;
}
```

Wydruk z programu ma postać:

ii = 1
1
Dwa

Dyskusja. Klasa `Boolean` imituje predefiniowany typ **bool**, który dopiero ostatnio wprowadzono do standardu języka C++. W klasie zdefiniowano cztery konstruktory, które mogą tworzyć obiekty typu `Boolean`. Pierwszy z nich, domyślny konstruktor bezparametrowy, wychodzi poza opisaną wcześniej konwencję, tym niemniej bywa użyteczny, np. przy tworzeniu tablic wartości logicznych. W definicji konstruktora z parametrem typu **int**, `Boolean(int num): z(num != 0) { }` użyto częściej obecnie stosowanej notacji, niż starsza, w której napisalibyśmy:
`Boolean(int num) { z = num != 0; }`

Wartości logiczne *prawda* i *fałsz* zostały zdefiniowane jako typ wyliczeniowy `logika` z jawnie zainicjowanymi stałymi `true` i `false`. Zdefiniowano także przeciążony operator negacji logicznej. Funkcja konwersji do typu **int** jest funkcją stałą, podobnie jak funkcja `operator!()`. Obiekty klasy `Boolean` są wykorzystywane w instrukcji przypisania `int ii = !b1 || b2;`
Wykonanie tej instrukcji przebiega następująco:

a) Dla obiektu `b1` zostaje wywołana funkcja `operator!();` powrót z tej funkcji wymaga utworzenia obiektu tymczasowego. Obiekt taki jest tworzony przez wywołanie konstruktora `Boolean(int)`.
b) Po wykonaniu negacji zostaje dwukrotnie wywołana funkcja konwersji `operator int()` dla obiektów `!b1` oraz `b2`, co pozwala obliczyć wartość alternatywy logicznej.
c) Wartość alternatywy logicznej zostaje przypisana do `ii`.

Obiekt `b3(wsk)` jest tworzony przez wywołanie konstruktora `Boolean(void* wsk) { z = wsk != 0; }`. Wykorzystuje się tutaj własność typu **void***, do którego może być automatycznie (niejawnie) przekształcony wskaźnik dowolnego typu.
W instrukcji **if** wywoływana jest funkcja `pierwiastki()`. Ponieważ funkcja jest typu `Boolean`, przy powrocie tworzony jest obiekt tymczasowy wywołaniem konstruktora `Boolean(int)`, a następnie zostaje wywołana dla tego obiektu funkcja konwersji, jako że wyrażenie w instrukcji **if** musi dawać wartość liczbową.

7.5.5. Klasa String

Konwersje typów okazują się szczególnie przydatne w operacjach na łańcuchach znaków i dlatego poświęcimy temu tematowi osobny podrozdział.
Przypomnijmy, że podstawowe operacje dla typu **char*** (obliczanie długości łańcucha, kopiowanie, konkatenacja łańcuchów, etc.) są zadeklarowane w pliku nagłówkowym `string.h`, zaś kilka operacji konwersji łańcuchów na liczby zadeklarowano w `stdlib.h`. Operacje te są zapożyczone z ANSI C. Zadeklarowane w pliku `string.h` bardzo użyteczne funkcje operują na zakończonych znakiem `'\0'` łańcuchach znaków języka C. Korzystanie z nich bywa jednak dość uciążliwe, szczególnie w odniesieniu do zarządzania pamięcią. Weźmy dla przykładu funkcję, która bierze dwa łańcuchy jako argumenty i scala je w jeden, pozostawiając spację pomiędzy łańcuchami wejściowymi:

```
char* SPkonkat(const char* wyraz1, const char* wyraz2)
{
 unsigned int rozmiar=strlen(wyraz1)+strlen(wyraz2)+1;
 char* wynik = new char[rozmiar];
 return
 strcat(strcat(strcpy(wynik,wyraz1), " "),wyraz2);
}
```

Pierwsza instrukcja w bloku funkcji oblicza długość wynikowego łańcucha, uwzględniając rozdzielającą wyrazy spację i terminalny znak zerowy (`'\0'`), a druga alokuje niezbędną pamięć. Po przydzieleniu pamięci trzecia instrukcja wykorzystuje dwie funkcje biblioteczne (ze `string.h`): najpierw kopiuje łańcuch `wyraz1` do zmiennej `wynik`, dołącza rozdzielającą spację, i na koniec dołącza drugi łańcuch `wyraz2`. Każda z funkcji bibliotecznych zwraca wskaźnik do swojego pierwszego argumentu (tj. łańcucha docelowego), dzięki czemu mogliśmy ustawić w sekwencję wywołania kolejnych funkcji. Funkcja wołająca jest “odpowiedzialna” za usunięcie wynikowego łańcucha:

```
char* wsk = SPkonkat("Jan", "Kowalski");
// ...
delete wsk;
```

Jak widać z powyższego przykładu, celowym byłoby zdefiniować klasę, która stanowiłaby swego rodzaju “opakowanie” dla istniejących funkcji języka C, ułatwiając użytkownikowi operowanie na łańcuchach znaków za pomocą kilku prostych operatorów.

Termin “opakowanie” odpowiada prawie dokładnie temu, co określiliśmy wcześniej jako hermetyzację albo ukrywanie informacji. Tutaj naszą intencją jest taka abstrakcja danych, aby

szczegóły reprezentacji łańcuchów języka C, tj. typu **char***, zostały ukryte przed użytkownikiem. Zamiast nich użytkownik powinien mieć do dyspozycji dobrze zaprojektowany interfejs (część publiczną) do klasy, której obiekty zachowywałyby się podobnie, jak łańcuchy języka C.

Podane niżej deklaracje, definicje i komentarze można traktować jako prostą implementację takiej klasy.

Przykład 6.27.

```
class String  {
public:
//Publiczny interfejs uzytkownika:
//Redefinicja "+" dla konkatenacji - trzy przypadki:
  String operator+(const String&) const;
  friend String operator+(const char*, const String&);
  friend String operator+(const String&, const char*);
  int length() const; // Length of string in chars
  String();
  String(const char*);
  String(const String&);
  ~String();
  String& operator=(const String&);
  operator const char*() const;
  friend ostream& operator<<(ostream&, const String&);
  char& operator [] (int);
private:
  char* dane;
};
```

Dyskusja. Pierwsze spostrzeżenie: zmienna `char* dane`, reprezentująca łańcuch znaków języka C, jest ukryta w części prywatnej klasy `String` i jest dostępna jedynie dla jej funkcji składowych i operatorów.

Klasa ma trzy konstruktory.

a) **Konstruktor domyślny String::String();**

```
String::String()
{ dane = new char[1]; dane[0] = '\0'; }
```

który tworzy łańcuch pusty. Konstruktor ten będzie wołany np. przy tworzeniu wektora obiektów klasy `String: String wektor[10];`

b) **Konstruktor String::String(const char*);**

```
String::String(const char* st)  {
int rozmiar = ::strlen(st) + 1;
dane = new char[rozmiar];
::strcpy(dane, s);              }
```

który dokonuje wspomnianej uprzednio konwersji łańcucha `st` w obiekt klasy `String`. W tej i w następnych definicjach będziemy używać unarnego operatora zasięgu "::" przy odwołaniach do zmiennych i funkcji globalnych (z pliku `string.h`), aby uniknąć konfliktu nazw. Rozmiar tworzonego dynamicznie podobiektu `dane` jest o 1 większy od długości łańcucha `st`, aby zmieścić terminalny znak `'\0'`.

c) Konstruktor kopiujący String::String(const String&);

```
String::String(const String& st);   {
dane = new char[st.length() + 1];
// kopiuj stare dane na nowe
::strcpy(dane, st.dane);            }
```

który jest wywoływany przy przesyłaniu parametrów przez wartość i niekiedy przy powrocie z funkcji.

Destruktor:

```
String::~String() { delete [] dane; }
```

Jego zadaniem jest zwolnienie wszystkich zasobów, które pozyskał obiekt dzięki wykonaniu konstruktorów lub funkcji składowych.

Operatorowa funkcja przypisania:

```
String& String::operator=(const String& st)
{
  if(dane != st.dane)             {
    delete [] dane;
    int rozmiar = st.length() + 1;
    dane = new char[rozmiar];
    ::strcpy(dane, st.dane);      }
  return *this; // referencja do obiektu
}
```

Funkcja zwraca referencję do klasy `String`, a więc nie jest tworzony żaden obiekt tymczasowy. W bloku funkcji najpierw sprawdza się, czy nie jest to próba przypisania obiektu do samego siebie; jeżeli nie, to usuwa się stare dane. Kolejna instrukcja tworzy obiekt w pamięci swobodnej, a następna kopiuje zawartość pola st.dane argumentu funkcji do nowego obiektu dane. Wynik operacji, `*this`, jest referencją do klasy `String`.

Funkcja konwersji z typu String do typu char*:

```
String::operator const char*() const { return dane; }
```

pozwala używać obiekty klasy `String` w tych samych kontekstach, co zwykłe łańcuchy znaków. Zwróćmy uwagę na dwukrotne wystąpienie modyfikatora **const**. Pierwszy z lewej ustala, że zawartość obszaru pamięci wskazywanego przez wartość do której następuje konwersja (typu `char*`) nie może być zmieniona przez żadną operację zewnętrzną w stosunku do klasy. Drugi **const** oznacza, że zdefiniowana wyżej funkcja składowa `operator const char*() const` nie zmienia stanu obiektu klasy `String`, na którym operuje.

Definicje przeciążonych operatorów '<<' i '[]' oraz funkcji przekazującej długość łańcucha są typowe i nie wymagają komentarzy:

```
ostream& operator<<(ostream& os, const String& cs)
{  return os << cs.dane;  }

char& String::operator [] (int indeks)
{  return dane[indeks];  }

int String::length() const
{ return ::strlen(dane); }
```

Natomiast szerszego komentarza wymagają operatory konkatenacji.

a) **Operator konkatenacji dla klasy**

```
String String::operator+(const String& st) const
{
 char* buf = new char[st.length() + length() + 1];
 ::strcpy(buf, dane);
 ::strcat(buf, st.dane);
 String retval(buf); //Obiekt tymczasowy
 delete [] buf;
 return retval;
}
```

Pierwsza instrukcja alokuje pamięć na wynikowy łańcuch; druga kopiuje dane do tego łańcucha, a trzecia scala te dane z danymi przekazanymi przez argument `st`. Następnie ze zmiennej `buf` jest tworzony nowy, lokalny obiekt `retval` klasy `String`, usuwany już niepotrzebny `buf`, a przy powrocie funkcja przekazuje kopię `retval`, tworzoną przez konstruktor kopiujący. Korzysta się z tej definicji w następujący sposób: jeżeli `s1` i `s2` są obiektami klasy `String`, to możemy je "dodać" do siebie, pisząc: `s1 + s2;` lub `s1.operator+(s2);`

b) **Zaprzyjaźnione operatory konkatenacji**

W klasie `String` zadeklarowano dwa operatory konkatenacji, aby było możliwe "dodawanie" obiektów klasy `String` do zwykłych łańcuchów znaków, z obiektami klasy `String` zarówno po prawej, jak i po lewej stronie operatora "+".

```
String operator+(const char* sc, const String& st)
{
 String retval;
 retval.dane = new char[::strlen(sc) + st.length()];
 ::strcpy(retval.dane, sc);
 ::strcat(retval.dane, st.dane);
 return retval;
}

String operator+(const String& st, const char* sc)
{
 String retval;
 retval.dane = new char[::strlen(sc) + st.length()];
 ::strcpy(retval.dane, st.dane);
 ::strcat(retval.dane, sc);
 return retval;
}
```

Jak już wspomniano, musimy mieć dwie funkcje operatorowe dla zapewnienia symetrii. Dzięki tym definicjom mogą zostać wykonane instrukcje:

```
String s1;
"abcd" + s1;
s1 + "abcd";
```

Symetrię można też zapewnić dla obiektów klasy, definiując dodatkowy konstruktor dwuargumentowy:

```
String::String(const String& st1, const String& st2):
dane(strcat(strcpy(new char[st1.strlen() +
st2.strlen()+1],st1.dane),st2.dane)) {  }
```

i redefiniując funkcję składową `operator+()`:

```
String operator+(const String& st1, const String& st2)
{  return String( st1, st2);  }
```

Konstruktor bierze dwa obiekty klasy `String`, alokuje pamięć dla połączonego łańcucha, kopiuje pierwszy argument do przydzielonego obszaru pamięci i na koniec dokonuje konkatenacji drugiego argumentu z poprzednim wynikiem. Taki specjalizowany konstruktor bywa nazywany *konstruktorem operatorowym*. Definiuje się go jedynie w celu implementacji danego operatora. Symetryczny operator "+" po prostu wywołuje ten konstruktor dla wykonania konkatenacji swoich argumentów.

Podsumowanie. Zaprezentowana wyżej klasa `String` nie może pretendować do przyjęcia jej jako standardowej klasy bibliotecznej dla łańcuchów języka C++, ponieważ przyjęliśmy zbyt wiele założeń upraszczających, np. nie sprawdzaliśmy powodzenia alokacji dynamicznej, etc. Pominęliśmy również wiele możliwych do wprowadzenia użytecznych funkcji, np. operatory porównania łańcuchów, kopiowania fragmentów łańcuchów, wyszukiwania znaków i ciągów znaków w łańcuchach, etc. Tym niemniej struktura tej klasy powinna ułatwić użytkownikowi zrozumienie podobnych konstrukcji bibliotecznych.
W charakterze wstępnego treningu można sprawdzić działanie podanego niżej programu, lokując przedtem deklarację klasy `String` wraz z definicjami funkcji składowych i zaprzyjaźnionych w pliku nagłówkowym "`wpstring.h`" (lub w dwóch plikach: w jednym, "`wpstring.h`", deklarację klasy, a w drugim, np. "`wpstring.cc`" albo "`wpstring.cpp`", definicje funkcji).

```
#include <iostream.h>
#include "wpstring.h"
int   main() {
      String s1("ABC");
      String s2 = s1;
      String s3;// pusty
      s3 = s2;
      String s4;//pusty
      s4 = s2 + s3;
//    lub s4 = s2.operator+(s3);
      "DEF" + s1;
      s1 + "ghi";
      cout << s1[1] << endl;
      cout << s1 + "ghi" << endl;
      return 0;
}
```

Wydruk ma postać:

B
ABCghi

7.6. Tablice obiektów

Kilkakrotnie już zwracaliśmy uwagę na fakt, że obiekty klas są zmiennymi typów definiowanych przez użytkownika i mają analogiczne własności, jak zmienne typów wbudowanych. Tak więc nic nie stoi na przeszkodzie, aby umieszczać obiekty w tablicy. Deklaracja tablicy obiektów jest w pełni analogiczna do deklaracji tablicy zmiennych innych typów. Także sposób dostępu do elementów takiej tablicy niczym się nie różni od poznanych już zasad. Jedyną istotną cechą wyróżniającą tablicę obiektów od innych tablic jest sposób ich inicjowania. Tablicę obiektów danej klasy inicjuje się przez jawne, bądź niejawne wywoływanie konstruktora klasy tyle razy, ile elementów zawiera tablica. Jeżeli w klasie zdefiniowano konstruktory, to można je użyć w wywołaniu jawnym w taki sam sposób, jak dla indywidualnych obiektów; parametry aktualne wywołań konstruktora będą wartościami inicjalnymi dla zmiennych składowych każdego obiektu tablicy. Dla konstruktorów domyślnych, tj. z pustym wykazem argumentów, wartości inicjalne zmiennych składowych będą zależne od tego, czy w bloku konstruktora podano wartości domyślne: jeżeli tak, to wartości inicjalne będą równe wartościom domyślnym; jeżeli nie, to będą przypadkowe.
Innym wygodnym sposobem jest zainicjowanie każdego elementu tablicy wcześniej utworzonym obiektem, lub przypisanie każdemu elementowi tablicy wcześniej utworzonego obiektu. W pierwszym przypadku wywoływany jest (niejawnie) konstruktor kopiujący generowany przez kompilator lub (jawnie) konstruktor kopiujący, zdefiniowany w klasie. W drugim przypadku będzie to generowany lub zdefiniowany operator przypisania. Ponadto jednowymiarowe tablice obiektów, których konstruktor zawiera tylko jeden parametr, można inicjować dokładnie tak samo, jak tablice, których elementami są zmienne typów podstawowych, podając wartości inicjalne w nawiasach klamrowych.
Dla wprowadzenia w zagadnienie posłużymy się obiektami wielokrotnie już eksploatowanej klasy `Punkt`.

Przykład 6.28.

```
#include <iostream.h>
class Punkt  {
public:
 Punkt() {}
 Punkt(int a): x(a) {}
 int fx() { return x; }
private:
 int x;
};
int  main() {
     Punkt p1[] = { Punkt(10), Punkt(20), Punkt(30) };
     for (int i = 0; i < 3; i++)
     cout << "p1[" << i << "].x = "
          <<  p1[i].fx() << '\t';
     cout << '\n';
     Punkt p2[] = { 15, 25, 35 };
     Punkt p3[] = { Punkt(), Punkt(), Punkt() };
     for (i = 0; i < 3; i++)
     cout << "p3[" << i << "].x = "
          <<  p3[i].fx() << '\t';
     cout << '\n';
     Punkt p4[] = { Punkt(), Punkt(40), Punkt() };
     for (i = 0; i < 3; i++)
     cout << "p4[" << i << "].x = "
          <<  p4[i].fx() << '\t';
     cout << '\n';
     return 0;
}
```

Dyskusja. Klasa `Punkt` ma dwa konstruktory: bezparametrowy konstruktor domyślny `Punkt()` oraz konstruktor z jednym parametrem `Punkt(int a)`. Pierwsza instrukcja deklaracji tworzy tablicę złożoną z trzech obiektów, wywołując trzykrotnie konstruktor z parametrem. W instrukcji:

```
Punkt p2[] = { 15, 25, 35 };
```

mamy uproszczoną postać zapisu wywołań konstruktora `Punkt(15)`, `Punkt(25)` i `Punkt(35)`. Instrukcja, tworząca tablicę `p3[]` wywołuje trzykrotnie konstruktor domyślny, zaś instrukcja definiująca tablicę `p4[]` wywołuje dwukrotnie konstruktor domyślny i jeden raz konstruktor z parametrem. Ponieważ wszystkie tablice były inicjowane żądaną liczbą obiektów, można było opuścić w deklaracji wymiary tablic.
Wydruk z przedstawionego programu może mieć postać:

p1[0].x = 10 p1[1].x = 20 p1[2].x = 30
p3[0].x = 1160 p3[1].x = -14 p3[2].x = 8607
p4[0].x = 12950 p4[1].x = 40 p4[2].x = 0

(Użyto tutaj słowa “może”, ponieważ wartości inicjalne, uzyskiwane przez wywołanie konstruktora domyślnego, będą przypadkowe).

Przykład 6.29.

```
#include <iostream.h>
class Punkt  {
public:
  Punkt(): x(0) {}
  Punkt( int a ): x(a) {}
  Punkt(const Punkt& p) { x = p.x; }
  Punkt& operator=(const Punkt& p)
  { this->x = p.x; return *this; }
  int fx() { return x; }
private:
  int x;
};
int  main() {
      Punkt p0(7);
      Punkt p1[] = { p0, p0, p0 };
      Punkt p2[3];
      for (int i = 0; i < 3; i++)
      p2[i] = p1[i];
      return 0;
}
```

Dyskusja. W programie wykorzystano trzy sposoby inicjowania tablicy obiektów. Instrukcja `Punkt p1[]={p0,p0,p0};` inicjuje zmienną składową `x` każdego obiektu trzyelementowej tablicy `p1[]` wartością 7, skopiowaną z obiektu `p0`. Kopiowanie każdego obiektu tablicy odbywa się przez wywołanie konstruktora kopiującego `Punkt(const Punkt& p)`. Zauważmy, że parametr aktualny (argument) dla konstruktora kopiującego musi być stały i przekazywany przez referencję (gdyby przyjąć przekazywanie przez wartość, to mielibyśmy wywołanie konstruktora kopiującego, który właśnie definiujemy).
Z kolei każdy obiekt tablicy `p2[]` jest inicjowany wartością zero przez konstruktor domyślny `Punkt(){x=0;}`. Następnie w pętli **for** kolejnym obiektom tablicy `p2[]` jest przypisywany obiekt `p1[i]` uprzednio zainicjowanej tablicy `p1[]`. W tym przypadku za każdym razem jest wywoływany przeciążony operator przypisania

```
Punkt& Punkt::operator=(const Punkt&)
```

Jak dla każdego operatora składowego klasy, pierwszym argumentem funkcji `operator=()` jest wskaźnik do obiektu, dla którego jest wywoływana (zamiast `x = p.x` można napisać `this->x = p.x`). Drugi argument jest przekazywany przez referencję i stały, co gwarantuje jego niezmienność. Funkcja `operator=()` zwraca referencję do obiektu klasy `Punkt`, wobec czego w instrukcji `return` występuje jawnie nazwa tego obiektu, `*this`.
Konstruktor kopiujący i przeciążony operator przypisania zostały użyte w tym przykładzie głównie dla celów dydaktycznych. Gdyby zabrakło ich definicji, odpowiednie operacje zostałyby wykonane przez operatory generowane. Faktyczna potrzeba takich definicji pojawia się dopiero wtedy, gdy obiekt zawiera wskaźniki do innych obiektów. Wówczas kopiowanie składowych, wykonywane przez operatory generowane, przestaje być w ogólności wystarczające, ponieważ kopiuje się wskaźniki, a nie obiekty, do których wskaźniki się odwołują.

Przykład 6.30.

```
#include <iostream.h>
class Punkt  {
public:
 Punkt(): x(0) {}
 Punkt( int a ): x(a){}
 void ustaw(int b) { x = b; }
 int fx() { return x; }
private:
 int x;
};

int   main() {
      int i,j;
      Punkt p1[2][3];
      for (i = 0; i < 2; i++)
      {
       for (j = 0; j < 3; j++)
       p1[i][j].ustaw(10 + j);
      }
      for(i = 0; i < 2; i++)
      {
       for (j = 0; j < 3; j++)
      {
       cout << "p1[" << i << "][" << j << "].x = ";
       cout << p1[i][j].fx() << '\n';
      }
      }
      cout << endl;
      return 0;
}
```

Dyskusja. W programie zadeklarowano tablicę `p[2][3]` o dwóch wierszach i trzech kolumnach. Instrukcja deklaracji: `Punkt p1[2][3];` wywołuje sześciokrotnie konstruktor domyślny `Punkt() { x = 0; }`, który ustawia zmienną składową `x` na wartość zero. Tworzona w ten sposób tablica jest inicjowana wierszami. W pierwszej pętli **for** dla każdego obiektu tablicy jest wywoływana funkcja składowa `Punkt::ustaw()`, która ustawia zmienne `Punkt::x` w kolejnych obiektach na wartości 10+j. Wydruk z programu ma postać:

p[0][0].x = 10
p[0][1].x = 11
p[0][2].x = 12
p[1][0].x = 10
p[1][1].x = 11
p[1][2].x = 12

Zauważmy, że i w tym przypadku moglibyśmy każdemu obiektowi tablicy `p1[2][3]` przypisać wcześniej utworzony obiekt. Np. tworząc obiekt `p0(7)` instrukcją deklaracji `Punkt p0(7);` tj. wywołując konstruktor z parametrem, możemy zamiast instrukcji

```
p1[i][j].ustaw(10 + j);
```

użyć w pętli **for** instrukcję

```
p1[i][j] = p0;
```

W tym przypadku dla każdego `i,j` będzie wołany generowany przez kompilator operator przypisania dla obiektów klasy `Punkt`.
Innym możliwym wariantem omawianej instrukcji przypisania mogłaby być instrukcja:

```
p1[i][j] = p0(i);
```

W tym przypadku dla każdego obiektu tablicy `p1[][]` będą wykonywane kolejno dwie operacje:

- Wywołanie konstruktora `Punkt::Punkt( int i ) { x = i; }`
- Wywołanie generowanego przez kompilator domyślnego operatora przypisania `Punkt& Punkt::operator=(const Punkt&).`

Operator przypisania, podobnie jak zdefiniowany w poprzednim przykładzie, przypisuje składowej `x` obiektu `p[i][j]` wartość składowej `x` obiektu `p0`.

Przykład 6.31.

```
// Alokacja dynamiczna - 1
#include <iostream.h>
class Punkt  {
public:
   Punkt() {};
   void ustaw(int c, int d)
   { x = c; y = d; }
   int fx()  { return x; }
   int fy()  { return y; }
private:
   int x,y;
};
int  main() {
      Punkt* wsk;
      wsk = new Punkt [3];
      if (!wsk)
      {
       cerr << "Nieudana alokacja\n";
       return 1;
      }
      for (int i = 0; i < 3; i++)
      wsk[i].ustaw(i,i);
```

```
      delete [] wsk;
      return 0;
}
```

Dyskusja. Program tworzy tablicę trzech obiektów typu `Punkt` w pamięci swobodnej. Ponieważ dla tablicy dynamicznej nie można podać wartości inicjujących w jej deklaracji, w klasie `Punkt` zdefiniowano tylko konstruktor domyślny. Inicjowanie tablicy odbywa się w pętli **for**, za pomocą funkcji składowej `Punkt::ustaw()`. Ponieważ w klasie `Punkt` nie zdefiniowano destruktora, zastosowano składnię operatora **delete** bez symbolu "[]".

Przykład 6.32.

```
// Alokacja dynamiczna - 2
#include <iostream.h>
class Punkt  {
public:
   Punkt(): x(0),y(0) {};
   ~Punkt() { cout << "Destrukcja...\n"; }
   int fx() { return x; }
   int fy() { return y; }
private:
   int x,y;
};

int  main() {
      Punkt* wsk;
      wsk = new Punkt [3];
      if (!wsk)
      {
       cerr << "Nieudana alokacja\n";
       return 1;
      }
      for (int i = 0; i < 3; i++)
      cout << wsk[i].fx() << '\t'
           << wsk[i].fy() << endl;
      delete [] wsk;
      return 0;
}
```

Dyskusja. W tym przykładzie klasa `Punkt` zawiera definicję destruktora; teraz odzyskiwanie pamięci przydzielonej dla `wsk` jest następuje przez wywołanie destruktora dla każdego obiektu składowego tablicy. Ilustruje to wydruk z programu:

```
0    0
0    0
0    0
Destrukcja...
Destrukcja...
Destrukcja...
```

7.7. Wskaźniki do elementów klasy

Poznane dotąd konwencje notacyjne nie dawały nam możliwości wyrażenia w sposób jawny wskaźnika do elementu składowego klasy. Dla zmiennej składowej klasy możliwość taką stwarza deklaracja o postaci:

```
klasa::*wsk = &klasa::zmienna;
```

gdzie: `klasa` jest nazwą klasy, w której jest zadeklarowana zmienna o nazwie `zmienna`, zaś `wsk` jest wskaźnikiem do tej zmiennej. Wskaźnik jest inicjowany wyrażeniem z prawej strony operatora "=".

Dla funkcji składowej klasy stosuje się następującą postać deklaracji wskaźnika:

```
typ (klasa::*wskf)(arg) = &klasa::funkcja;
```

gdzie: `typ` jest typem wartości obliczonej przez funkcję `funkcja`, `wskf` jest wskaźnikiem do tej funkcji, zaś `arg` jest wykazem argumentów (sygnaturą), z opcjonalnymi identyfikatorami. Wskaźnik jest inicjowany wyrażeniem z prawej strony operatora "=".

Zauważmy, że w powyższych deklaracjach używa się binarnego operatora "::*", który informuje kompilator, iż wskaźniki `wsk` i `wskf` mają zasięg klasy. Wskaźniki są inicjowane adresami wskazywanych składowych klasy. Przypomnijmy, że dla dostępu bezpośredniego do elementów klasy używaliśmy operatora "::".

Inicjowanie wskaźników adresami elementów składowych bezpośrednio w ich deklaracji nie jest, rzecz jasna, obowiązkowe. W instrukcji deklaracji wskaźnik można zainicjować zerem (`0` lub `NULL`), lub też można mu przypisać odpowiedni adres w instrukcji przypisania. Tak więc możemy np. zadeklarować:

```
klasa::*wsk1 = 0; // lub NULL
klasa::*wsk2 = &klasa::zmienna;
wsk1 = wsk2;
```

Wskaźniki do zmiennych składowych klasy okazały się użyteczną metodą wyrażania "topografii" klasy, tj. względnego położenia elementów w klasie w sposób niezależny od implementacji. Podany niżej przykład ilustruje deklarację wskaźnika oraz sposób dostępu do zmiennej składowej `x` klasy `Punkt`.

Przykład 6.33.

```
#include <iostream.h>
class Punkt {
public:
  int x;
};
int Punkt::*wskx = &Punkt::x;
int  main() {
     Punkt punkt1;
     punkt1.*wskx = 10;
     cout << punkt1.x << endl;
     return 0;
}
```

Dyskusja. Zauważmy, że odwołanie do zmiennej składowej `Punkt::x` jest możliwe tylko poprzez wystąpienie tej klasy, tj. obiekt `punkt1`. Postać odwołania pośredniego (`.*`) różni się od bezpośredniego (`.`) dodaniem symbolu "*", zaś typem wskaźnika `wskx` jest `Punkt::*`. Zwróćmy też uwagę na fakt, że zmienna składowa `x` jest publiczna w klasie. Gdyby zmienna składowa była prywatna, to próba dostępu do adresu tej składowej byłaby zasygnalizowana przez kompilator jako błąd. W takim przypadku należałoby rozszerzyć deklarację klasy o odpowiednią funkcję zaprzyjaźnioną, dodać w klasie `Punkt` definicję funkcji, przekazującej wartość `x`:

```
friend int f(Punkt& p)
```

```
{
 int Punkt::*wskx = &Punkt::x;
 p.*wskx = 10;
 return p.x;
}
```

i wywołać ją dla obiektu `punkt1`, np w instrukcji:

```
cout << f(punkt1);
```

Przy deklarowaniu wskaźnika do statycznej zmiennej (a także funkcji) składowej klasy nie można mu nadać typu `klasa::*`, jak dla składowej niestatycznej, ponieważ kompilator nie przydziela pamięci dla składowej statycznej w żadnym obiekcie danej klasy. Tak więc w tym przypadku mamy zwykły wskaźnik.

Przykład 6.34.

```
#include <iostream.h>
class Punkt {
public:
  static int x;
};
int Punkt::x = 10;
int* wskx = &Punkt::x;
int  main() {
     cout << *wskx << endl;
     return 0;
}
```

Dyskusja. Zgodnie z obowiązującymi zasadami składowa statyczna `x` została zainicjowana poza blokiem klasy `Punkt` wartością 10. W deklaracji wskaźnika do tej składowej, `wskx`, nie wystąpił operator "::*", lecz tylko "*". Ponieważ `x` jest samodzielnym obiektem typu **int**, zatem `wskx` jest typu **int***, a odwołanie do wartości zmiennej `x` ma postać `*wskx`.

Znacznie więcej korzyści daje zadeklarowanie wskaźnika do funkcji składowej klasy. Jest to jeden ze sposobów wprowadzenia zmiany zachowania się funkcji w fazie wykonania. Wskaźnik do niestatycznej funkcji składowej możemy wprowadzić w deklaracji klasy i używać go do wywołań funkcji po uprzednim właściwym zainicjowaniu. W deklaracji wskaźnika należy podać zarówno typ klasy zawierającej wskazywaną funkcję składową, jak i sygnaturę tej funkcji. Podanie tych informacji jest wymagane przez kompilator, jako że C++ jest językiem o silnej typizacji i sprawdza nawet wskaźniki do funkcji.
Mając zadeklarowany wskaźnik możemy mu przypisywać adresy innych funkcji składowych pod warunkiem, że funkcje te mają tę samą liczbę i typy parametrów oraz typ obliczanej wartości. Podany niżej przykład ilustruje używaną w tym przypadku notację.

Przykład 6.35.

```
#include <iostream.h>
class Firma {
public:
  char* nazwa;
  Firma(char* z): nazwa(z) {}; // Konstruktor
  ~Firma() {}; // Destruktor
  void podaj(int x)
  {
   cout << nazwa << " " << x << '\n';
  }
};

typedef void(Firma::*WSKFI) (int);

int   main() {
      Firma frm1("firma1");
      Firma* wsk = &frm1;
      WSKFI wskf = &Firma::podaj;
      frm1.podaj(1);
       (frm1.*wskf)(2);
       (wsk->*wskf)(3);
      return 0;
}
```

Wydruk z programu ma postać:

firma1 1
firma1 2
firma1 3

Dyskusja. Deklaracja `typedef void(Firma::*WSKFI) (int);`
wprowadza nazwę `WSKFI` dla wskaźnika do funkcji typu **void** o zasięgu klasy `Firma` i jednym argumencie typu **int**. Identyfikator `WSKFI` używa się następnie dla zadeklarowania wskaźnika `wskf`, zainicjowanego adresem funkcji składowej `podaj()` klasy `Firma`. W programie pokazano trzy sposoby drukowania zawartości pola `Firma::nazwa` obiektu `frm1`:
instrukcja `frm1.podaj(1);`

korzysta z operatora "`.`" dostępu bezpośredniego;
instrukcja `(frm1.*wskf)(2);`
korzysta z operatora "`.*`" dostępu pośredniego;
instrukcja `(wsk->*wskf)(3);`
korzysta z operatora "`->*`", który jest operatorem dostępu pośredniego dla wskaźnika `wsk` do obiektu `frm1`.

Operatory `.*` i `->*` wiążą wskaźniki z konkretnym obiektem, dając w wyniku funkcję, która może być użyta w wywołaniu. Binarny operator `.*` wiąże swój prawy operand, który musi być typu "wskaźnik do składowej klasy T" z lewym operandem, który musi być typu "klasy T". Wynikiem jest funkcja typu określonego przez drugi operand. Analogicznie `->*`. Priorytet `()` jest wyższy niż `.*` i `->*`, tak że nawiasy są konieczne. Gdyby pominąć nawiasy, to np. wyrażenie:

```
frm1.*wskf(2);
```

byłoby potraktowane jako

```
frm1.*(wskf(2))
```

czyli jako wartość składowej obiektu `frm1`, na którą wskazuje wartość zwracana przez zwykłą funkcję `wskf()`.

☞*Uwaga. Wartościowanie wyrażenia z operatorem '.*' lub '->*' daje l-wartość, jeżeli prawy operand jest l-wartością.*

Kolejny przykład można potraktować jako fragment oprogramowania systemu nadzorowania obiektu, w którym sygnał przychodzący do miejsca oznaczonego jako `przycisk` wyzwala odpowiednią reakcję systemu. Sygnał `0` oznacza stan normalny, sygnał `1` – ewentualne zagrożenie (alert), zaś sygnał 2 – alarm. Stany te mogą być wyświetlane na ekranie (jak w programie), lub powodować inną reakcję (np. odpowiednie sygnały dźwiękowe).

Przykład 6.36.

```
#include <iostream.h>
class Ochrona {
public:
 char* nazwa;
 Ochrona(char* z): nazwa(z) {}; // Konstruktor
 ~Ochrona() {}; // Destruktor
 void (Ochrona::*przycisk) (int j); // Wskaznik
 void kontrola(int);
 void norma(int x)
 { cout << "STAN NORMALNY" << endl; }
 void alert(int y) { cout << "POGOTOWIE" << endl; }
 void alarm(int z) { cout << "ALARM!!!" << endl; }
};
 void Ochrona::kontrola(int w)
 {
  switch (w) {
     case 0: przycisk = &Ochrona::norma; norma(w);
             break;
     case 1: przycisk = &Ochrona::alert; alert(w);
             break;
     case 2: przycisk = &Ochrona::alarm; alarm(w);
             break;
             }
 }

int   main() {
      Ochrona ob("System1");
      typedef void (Ochrona::*WSKFI) (int);
      WSKFI wskf = &Ochrona::kontrola;
      int ii;
      cin >> ii;
      (ob.*wskf)(ii);
      Ochrona* wsk = &ob;
      (wsk->*wskf)(ii);
      (ob.*(ob.przycisk))(ii);
      return 0;
}
```

Przykładowy wydruk z programu dla `ii==1`, ma postać:

ALARM!!!
ALARM!!!
ALARM!!!

Dyskusja. Podobnie jak w poprzednim przykładzie użyto deklaracji **typedef** dla łatwiejszego odwoływania się do wskaźników funkcji. W klasie `Ochrona` zadeklarowano "funkcję składową" `przycisk`, która posłużyła – w ostatniej przed `return` instrukcji programu – do sprawdzenia, jaki sygnał został przekazany do systemu nadzorowania. Termin "funkcja składowa" ujęliśmy w znaki cudzysłowu, ponieważ `przycisk` nie jest funkcją, lecz wskaźnikiem funkcji, ustawianym w bloku funkcji `kontrola` na adres funkcji `norma()`, `alert()`, lub `alarm()`. W instrukcji **switch** jest podejmowana decyzja o tym, którą z funkcji składowych należy wywołać dla zadanego parametru aktualnego `ii`. Wykonanie programu przebiega następująco:
Pierwsza instrukcja w bloku `main()` woła konstruktor `Ochrona(char*)` i tworzy obiekt `ob`. W instrukcji `WSKFI wskf = &Ochrona::kontrola;` wskaźnik `wskf` jest inicjowany na adres funkcji składowej `kontrola()`. Po wczytaniu wartości `ii` (np. 2), wykonywana jest instrukcja `(ob.*wskf)(ii);`, tzn. przez wskaźnik `wskf` zostaje wywołana funkcja składowa `kontrola(2)`. W funkcji `kontrola()` wskaźnikowi `przycisk` zostaje przypisany adres funkcji składowej `alarm()`, po czym zostaje wywołana funkcja `alarm()` z parametrem aktualnym 2. Po wykonaniu funkcji `alarm()`, a następnie instrukcji `break;` program opuszcza blok funkcji `kontrola()`. W programie zadeklarowano również wskaźnik do obiektu klasy `Ochrona` i zainicjowano go adresem obiektu `ob`. Posłużyło to do wywołania funkcji `kontrola()` z instrukcji `(wsk->*wskf)(ii)`, w której wskaźnik `wsk` odwołuje się do wskaźnika `wskf`, a ten wywołuje funkcję `kontrola(2)` i dalej proces biegnie jak poprzednio. Ostatnia instrukcja, `(ob.*(ob.przycisk))(ii)` odwołuje się bezpośrednio do funkcji `alarm()`, ponieważ wskaźnik `przycisk` został już wcześniej ustawiony na adres tej funkcji. Program kończy wykonanie wywołaniem destruktora `~Ochrona()`.

7.8. Klasy w programach wieloplikowych

Przyjętym standardem dla programów wieloplikowych jest umieszczanie deklaracji klas w pliku (plikach) nagłówkowym. Definicje funkcji składowych oraz definicje inicjujące zmienne statyczne klas są umieszczane w pliku (plikach) źródłowym. Możliwa jest wtedy oddzielna kompilacja plików źródłowych, która oszczędza czas, ponieważ w przypadku zmian w programie powtórna kompilacja jest wymagana tylko dla plików, które zostały zmienione. Ponadto wiele implementacji zawiera program o nazwie *make*, który zarządza całą kompilacją i konsolidacją dla projektu programistycznego; program *make* rekompiluje tylko te pliki, które zostały zmienione od czasu ostatniej kompilacji. Zauważmy też, że rozdzielna kompilacja zachęca programistów do tworzenia ogólnie użytecznych plików tymczasowych (pliki z rozszerzeniem nazwy `.o` pod systemem Unix, lub `.obj` pod systemem MS-DOS) i bibliotek, które mogą wielokrotnie wykorzystywać w swoich własnych programach i dzielić je z innymi użytkownikami.
Podany niżej przykład ilustruje wykorzystanie w programie wieloplikowym tzw. *bezpiecznych tablic*. Przymiotnika "bezpieczny" używamy tutaj nie bez powodu. Dotychczas pomijaliśmy milczeniem fakt, że język C++ nie ma wbudowanego mechanizmu kontroli zakresu tablic. Wskutek tego jest możliwe np. wpisywanie wartości do elementów tablicy poza zakresem wcześniej zadeklarowanych indeksów. Ten niepożądany efekt odnosi się zarówno do przekroczenia zakresu od dołu (ang. underflow), jak i od góry (ang. overflow), i to w równym stopniu do tablic alokowanych w pamięci statycznej, na stosie funkcji, czy też w przypadku alokacji dynamicznej w pamięci swobodnej. I tutaj, jak w wielu innych przypadkach, przychodzi nam z pomocą mechanizm klas. W podanym niżej przykładzie zdefiniowano trzy klasy tablic z elementami typu **int**, **double** oraz **char***. Wszystkie trzy klasy zawierają dwa rodzaje funkcji składowych – pierwsza, `wstaw()`, służy do zapisywania informacji w tablicy, zaś druga, `pobierz()`, służy do wyszukiwania informacji w tablicy. Te dwie funkcje są w stanie sprawdzać w fazie wykonania programu, czy zakresy tablicy nie zostały przekroczone.

Przykład 6.37.

```
// Plik TABLICA.H pod DOS, tablica.h pod Unix
// Bezpieczna tablica: elementy typu int
class tabint   {
public:
  tabint(int liczba);
  int& wstaw(int i);
  int pobierz(int i);
private:
  int rozmiar, *wsk;
};

// Bezpieczna tablica: elementy typu double
class tabdouble  {
  double* wsk;
public:
  tabdouble(int liczba);
  double& wstaw(int i);
  double pobierz(int i);
private:
  int rozmiar;
};

// Bezpieczna tablica: elementy typu char
class tabchar  {
public:
  tabchar(int liczba);
  char& wstaw(int i);
  char pobierz(int i);
private:
  int rozmiar;
  char* wsk;
};

// Plik Tablica.CPP pod DOS, tablica.c pod Unix
// Definicje funkcji klas tabint, tabdouble, tabchar
#include "tablica.h"
#include <iostream.h>
#include <stdlib.h>
// Funkcje klasy tabint
// Konstruktor
tabint::tabint(int liczba)  {
  wsk = new int [liczba];
  if (!wsk)      {
    cout  << "Brak alokacji\n";
    exit (1);    }
  rozmiar = liczba;         }

// Wstaw element do tablicy.
int& tabint:: wstaw (int i) {
  if(i < 0 || i >= rozmiar)  {
     cout << "Przekroczenie zakresu!!!\n";
     exit (1);
    }
  return wsk[i]; // zwraca referencje do wsk[i]
 }
```

```
// Pobierz element z tablicy.
int tabint::pobierz(int i)  {
   if (i < 0 || i > rozmiar)
    {
     cout << "Przekroczenie zakresu!!!\n";
     exit(1);
    }
   return wsk[i]; // zwraca element
  }

// Funkcje klasy tabdouble
 tabdouble::tabdouble(int liczba)
 {
  wsk = new double [liczba];
  if (!wsk)
   {
    cout  << "Brak alokacji\n";
    exit (1);
    }
  rozmiar = liczba;
}

double& tabdouble:: wstaw (int i)
 {
  if(i < 0 || i >= rozmiar)
    {
     cout << "Przekroczenie zakresu!!!\n";
     exit (1);
    }
  return wsk[i]; // zwraca referencje do wsk[i]
 }

 double tabdouble::pobierz(int i)
  {
   if (i < 0 || i > rozmiar)
    {
     cout << "Przekroczenie zakresu!!!\n";
     exit(1);
    }
   return wsk[i]; // zwraca element
  }

// Funkcje klasy tabchar
tabchar::tabchar(int liczba)   {
  wsk = new char [liczba];
  if (!wsk)      {
    cout  << "Brak alokacji\n";
    exit (1);    }
  rozmiar = liczba;
}

char& tabchar:: wstaw (int i)  {
  if(i < 0 || i >= rozmiar)    {
     cout << "Przekroczenie zakresu!!!\n";
     exit (1);                 }
  return wsk[i]; // zwraca referencje do wsk[i]
 }
```

```
char tabchar::pobierz(int i)
  {
   if (i < 0 || i > rozmiar)
    {
     cout << "Przekroczenie zakresu!!!\n";
     exit(1);
    }
   return wsk[i]; // zwraca element
  }

// Plik MAINTAB.CPP pod DOS, maintab.c pod Unix
#include "tablica.h"
#include <iostream.h>
int   main() {
      tabint tab(5);
      tab.wstaw(3) = 13;
      tab.wstaw(2) = 12;
      cout << tab.pobierz(3) << endl
           << tab.pobierz(2) << endl;
//    Teraz przekroczenie zakresu
      tab.wstaw(6) = '!';
      return 0;
}
```

7.9. Szablony klas i funkcji

Szablony lub wzorce (ang. template), nazywane również typami parametryzowanymi, pozwalają definiować wzorce dla tworzenia klas i funkcji. Dla lepszego zrozumienia podstawowych koncepcji posłużymy się analogią z matematyki.

Matematyka operuje często pojęciem równania parametrycznego, które pozwala generować jedno- lub wieloparametrowe rodziny krzywych i prostych. W takich równaniach występują ogólne wyrażenia matematyczne, których wartości są zależne od zmiennych lub stałych, nazywanych parametrami. Tak więc parametr można określić jako zmienną lub stałą, która wyróżnia przypadki szczególne ogólnego wyrażenia. Np. ogólna postać równania prostej

`y = mx + b`

gdzie `m` jest stałą, pozwala generować rodzinę prostych równoległych o nachyleniu `m`. Podobnie równanie

`(x-a)2+(y-b)2=r2`

przy ustalonej wartości `r`, służy do generacji rodziny okręgów o położeniu środka zależnym od parametrów `a` i `b`.

Koncepcja klasy nawiązuje w pewnym stopniu do tych idei. Klasę można traktować jako generator rodziny obiektów, w której każdemu obiektowi przydziela się niejawny parametr, ustalający jego tożsamość. Ponadto rodzinę obiektów można parametryzować przez nadawanie różnych wartości ich zmiennym składowym. Można wtedy wyróżnić dwa charakterystyczne przypadki.

1. Obiekt jest tworzony za pomocą konstruktora generowanego przez kompilator.
2. Obiekt jest tworzony za pomocą konstruktora zdefiniowanego przez programistę.

W pierwszym przypadku zmiennym składowym różnych obiektów można nadawać różne wartości po uprzednim utworzeniu obiektów z wartościami domyślnymi. W przypadku drugim użytkownik ma więcej możliwości: może on np. zdefiniować konstruktor z pustą listą argumentów, z argumentami domyślnymi, bądź definiować konstruktory przeciążone.

Wszystko to jednak dzieje się na poziomie jednej rodziny obiektów, w ramach jednej definicji klasy. Nasuwa się w związku z tym pytanie: czy można zmienić deklarację klasy w taki sposób, aby stworzyć sobie możliwość generowania wielu rozłącznych rodzin obiektów?

Skoro zaczęliśmy od analogii matematycznej dla schematu klasa-rodzina obiektów, spróbujmy poszukać następnej analogii. W geometrii analitycznej rozważa się m.in. przekroje stożka kołowego płaszczyznami. W zależności od nachylenia płaszczyzny tnącej otrzymuje się szereg rodzin parametryzowanych krzywych: rodzinę okręgów, rodzinę elips, rodzinę parabol i rodzinę hiperbol.

Być może, iż takie właśnie analogie nasuwały się twórcom obowiązującej obecnie wersji 4.0 kompilatora języka C++. Wprowadzona w tej wersji deklaracja szablonu ma następującą postać ogólną:

```
template<argumenty> deklaracja
```

gdzie:

- słowo kluczowe (specyfikator) **template** sygnalizuje kompilatorowi wystąpienie deklaracji szablonu,
- słowo `argumenty` oznacza listę argumentów (parametrów) szablonu,
- słowo `deklaracja` oznacza deklarację klasy lub funkcji.

Lista argumentów szablonu może się składać z rozdzielonych przecinkami argumentów, przy czym każdy argument musi być postaci `class nazwa`, lub deklaracją argumentu. Wynika stąd, że nazwy argumentów mogą się odnosić zarówno do klas, jak i do typów wbudowanych, bądź zdefiniowanych przez użytkownika.
Deklaracja szablonu może występować jedynie jako deklaracja globalna, a nazwy używane w szablonie stosują się do wszystkich reguł dostępu i kontroli.

7.9.1. Szablony klas

Deklarację szablonu można wykorzystywać do definiowania klas wzorcowych. W takim przypadku po zamykającym nawiasie kątowym umieszcza się definicję odpowiedniej klasy, np.

```
template <class T> class stream { /* ... */ };
```

Występująca we wzorcu nazwa klasy `stream` może być następnie używana wszędzie tam, gdzie dopuszcza się używanie nazwy klasy, np. w bloku funkcji możemy zadeklarować wskaźniki do tej klasy:

```
class stream<char>  *firststream, *secondstream;
```

Przykład 6.38.

```
#include <iostream.h>
template<class Typ>
class Tablica {
// Klasa parametryzowana typem elementów tablicy
public:
 Tablica(int); // Deklaracja konstruktora
 ~Tablica() { delete [] tab; } // Destruktor
 Typ& operator [] (int j) { return tab[j]; }
private:
 Typ* tab; // Wskaz do tablicy
 int rozmiar;
};

template<class Typ>
Tablica<Typ>::Tablica(int n) // Konstruktor
 { tab = new Typ[n]; rozmiar = n; }
int  main() {
//    Tablica 10-elementowa. Elementy typu int
      Tablica<int> x(10);
      for(int i = 0; i < 10; ++i) x[i] = i;
      for(i = 0; i < 10; ++i) cout << x[i] << ' ';
      cout << endl;
```

```
//    Tablica 5-elementowa. Elementy typu double
      Tablica<double> y(5);
//    Tablica 6-elementowa. Elementy typu char
      Tablica<char> z(6);
      return 0;
}
```

Dyskusja. Każde wystąpienie nazwy klasy `Tablica` wewnątrz deklaracji klasy jest skrótem zapisu `Tablica<Typ>`, np. gdyby deklarację konstruktora zastąpić definicją, to miałaby ona postać:

```
Tablica<int n> {tab = new Typ[n]; rozmiar = n; }
```

Deklaracja `Tablica<int> x(10);` wywołuje konstruktor z parametrem 10 przekazywanym przez wartość. Konstruktor tworzy 10-elementową tablicę dynamiczną o elementach typu **int**. Deklarator tablicy, `[]`, został zdefiniowany w treści klasy jako przeciążony `operator[]`. Dzięki temu obiekty klasy `Tablica<Typ>` można indeksować tak samo, jak zwykłe tablice. Warto również zwrócić uwagę na zwięzłość zapisu: deklaracje tablic typu **double** oraz **char** korzystają z tego samego szablonu, co tablica typu **int** – żadna z nich nie wymaga uprzedniej definicji.
Gdyby pisanie nazwy klasy wzorcowej z obowiązującymi nawiasami kątowymi okazało się nużące dla użytkownika, można użyć konstrukcji z **typedef** dla wprowadzenia synonimów. Można np. wprowadzić zastępcze nazwy deklaracjami

```
typedef Tablica<int>  tabint;
typedef Tablica<char> tabchar;
```

i stosować te nazwy dla tworzenia odpowiednich tablic

```
tabint   x(10);
tabchar  z(6);
```

7.9.2. Szablony funkcji

Szablon funkcji określa sposób konstrukcji poszczególnych funkcji. Np. rodzina funkcji sortujących może być zadeklarowana w postaci:

```
template <class Typ>  void sort(Tablica<Typ>);
```

Szablon funkcji wyznacza de facto nieograniczony zbiór (przeciążonych) funkcji. Generowaną z szablonu funkcję nazywa się funkcją wzorcową. Przy wywołaniach funkcji wzorcowych nie podaje się jawnie argumentów wzorca. Zamiast tego używa się mechanizmu rozpoznawania (ang. resolution) wywołania. Dopuszcza się przy tym przeciążanie funkcji wzorcowej przez zwykłe funkcje o takiej samej nazwie, lub przez inne funkcje wzorcowe o takiej samej nazwie. W omawianych przypadkach stosowany jest następujący algorytm rozpoznawania.

1. Sprawdź, czy istnieje dokładne dopasowanie wywołania do prototypu funkcji. Jeżeli tak, to wywołaj funkcję.
2. Poszukaj szablonu funkcji, z którego może być wygenerowana funkcja dokładnie dopasowana do wywołania. Jeżeli znajdziesz taki szablon, wywołaj go.
3. Wypróbuj zwykłe metody dopasowania argumentów (promocja,konwersja) dla funkcji przeciążonych. Jeżeli znajdziesz odpowiednią funkcję, wywołaj ją.

Taki sam proces rozpoznawania jest stosowany dla wskaźników do funkcji.
Podany niżej elementarny przykład jest ilustracją kroku 2 algorytmu. Jedyną, jak się wydaje, korzyścią z zastosowanego tutaj szablonu jest krótki kod źródłowy. Gdybyśmy zastosowali poznany wcześniej mechanizm przeciążania funkcji, to byłoby konieczne podanie trzech oddzielnych definicji odpowiednio dla typów **int**, **double** oraz **char**.

Przykład 6.39.

```
#include <iostream.h>
// Szablon
template <class T>
 T max( T a, T b ) { return a > b ? a : b; }
int  main() {
     int i = 2, j = max(i,0);
     cout << "Max integer: " << j << endl;
     double db = 1.7, dc = max(db,3.14);
     cout << "Max double: " << dc << endl;
     char z1 = 'A', z2 = max(z1,'B');
     cout << "Max char: " << z2 << endl;
     return 0;
}
```

Interesującym przykładem zastosowania szablonów może być definicja funkcji, której zadaniem jest kopiowanie sekwencyjnych struktur danych. Definicję tę zastosujemy do kopiowania sekwencji znaków.

Przykład 6.40.

```
#include <iostream.h>
template < class We, class Wy >
Wy copy ( We start, We end, Wy cel )
{
  while ( start != end )
     *cel++=*start++;
  return cel;
}
void main () {
     char* hello = "Hello ";
     char* world = "world";
     char komunikat[15];
     char* wsk = komunikat;
     wsk = copy ( hello, hello+6, wsk );
     wsk = copy ( world, world+5, wsk );
     *wsk = '\0';
     cout << komunikat << endl;
}
```

Analiza programu. Pierwsze wywołanie funkcji copy kopiuje wartość "Hello" do pierwszych sześciu znaków (zwróćmy uwagę na spację po znaku 'o') tablicy `komunikat`, zaś druga kopiuje wartość "world" do następnych pięciu znaków tablicy. Po tych operacjach zmienna p wskazuje na znak (nieokreślony, ponieważ tablica `komunikat` nie została zainicjowana) występujący bezpośrednio po literze 'd' słowa "world". Następnie do wskazania `*p` przypisujemy znak zerowy '\0', aby otrzymać normalną postać łańcucha stosowaną w języku C++. Ostatnią instrukcją wstawiamy zmienną `komunikat` do strumienia `cout`.

7.10. Klasy zagnieżdżone

Klasa może być deklarowana wewnątrz innej klasy. Jeżeli klasę B zadeklarujemy wewnątrz klasy A, to klasę B nazywamy *zagnieżdżoną* w klasie A. Klasa zagnieżdżona jest widoczna tylko w zasięgu związanym z deklaracją klasy wewnętrznej. Miejscem deklaracji klasy zagnieżdżonej może być część publiczna, prywatna, lub chroniona klasy otaczającej. Zauważmy, że zagnieżdżenie jest przykładem asocjacji: klasa wewnętrzna jest deklarowana jedynie w tym celu, aby umożliwić wykorzystanie jej zmiennych i funkcji składowych klasie zewnętrznej. Typowym przykładem może tu być asocjacja

grupująca; np. w klasie `Komputer` możemy zagnieździć klasy `Procesor` i `Pamięć`. Jeżeli pamięć potraktujemy jako tablicę jednowymiarową, to wykonanie operacji `dodaj liczbę a do liczby b` będzie wymagać zadeklarowania obiektu klasy `Procesor` z funkcjami `pobierz()` i `wykonaj()` oraz z licznikiem rozkazów, który będzie wskazywał kolejne adresy pamięci.

Przykład 6.41.

```
#include <iostream.h>
class Otacza {
public:
 class Zawarta {
 public:
   Zawarta(int i): y(i) {}  //Konstruktor
   int podajy() { return y;  }
 private:
   int y;
 };
 Otacza(int j): x(j){}  //Konstruktor
 int podajx() { return x; }
private:
 int x;
};
int  main() {
     Otacza zewn(7);
     int m,n;
     m = zewn.podajx();
     cout << m << endl;
     Otacza::Zawarta wewn(9);
     n = wewn.podajy();
     cout << n << endl;
     return 0;
}
```

Dyskusja. Instrukcja deklaracji `Otacza zewn(7);` wywołuje konstruktor klasy zewnętrznej `Otacza(int j){x=j;}` z parametrem aktualnym `j=7`, inicjując `x` na wartość 7. Instrukcja `m=zewn.podajx();` wywołuje funkcję składową `podajx()` klasy zewnętrznej. Obiekt `wewn` klasy `Zawarta` jest tworzony instrukcją deklaracji `Otacza::Zawarta wewn(9);`.

7.11. Struktury i unie jako klasy

W języku C++ struktury i unie są w zasadzie "pełnoprawnymi" klasami. Poniżej wyliczono te cechy struktur i unii, które wyróżniają je w stosunku do klas, deklarowanych ze słowem kluczowym **class**.

- Struktura jest klasą, deklarowaną ze słowem kluczowym **struct**; elementy składowe struktury oraz jej klasy bazowe są domyślnie publiczne.
- Unia jest klasą, deklarowaną ze słowem kluczowym **union**; elementy składowe unii są domyślnie publiczne; unia utrzymuje w danym momencie czasu tylko jedną składową.
- Struktura może zawierać funkcje składowe, włącznie z konstruktorami i destruktorami.
- Unia nie może dziedziczyć żadnej klasy, ani nie może być klasą bazową dla klasy pochodnej.
- Unia nie może zawierać żadnej składowej statycznej, tj. deklarowanej ze słowem kluczowym **static**.
- Unia nie może zawierać obiektu, który ma konstruktor lub destruktor. Tym niemniej unia może mieć konstruktor i destruktor.
- Niektóre kompilatory nie akceptują prywatnych składowych unii.

Tak więc struktura różni się od klasy, deklarowanej ze słowem kluczowym **class** jedynie tym, że jeżeli jej składowe nie są poprzedzone etykietą **private:**, to są one publiczne. Oznacza to, że wszystkie

zmienne i funkcje składowe są dostępne za pomocą operatora selekcji '.' lub – w przypadku wskaźnika do obiektu – operatora "->". Wobec" tego deklaracje:

```
class Punkt { public: int x,y; };
```

oraz

```
struct Punkt { int x,y; };
```

są równoważne. Podobnie równoważne będą deklaracje:

```
class Punkt { int x,y; };
```

oraz

```
struct Punkt { private: int x,y; };
```

Pokazany niżej przykład jest nieznaczną modyfikacją poprzedniego; w programie wyeliminowano funkcje składowe `podajx()` oraz `podajy()`, ponieważ składowe `x` oraz `y` są teraz publiczne, a więc dostępne bezpośrednio.

Przykład 6.42.

```
#include <iostream.h>
struct Otacza {
 int x;
 struct Zawarta
 {
  int y;
  Zawarta(int i): y(i) {}  //Konstruktor
 };
 Otacza(int j): x(j) {}  //Konstruktor
};
int   main() {
      Otacza zewn(7);
      int m,n;
      cout << zewn.x << endl;
      Otacza::Zawarta wewn(9);
      cout << wewn.y << endl;
      return 0;
}
```

Chociaż struktury mają w zasadzie te same możliwości co klasy, większość programistów stosuje struktury bez funkcji składowych. Są one wtedy odpowiednikami struktur języka C, bądź rekordów, definiowanych w językach Pascal, czy Modula-2. Zaletą takiego stylu programowania jest większa czytelność programów: tam, gdzie nie jest wymagane ukrywanie informacji, używa się struktur – w przeciwnym przypadku – klas.

Pamiętamy z wcześniejszego wprowadzenia, że wprawdzie unia może grupować składowe różnych typów, ale tylko jedna ze składowych może być "aktywna" w danym momencie. Jest to cecha, istotna w aspekcie ukrywania informacji: używając unii możemy tworzyć klasy, w których wszystkie dane współdzielą ten sam obszar w pamięci. Jest to coś, czego nie da się zrobić, używając klas, deklarowanych ze słowem kluczowym **class**.

Przykład 6.43.

```
#include <iostream.h>
union bity {
 bity (int n); //Deklaracja konstruktora
 void podajbity();
 int d;
 unsigned char z[sizeof(int)];
};
bity::bity(int n): d(n) {} //Definicja konstruktora
void bity::podajbity()
 {
  int i,j;
  for (j = sizeof(int)-1; j >= 0; j--)
  {
   cout << "Wzorzec bitowy w bajcie " << j << ": ";
   for (i = 128; i; i >>= 1)
    if (i & z[j]) cout << "1";
    else cout << "0";
   cout << "\n";
  }
 }
int  main() {
     bity obiekt(255);
     obiekt.podajbity();
     return 0;
}
```

Dyskusja. Przykład prezentuje wykorzystanie unii do wyświetlenia układu bitów w kolejnych bajtach liczby (255), podanej w wywołaniu konstruktora. Wykonanie programu powinno dać wydruk o postaci:

Wzorzec bitowy w bajcie 1: 00000000
Wzorzec bitowy w bajcie 0: 11111111

7. Dziedziczenie i hierarchia klas

Definiowane dotąd klasy były jednostkowymi konstrukcjami programistycznymi. Obiekty takich klas funkcjonowały niezależnie jeden od drugiego, podobnie jak zmienne innych typów.
Jednakże w praktyce, gdy zaczynamy analizować klasy, które mają jedną lub więcej cech wspólnych, to dochodzimy często do wniosku, że warto byłoby je w jakiś sposób uporządkować. Narzucającym się natychmiast sposobem uporządkowania jest generalizacja albo uogólnienie: należy zdefiniować taką klasę, która będzie zawierać tylko te atrybuty i operacje, które są wspólne dla pewnej grupy klas. W językach obiektowych taką uogólnioną klasę nazywa się *superklasą* lub *klasą rodzicielską*, a każdą z klas związanej z nią grupy nazywa się *podklasą* lub *klasą potomną*.

7.1. Klasy pochodne

W języku C++ odpowiednikami tych terminów są *klasa bazowa* (ang. base class) i *klasa pochodna* (ang. derived class). Tak więc klasa bazowa może zawierać tylko te elementy składowe, które są wspólne dla wyprowadzanych z niej klas pochodnych. Własność tę wyraża się zwykle w inny sposób: mówimy, że klasa pochodna *dziedziczy* wszystkie cechy swojej klasy bazowej.
Gdyby dziedziczenie ograniczyć jedynie do przekazywania klasie pochodnej cech klasy bazowej, to taki mechanizm byłby raczej mało przydatnym kopiowaniem, albo czymś, co przypomina klonowanie. Dlatego też w języku C++ mechanizm dziedziczenia wzbogacono o następujące możliwości.

1. W klasie pochodnej można dodawać nowe zmienne i funkcje składowe.
2. W klasie pochodnej można redefiniować funkcje składowe klasy bazowej.

Tak określony mechanizm pozwala w naszej abstrakcji zbliżyć się do tego, co dziedziczenie oznacza w języku potocznym: sensownie określone klasy pochodne zawierają cechy klasy (lub kilku klas) bazowej oraz nowe cechy, które wyraźnie odróżniają je od klasy bazowej.

☞*Uwaga. Terminy "superklasa" i "podklasa" mogą być mylące dla osób, które obserwują, że obiekt klasy pochodnej zawiera obiekt klasy bazowej jako jeden ze swoich elementów i że klasa pochodna jest większa niż jej klasa bazowa w tym sensie, że mieści w sobie więcej danych i funkcji. Zatem to klasa pochodna jest nadzbiorem dla klasy bazowej, a nie odwrotnie.*

Byłoby trudno przecenić znaczenie dziedziczenia w programowaniu obiektowym. Każdą klasę pochodną można wykorzystać jako klasę bazową dla następnej klasy pochodnej. Zatem schemat dziedziczenia pozwala budować hierarchiczne, wielopoziomowe struktury klas, w których każda klasa pochodna ma swoją bezpośrednią klasę bazową. Jeżeli każda klasa pochodna dziedziczy cechy tylko jednej klasy bazowej, to otrzymuje się strukturę drzewiastą. Jeżeli klasa pochodna dziedziczy cechy wielu klas bazowych, to otrzymuje się strukturę, nazywaną grafem acyklicznym. W obu tych hierarchicznych strukturach przejście od poziomu najwyższego do najniższego oznacza przejście od klasy najbardziej ogólnej do klasy najbardziej szczegółowej.
Wymienione struktury hierarchiczne przechowuje się zwykle w bibliotekach klas, co pozwala na wielokrotne ich wykorzystanie.
Hierarchie klas są również wygodnymi narzędziami dla tzw. szybkiego prototypowania (ang. rapid prototyping). Jest to technika szybkiego opracowania modelu systemu, jeszcze przed rozpoczęciem głównych prac implementacyjnych. Taki działający model systemu pozwala skonfrontować wymagania użytkownika z propozycjami projektanta, skraca czas projektowania i z reguły znacznie obniża koszty.

7.1.1. Dziedziczenie pojedyncze

Deklaracja klasy pochodnej, która dziedziczy cechy tylko jednej klasy bazowej, ma następującą postać:

```
class pochodna : specyfikator-dostępu bazowa
```

```
{
 //...
};
```

gdzie: dwukropek po nazwie klasy pochodnej wskazuje, że klasa pochodna wywodzi się od wcześniej zdefiniowanej klasy bazowa;
specyfikator-dostępu może być jednym z trzech słów kluczowych: **public**, **private**, lub **protected**.

Zasięg klasy `pochodna` mieści się wewnątrz zasięgu klasy `bazowa`. Podobnie, jak dla klas nie związanych relacją dziedziczenia, widoczność funkcji składowych klas `bazowa` i `pochodna` nie wykracza poza zasięg ich klas.
Umieszczony przed nazwą klasy bazowej specyfikator dostępu ustala, jak elementy klasy bazowej są dziedziczone przez klasę pochodną.

1. Jeżeli specyfikatorem jest **public**, to informujemy kompilator, że klasa pochodna ma dziedziczyć wszystkie elementy klasy bazowej i że wszystkie elementy publiczne w klasie bazowej będą także publicznymi elementami klasy pochodnej. Natomiast wszystkie elementy prywatne klasy bazowej pozostaną jej wyłączną własnością i nie będą bezpośrednio dostępne w klasie pochodnej.
2. Jeżeli specyfikatorem jest **private**, to wszystkie elementy publiczne klasy bazowej staną się prywatnymi elementami klasy pochodnej. Podobnie, jak w pierwszym przypadku, elementy prywatne klasy bazowej pozostaną jej wyłączną własnością i nie będą bezpośrednio dostępne w klasie pochodnej. Dodajmy jeszcze, że specyfikator private jest specyfikatorem domyślnym; jeżeli przed nazwą klasy bazowej nie umieścimy żadnego specyfikatora, to kompilator przyjmie private. Dla czytelności zapisu nie należy jednak pomijać słowa private.
3. Użycie specyfikatora **protected** wymaga nieco szerszego omówienia. Z punktów 1 i 2 wynika, że klasa pochodna ma dostęp tylko do elementów publicznych klasy bazowej. W przypadku, gdy chcemy pozostawić element klasy bazowej prywatnym, ale dać do niego dostęp klasie pochodnej, wtedy deklarację takiego elementu w klasie bazowej poprzedzamy etykietą **protected:**, np.
   ```
   class bazowa  {
   int pryw;
   protected:
   int chron;
   public:
   int publ;
   };
   ```
   Jeżeli tak zadeklarowana klasa bazowa jest dla klasy pochodnej:
   - publiczna, to zmienna `chron` stanie się elementem chronionym w klasie pochodnej, tzn. jej elementem prywatnym, ale dostępnym dla jej własnych klas pochodnych;
   - prywatna, to zmienna `chron` stanie się prywatnym elementem klasy pochodnej (podobnie, jak zmienna `publ`);
   - chroniona, to chronione (`chron`) i publiczne (`publ`) elementy klasy bazowej staną się chronionymi elementami klasy pochodnej.

Z powyższej dyskusji wynikają dwa wnioski ogólne. Po pierwsze, wszystkie elementy publiczne każdej klasy w hierarchii są dostępne dla każdej klasy, leżącej niżej w hierarchii, i dla dowolnej funkcji, nie mającej związku z daną hierarchią klas. Po drugie, elementy prywatne pozostają własnością klasy, w której zostały zadeklarowane. Elementy chronione są równoważne prywatnym w danej klasie bazowej, ale są dostępne dla jej klas pochodnych.

7.1.2. Dziedziczenie z konstruktorami generowanymi

Konstruktory i destruktory nie są dziedziczone (podobnie jak funkcje i klasy zaprzyjaźnione). Natomiast konstruktory niejawne są zawsze generowane przez kompilator dla obiektów każdej klasy w hierarchii dziedziczenia. Są to zawsze funkcje, poprzedzone domyślnym specyfikatorem **public**.

Przykład 7.1.

```
#include <iostream.h>
class Bazowa  {
public:
  void ustawx(int n): x(n) {}
  void podajx() { cout << x << "\n"; }
private:
  int x,z;
};
class Pochodna : public Bazowa {
public:
  void ustawy(int m)  { y = m; }
  void podajy() { cout << y << "\n"; }
private:
  int y;
};
int main() {
    Pochodna ob;
    ob.ustawx(10);
    ob.ustawy(20);
    ob.podajx();
    ob.podajy();
    cout << sizeof(Bazowa) << endl;
    cout << sizeof(Pochodna) << endl;
    cout << sizeof ob << endl;
    return 0;
}
```

Dyskusja. Wydruk z powyższego programu ma postać:

10
20
4
6
6

Obiekt `ob` klasy `Pochodna` nie ma dostępu do prywatnej składowej `x` klasy `Bazowa`, ale ma dostęp do publicznych funkcji `ustawx()` oraz `podajx()`. Funkcje te są dostępne z "wnętrza" obiektu `ob`. Operatory **sizeof** wykorzystano tutaj dla pokazania, że obiekt klasy bazowej jest częścią obiektu klasy pochodnej (w pokazanej implementacji zmienna całkowita zajmuje 2 bajty w pamięci).

7.1.3. Dziedziczenie z konstruktorami definiowanymi

Prześledzimy teraz przypadki, gdy klasy bazowa i pochodna mają konstruktory i destruktory zdefiniowane przez użytkownika. Można wtedy pokazać, że konstruktory są wykonywane w porządku dziedziczenia (najpierw jest wykonywany konstruktor klasy bazowej, a następnie pochodnej). Destruktory są wykonywane w kolejności odwrotnej. Kolejność ta jest oczywista: ponieważ klasa bazowa nie ma żadnej informacji o jakiejkolwiek klasie pochodnej, musi inicjować swoje obiekty niezależnie. W odniesieniu do klasy pochodnej jest to czynność przygotowawcza dla zainicjowania jej własnego obiektu. Dlatego konstruktor klasy bazowej musi być wykonywany jako pierwszy. Z drugiej strony, destruktor klasy pochodnej musi być wykonany przed destruktorem klasy bazowej. Gdyby zamienić kolejność, to destruktor klasy bazowej zniszczyłby część obiektu klasy pochodnej, uniemożliwiając jego prawidłową destrukcję.

Przykład 7.2.

```
#include <iostream.h>
class Bazowa {
public:
  Bazowa() { cout<<"Konstruktor klasy bazowej\n"; }
  ~Bazowa(){ cout<<"Destruktor klasy bazowej\n"; }
};
class Pochodna : public Bazowa {
public:
  Pochodna(){ cout<<"Konstruktor klasy pochodnej\n"; }
  ~Pochodna(){ cout<<"Destruktor klasy pochodnej\n"; }
};
int main() {
    Pochodna ob;
    return 0;
}
```

Wydruk z powyższego programu ma postać:

Konstruktor klasy bazowej
Konstruktor klasy pochodnej
Destruktor klasy pochodnej
Destruktor klasy bazowej

W podanym wyżej przykładzie mieliśmy konstruktory z pustymi listami argumentów. Na ogół jednak konstruktory występują z parametrami, których zadaniem jest ustalenie stanu początkowego obiektu. W takich przypadkach używa się rozszerzonej postaci deklaracji konstruktora klasy pochodnej:

```
Pochodna(arg1) : Bazowa(arg2)
{  ciało konstruktora klasy Pochodna  }
```

gdzie: `arg1` jest łącznym wykazem argumentów dla konstruktorów klas `Pochodna` i `Bazowa`, zaś `arg2` jest wykazem argumentów konstruktora klasy `Bazowa`.

Pokazana wyżej postać deklaracji pozwala przekazać do konstruktora klasy `Pochodna` zarówno parametry aktualne, specyficzne dla obiektów tej klasy, jak i parametry aktualne dla wywołania konstruktora klasy `Bazowa`.

Przykład 7.3.

```
#include <iostream.h>
class Bazowa {
public:
  Bazowa(int i, int j): x(i),y(j) {}
  void podaj();
private:
  int x,y;
};
void Bazowa::podaj()
{  cout << "x= " << x << " y= " << y << endl;  }

class Pochodna : public Bazowa {
public:
  Pochodna(int m, int n) : Bazowa(m,n)  {}
};
int main() {
    Bazowa ba(3,4);
    ba.podaj();
    Pochodna po(5,9);
    po.podaj();
    return 0;
}
```

Dyskusja. Ponieważ klasa `Pochodna` nie zawiera żadnych rozszerzeń w stosunku do klasy `Bazowa`, ciało konstruktora klasy `Pochodna` jest puste, zaś przyjmowane przez tę funkcję dwa parametry są przekazywane do konstruktora klasy `Bazowa`. Przesłanie parametrów aktualnych do konstruktora klasy `Bazowa` jest konieczne, ponieważ, jak pamiętamy, konstruktory nie są dziedziczone. Instrukcja deklaracji `Bazowa ba(3,4);` wywołuje konstruktor `Bazowa::Bazowa(int,int)`. Natomiast instrukcja deklaracji
`Pochodna  po(5,9);`
wywołuje najpierw konstruktor klasy `Pochodna`, który przekazuje parametry aktualne do konstruktora klasy `Bazowa`. Zgodnie z kolejnością wykonywania konstruktorów, najpierw zostanie wykonany konstruktor
`Bazowa::Bazowa(5,9)`, a następnie konstruktor klasy `Pochodna`.
Ponieważ funkcja `podaj()` jest publiczną funkcją składową klasy `Bazowa`, jest ona wykorzystywana w programie zarówno dla obiektu `ba` klasy `Bazowa`, jak i dla obiektu `po` klasy `Pochodna`.

Poprzedni przykład miał znaczenie czysto dydaktyczne, ponieważ klasa pochodna nie zawierała żadnych zmian w stosunku do klasy bazowej. W następnym przykładzie deklarację klasy pochodnej rozszerzono o dodatkowe elementy.

Przykład 7.4.

```
#include <iostream.h>
class Bazowa {
public:
  Bazowa(int i, int j): x(i),y(j) {}
  void podajxy();
private:
  int x,y;
};
void Bazowa::podajxy()
{  cout << "x= " << x << " y= " << y << endl;  }
```

```
class Pochodna : public Bazowa {
public:
  Pochodna(int k, int m, int n) : Bazowa(m,n)
  { z = k; }
  void podaj()
  {
   podajxy(); //Bazowa::podajxy()
   cout << "z= " << z << endl;
  }
private:
  int z;
};

int main() {
    Pochodna  po(3,100,200);
    po.podaj();
    return 0;
}
```

Dyskusja. Konstruktor klasy `Bazowa` wymaga dwóch argumentów dla zainicjowania zmiennych składowych `x` oraz `y`. Klasa `Pochodna` zawiera dodatkowo zmienną `z`. Dlatego konstruktor klasy `Pochodna` jest funkcją o trzech argumentach, ponieważ dwa spośród nich muszą być przesłane do konstruktora klasy `Bazowa`. Tak więc wykonanie instrukcji deklaracji:
`Pochodna  po(3,100,200);`
powoduje przekazanie parametrów aktualnych 100 i 200 do argumentów konstruktora klasy `Bazowa`, wykonanie konstruktora `Bazowa::Bazowa(i,j)`, a następnie wykonanie konstruktora klasy `Pochodna`.
Zwróćmy uwagę na funkcje składowe `podajxy()` oraz `podaj()`. Założyliśmy tutaj, że dla obiektu klasy `Pochodna` należy podać wartości wszystkich zmiennych składowych. Dlatego w definicji funkcji
`void Pochodna::podaj()`
musieliśmy umieścić wywołanie funkcji `Bazowa::podajxy()`, która jest zdefiniowana w części publicznej klasy `Bazowa`, a więc dostępna w klasie `Pochodna`. Podobne postępowanie musimy zastosować dla identycznych sygnatur funkcji w klasach `Bazowa` i `Pochodna`; jeżeli definicja funkcji `podaj()` z klasy bazowej została przesłonięta przez definicję funkcji o tej samej nazwie w klasie pochodnej, to możemy ją odsłonić przy pomocy operatora zasięgu. Jak pokazano w poniższym przykładzie, dotyczy to również zmiennych składowych klasy bazowej.

Przykład 7.5.

```
#include <iostream.h>
class Bazowa {
public:
  int x,y;
  Bazowa(int i, int j): x(i),y(j) {}
  void podaj();
};
void Bazowa::podaj()
{ cout << "Funkcja podaj() klasy Bazowa" << endl; }
class Pochodna : public Bazowa {
public:
  int x;
  Pochodna(int k,int m,int n):Bazowa(m,n) { x = k; }
  void podaj()
  {
  Bazowa::podaj();
  cout << "Funkcja podaj() klasy Pochodna" << endl;
```

```
  }
};
int main() {
    Pochodna  po(3,100,200);
    po.podaj();
    po.Bazowa::podaj();
    cout << "po.x= " << po.x << endl;
    cout<<"po.Bazowa::x= "<<po.Bazowa::x<<endl;
    return 0;
}
```

Wydruk z programu będzie miał postać:

Funkcja podaj() klasy Bazowa
Funkcja podaj() klasy Pochodna
Funkcja podaj() klasy Bazowa
po.x= 3
po.Bazowa::x= 100

7.2. Konwersje w hierarchii klas

Jeżeli tworzymy obiekty różnych klas, to tworzone obiekty są zmiennymi wystąpienia swoich klas. Każda taka zmienna jest skojarzona z pewnym obszarem pamięci na tyle dużym, aby mógł pomieścić obiekt danej klasy. Obiekty różnych klas będą w ogólności mieć różne rozmiary, a zatem nie można zmieścić obiektu danej klasy w obszarze pamięci, alokowanym dla obiektu innej klasy, która jest jej podzbiorem. Weźmy dla przykładu deklaracje:

Przykład 7.6.

```
class Bazowa {
public:
  int a,b;
  // ...
};
class Pochodna: public Bazowa {
public:
  int c,d;
  // ...
};
```

Obiekty klasy `Bazowa` mają dwie zmienne składowe: `a` oraz `b`. Obiekty klasy `Pochodna` mają cztery zmienne składowe: `a` oraz `b` odziedziczone od klasy `Bazowa` i `c` oraz `d` zadeklarowane w klasie `Pochodna`. Z zasad dziedziczenia wynika, że typ obiektu, wskaźnika, lub referencji, może być niejawnie przekształcony z typu `Pochodna` do publicznego typu `Bazowa`. Jeśli więc zadeklarujemy:

```
Bazowa baz;
Pochodna poch;
```

to dopuszczalne jest przypisanie

```
baz = poch;
```

lub równoważne przypisanie zmiennych składowych

```
baz.a = poch.a
baz.b = poch.b;
```

W powyższych instrukcjach przypisania wartość typu `Pochodna` podlega niejawnej konwersji do typu `Bazowa` przez odrzucenie zmiennych wystąpienia `c` i `d`. Tak więc odwołanie do składowej obiektu klasy pochodnej (w omawianym przypadku było to odczytanie wartości i następnie przypisanie) wymagało zmiany (konwersji) obiektu.

Przykład 7.7.

```
// Konwersja niejawna z Pochodnej do Bazowej
#include <iostream.h>
class Pochodna;
class Bazowa {
public:
  Bazowa(int i): x(i) {}
  void podajx()
   { cout << "x: " << x << endl; }
  Bazowa& operator=(const Bazowa& b)
   { this->x = b.x; return *this; }
protected:
  int x;
};

class Pochodna: public Bazowa {
public:
  Pochodna(int j, int k): Bazowa(j) { y = k; }
  void podaj()
  {
   Bazowa::podajx();
   cout << "y: " << y << endl;
  }
private:
  int y;
};

int main()  {
    Bazowa ba(10);
    Pochodna po(20,30);
    cout << sizeof ba << '\t' << sizeof po << endl;
    ba.podajx();
    po.podaj();
    ba = po;
    ba.podajx();
    Bazowa& refbaz = Pochodna(100,200);
    refbaz.podajx();
    return 0;
}
```

Wydruk z programu ma postać:

2 4
x: 10
x: 20
y: 30
x: 20
x: 100

Dyskusja. Obiekt `ba` jest tworzony przez konstruktor `Bazowa(int)`, zaś obiekt `po` jest tworzony przez konstruktor `Pochodna(int, int)`, który wywołuje ten sam konstruktor `Bazowa(int)`

dla zainicjowania swojego podobiektu klasy `Bazowa`. Z pozostałych instrukcji komentarza wymagają dwie:

`ba = po;`

Jest to konwersja standardowa (niejawna) obiektu klasy `Pochodna` w obiekt klasy `Bazowa`, przy czym przypisanie wykonuje funkcja operatorowa

`Bazowa& operator=(const Bazowa& b)`

wywoływana dla obiektu `ba` (niejawny argument **this**, ulokowany przed argumentem jawnym `const Bazowa& b`) z argumentem referencyjnym `b`.

`Bazowa& refbaz = Pochodna(100,200);`

Jest to również konwersja standardowa, przy której referencja do klasy `Bazowa` jest inicjowana obiektem (tj. jedynie składową `x==100`) klasy `Pochodna`.

Przykład 7.8.

```
// Konwersje jawne
#include <iostream.h>
class Pochodna;
class Bazowa {
public:
  Bazowa(int i): x(i) {  }
  Bazowa(const Bazowa& b) { this->x = b.x; }
  Bazowa& operator=(const Bazowa& b)
  { this->x = b.x; return *this; }
  operator Pochodna();
private:
  int x;
};
class Pochodna: public Bazowa {
public:
  Pochodna(int j, int k):
  Bazowa(j) { y = k; }
private:
  int y;
};
Bazowa::operator Pochodna()
{ return Pochodna(x, 5); }
int main()  {
    Pochodna po(20,30);
    Pochodna poch = po;//zgodne
    Pochodna& refpo = po;//zgodne
    Pochodna* wskpo = &po;//zgodne
    Bazowa ba(50);
    Bazowa bazo = ba;//zgodne
    Bazowa& refba = ba;//zgodne
    Bazowa* wskbaz = &ba;//zgodne
    bazo = po;//konwersja standardowa
    refba = po;//konwersja standardowa
    wskbaz = &po;//konwersja standardowa
    ba = Bazowa(poch);//konwersja standardowa
    poch = ba;//operator konwersji
    poch = Pochodna(ba);//operator konwersji
    refpo = ba;//operator konwersji
//  wskpo = &ba; Niedozwolone
    wskpo = &Pochodna(ba);//operator konwersji
    return 0;
}
```

Dyskusja. W przykładzie pokazano sposób deklarowania i wykorzystania operatorowej funkcji konwersji z klasy pochodnej do klasy bazowej. Dość długa sekwencja instrukcji w bloku `main()` ilustruje różne warianty inicjowania i przypisywania obiektów. Ze względu na tę długą sekwencję instrukcji, zajmiemy się tylko tymi, które wnoszą nowe elementy. Instrukcja:
`poch = ba;` woła operatorową funkcję konwersji
`Bazowa::operator Pochodna() { return Pochodna(x, 5); }`
która przekształca obiekt klasy `Bazowa` w obiekt klasy `Pochodna`. Jest to funkcja operatorowa w zasięgu klasy `Bazowa`, zwracająca obiekt typu `Pochodna`. Dla wykonania tego zadania, a konkretnie wykonania instrukcji `return Pochodna(x, 5);` funkcja ta wywołuje konstruktor klasy Pochodna, który z kolei woła konstruktor `Bazowa(int)`, po czym kończy wykonanie swojego bloku i ponownie przekazuje sterowanie do funkcji konwersji. Ostatnią czynnością przed opuszczeniem bloku tej funkcji jest wywołanie operatora przypisania dla świeżo utworzonego obiektu klasy `Pochodna`.
Dokładnie tak samo są wykonywane instrukcje:
`poch = Pochodna(ba);`
`refpo = ba;`
Podobnie jest wykonywana ostatnia instrukcja `wskpo = &Pochodna(ba);`. Jedyną różnicą jest brak wywołania przeciążonego operatora przypisania. Instrukcja `wskpo = &ba;` jest nielegalna, ponieważ nie ma konwersji z `Bazowa*` do `Pochodna*` (`wskpo++` adresowałby następny obiekt klasy `Pochodna`, a nie następny obiekt klasy `Bazowa`).

7.3. Dziedziczenie mnogie

Klasa pochodna może mieć więcej niż jedną klasę bazową. Możliwe są wtedy dwa schematy dziedziczenia. Schemat pierwszy to proces kaskadowy: klasa C dziedziczy od klasy B, która z kolei dziedziczy od klasy A. W tym przypadku klasa C ma dwie klasy bazowe: bezpośrednią klasę bazową B i pośrednią klasę bazową A. W schemacie drugim klasa pochodna ma dwie lub więcej bezpośrednich klas bazowych, które z kolei mogą mieć tę samą bezpośrednią klasę bazową, lub różne klasy bazowe. Schemat pierwszy sprowadza się do dziedziczenia pojedynczego w układzie klas A-B, a następnie B-C. Podobnie jak dla układu dwupoziomowego, konstruktory wszystkich trzech klas są wołane w porządku dziedziczenia: najpierw konstruktor klasy A, następnie B, a na końcu konstruktor klasy C.

Przykład 7.9.

```
#include <iostream.h>
class Bazowa {
public:
  Bazowa()
  { cout<<"Konstrukcja obiektu klasy Bazowa\n"; }
  ~Bazowa()
  { cout << "Destrukcja obiektu klasy Bazowa\n"; }
};
class Pochodna1: public Bazowa {
public:
  Pochodna1() {
  cout<<"Konstrukcja obiektu klasy Pochodna1\n"; }
  ~Pochodna1() {
  cout<<"Destrukcja obiektu klasy Pochodna1\n"; }
};
class Pochodna2: public Pochodna1 {
public:
  Pochodna2() {
  cout<<"Konstrukcja obiektu klasy Pochodna2\n"; }
  Pochodna2() {
  cout<<"Destrukcja obiektu klasy Pochodna2\n"; }
```

```
};
int main()  {
    Pochodna2 obiekt;
    return 0;
}
```

Dyskusja. W programie mamy trzy poziomy hierarchii dziedziczenia:
Bazowa-Pochodna1-Pochodna2.
Instrukcja deklaracji `Pochodna2 obiekt;` wywołuje konstruktor bezparametrowy `Pochodna2()`. Ponieważ jedną ze składowych obiektu klasy `Pochodna2` będzie obiekt klasy `Pochodna1`, wykonanie tego konstruktora zostaje zawieszone do czasu wykonania konstruktora `Pochodna1()`. Z kolei wykonanie tego konstruktora zostaje zawieszone do czasu utworzenia obiektu klasy `Bazowa`. Po wykonaniu funkcji `Bazowa()`, zostanie dokończone wykonanie funkcji `Pochodna1()`, a następnie funkcji `Pochodna2()`. W rezultacie obiekt klasy `Pochodna2`, będzie zawierał dwa pod-obiekty: pod-obiekt klasy `Bazowa` oraz pod-obiekt klasy `Pochodna1`. Wydruk z programu będzie miał postać:

Konstrukcja obiektu klasy Bazowa
Konstrukcja obiektu klasy Pochodna1
Konstrukcja obiektu klasy Pochodna2
Destrukcja obiektu klasy Pochodna2
Destrukcja obiektu klasy Pochodna1
Destrukcja obiektu klasy Bazowa

Omówimy teraz pokrótce zagadnienia, związane z drugim schematem dziedziczenia mnogiego.
Zacznijmy od notacji. Gdy klasa pochodna dziedziczy bezpośrednio od kilku klas bazowych, to instrukcja deklaracji takiej klasy ma postać:

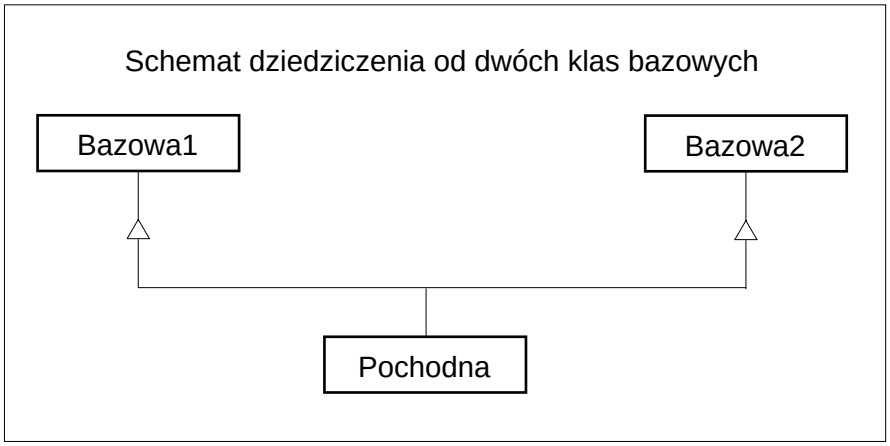
```
class Pochodna: dostęp Bazowa1,
                dostęp Bazowa2,
                ..., dostęp BazowaN
{
 //... ciało klasy Pochodna
};
```

gdzie `Bazowa1` do `BazowaN` są nazwami klas bazowych, zaś `dostęp` jest specyfikatorem dostępu (private, protected, public), który może być różny dla każdej klasy bazowej. Przy dziedziczeniu od kilku klas bazowych konstruktory są wykonywane w takiej kolejności, w jakiej podano klasy bazowe. Destruktory są wykonywane w kolejności odwrotnej. Jeżeli klasy bazowe mają konstruktory definiowane, które wymagają podania argumentów, to przekazywanie argumentów z klas pochodnych odbywa się za pomocą rozszerzonej postaci konstruktora:

```
Pochodna(argumenty): Bazowa1(argumenty),
                     Bazowa2(argumenty),
                     ...
                     BazowaN(argumenty)
{
 // ciało konstruktora klasy Pochodna
}
```

Gdy klasa pochodna dziedziczy hierarchię klas, to każda klasa pochodna w łańcuchu dziedziczenia musi przekazać wstecz, do swojej bezpośredniej klasy bazowej, tyle i takich argumentów, jakie są wymagane.
W przedstawionym niżej przykładzie przyjęto schemat dziedziczenia, pokazany na rysunku 7-1. Przyjęte na nim piktogramy klas (prostokąty z nazwą klasy) i relacji dziedziczenia (trójkąty skierowane wierzchołkami ku klasom bazowym i połączone odcinkami linii prostych z tymi klasami)

są zgodne z powszechnie obecnie stosowanymi oznaczeniami OMT (Object Modeling Technique) wprowadzonymi przez J. Rumbaugh i in. w pracy [10].



Rys. 7-1 Dziedziczenie mnogie od dwóch klas bazowych

Przykład 7.10.

```
#include <iostream.h>
class Bazowa1 {
protected:
  int x;
public:
  Bazowa1(int i): x(i) {} // Definicja konstruktora
  int podajx()  { return x; }
};
class Bazowa2 {
protected:
  int y;
public:
  Bazowa2(int j): y(j) {} // Definicja konstruktora
  int podajy()  { return y; }
};
class Pochodna: public Bazowa1, public Bazowa2 {
public:
  Pochodna (int, int, int); //Deklaracja konstruktora
  void przedstaw()
  {
  cout << podajx() << ' ' << podajy() << ' ';
  cout << z << endl;
  }
private:
  int z;
};
//Konstruktor klasy pochodnej:
  Pochodna::Pochodna(int a, int b, int c):
  Bazowa1(a), Bazowa2(b) { z = c; }

int main() {
    Pochodna obiekt(1,2,3);
    obiekt.przedstaw();
    //podajx(),podajy()–publiczne w klasie Pochodna
    cout << obiekt.podajx() << ' ' << obiekt.podajy()
         << endl;
```

```
    return 0;
}
```

Wydruk z programu będzie miał postać:

1 2 3
1 2

Dyskusja. Instrukcja deklaracji `Pochodna obiekt(1,2,3);` woła konstruktor `Pochodna::Pochodna(a,b,c)`. Parametry `a==1` oraz `b==2` zostaną przekazane w odpowiedniej kolejności do konstruktorów klas `Bazowa1` i `Bazowa2`. Po utworzeniu obiektów tych klas zostanie dokończone wykonanie konstruktora `Pochodna(1,2,3)`, tj. zostanie utworzony i zainicjowany obiekt klasy `Pochodna` z pod-obiektami dwóch klas bazowych. Zwróćmy uwagę na fakt, że ze względu na publiczne dziedziczenie klas `Bazowa1` i `Bazowa2`, funkcje publiczne `podajx()` oraz `podajy()` pozostają publicznymi w klasie `Pochodna`. Wobec tego wartości `x` oraz `y` można odczytać zarówno za pomocą funkcji składowej `przedstaw()`, jak i funkcji podajx() i podajy() obiektu klasy Pochodna.

W dziedziczeniu mnogim, podobnie jak w pojedynczym, może wystąpić przypadek identycznych nazw zmiennych lub funkcji w dziedziczonych klasach. Podany niżej przykład ilustruje taki właśnie przypadek.

Przykład 7.11.

```
//Identyczne nazwy zmiennych skladowych int n
#include <iostream.h>
class Bazowa1 {
public:
  int n;
  Bazowa1(int i): n(i) {}
};

class Bazowa2 {
public:
  int n;
  Bazowa2 (int j): n(j) {}
};
class Pochodna: public Bazowa1, public Bazowa2 {
public:
  int k;
  Pochodna (int a, int b, int c);
};
 Pochodna::Pochodna(int a, int b, int c)
 : Bazowa1(a), Bazowa2(b)  { k = c; }

int main() {
    Pochodna obiekt(1,2,3);
//  cout << obiekt.n << endl; Niejednoznaczne
    cout << obiekt.Bazowa1::n << ' ';
    cout << obiekt.Bazowa2::n << endl;
    return 0;
}
```

Wydruk z programu będzie miał postać:

1 2

Dyskusja. W klasach `Bazowa1` i `Bazowa2` występuje taka sama nazwa zmiennej składowej `int n`. Bezpośrednie odwołanie do `n` w obiekcie klasy `Pochodna` byłoby niejednoznaczne. Odczytanie wartości `n` jest możliwe dopiero po jego skojarzeniu z odpowiednią klasą za pomocą operatora zasięgu.

Przy dziedziczeniu mnogim klasa bazowa nie może wystąpić więcej niż jeden raz w wykazie klas bazowych klasy pochodnej. Np. w ciągu deklaracji

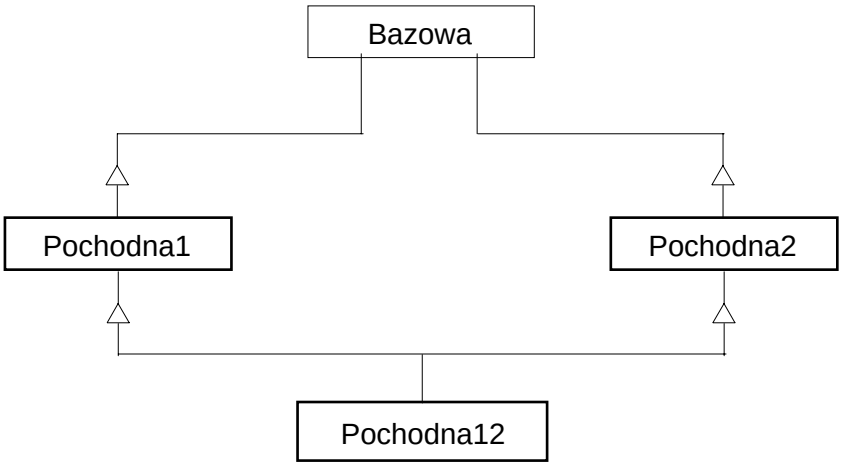
```
class Bazowa { //... };
class Pochodna: Bazowa, Bazowa { /... };
```

druga z nich jest błędna.
Natomiast klasa bazowa może być przekazana *pośrednio* do klasy pochodnej więcej niż jeden raz. Ilustruje to pokazany niżej przykład.

Przykład 7.12.

```
#include <iostream.h>
class Bazowa {
public:
  int a;
  Bazowa(int i): a(i) {} // Definicja konstruktora
};
class Pochodna1: public Bazowa {
public:
  int b;
  Pochodna1 (int j, int k):Bazowa(j) { b = k; }
};
class Pochodna2: public Bazowa {
public:
  int c;
  Pochodna2 (int m, int n):Bazowa(m) { c = n; }
};
class Pochodna12: public Pochodna1, public Pochodna2 {
public:
 int d;
 Pochodna12(int, int, int, int, int);
};

// Konstruktor klasy Pochodna12:
 Pochodna12::Pochodna12(int u,int w,int x,int y,int z)
 :Pochodna1(u,w), Pochodna2(x,y) { d = z; }

int main()
{
 Pochodna12 obiekt(1,2,3,4,5);
// cout << obiekt.a << endl; Niejednoznaczne
 cout << obiekt.Pochodna1::a << ' ';
 cout << obiekt.Pochodna1::b << ' ';
 cout << obiekt.Pochodna2::a << endl;
 return 0;
}
```

Dyskusja. Przyjęty w programie schemat dziedziczenia pokazano na rysunku 7-2.



Rys. 7-2 Przykład niejednoznaczności w hierarchii dziedziczenia

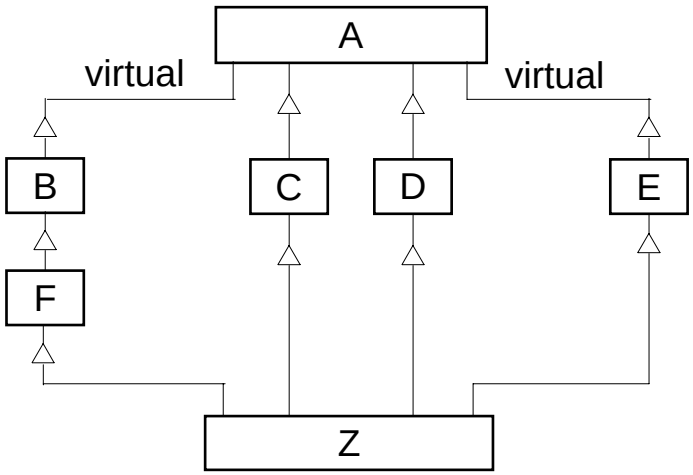
Jak widać z rysunku, każdy obiekt klasy `Pochodna12` będzie miał dwa pod-obiekty klasy `Bazowa`: jeden, dziedziczony za pośrednictwem klasy `Pochodna1` i drugi, dziedziczony za pośrednictwem klasy `Pochodna2`. Taka konstrukcja bywa nazywana *kratą klas*, lub skierowanym grafem acyklicznym (ang. DAG – akronim od Directed Acyclic Graph), w którym krawędzie grafu są skierowane od klas pochodnych do klas bezpośrednio rodzicielskich. W kracie dziedziczenia klasy `Pochodna1` i `Pochodna2` są nazywane "klasami siostrzanymi" (ang. sibling classes), ponieważ komunikują się przez wspólną klasę klasę bazową, a dziedziczone przez nie obiekty klasy `Bazowa` różnią się jedynie wartościami zmiennej `a`. W ogólności klasy siostrzane są to takie klasy, które mogą się komunikować za pośrednictwem wspólnej klasy bazowej, poprzez zmienne globalne, lub poprzez jawne wskaźniki.

8. Klasy i funkcje wirtualne

Dziedziczenie mnogie może być środkiem dla organizacji bibliotek wokół prostszych klas z mniejszą liczbą zależności pomiędzy klasami, niż w przypadku dziedziczenia pojedynczego. Gdyby ograniczyć dziedziczenie do pojedynczego, to każda biblioteka byłaby jednym drzewem dziedziczenia, w ogólności bardzo wysokim i rozgałęzionym. Mechanizm dziedziczenia mnogiego pozwala budować biblioteki w postaci „lasu mieszanego", w którym drzewa i grafy dziedziczenia mogą mieć zmienną liczbę poziomów i rozgałęzień. Z przeprowadzonej w r. 7 dyskusji wynika, że takie struktury można tworzyć stosunkowo łatwo, gdy klasa pochodna dziedziczy własności kilku niezależnych (rozłącznych) klas bazowych, nie mających wspólnej superklasy. Jeżeli jednak bezpośrednie klasy bazowe danej klasy pochodnej są zależne, należy zastosować omówione niżej mechanizmy językowe.

8.1. Wirtualne klasy bazowe

Przedstawiony w p.7.3 przykład ilustruje niejednoznaczności, jakie mogą się pojawić w hierarhii dziedziczenia, gdy klasa pochodna dziedziczy tę samą klasę bazową kilkakrotnie, idąc po różnych krawędziach grafu dziedziczenia. Odwołania do elementów składowych takiej klasy bazowej są wówczas możliwe, ale kłopotliwe (np. obiekt.Pochodna1::a). Język C++ oferuje tutaj mechanizm, dzięki któremu "klasy siostrzane" współdzielą informację (w tym przypadku jeden obiekt wspólnej klasy bazowej) bez wpływu na inne klasy w grafie dziedziczenia. Mechanizm ten polega na potraktowaniu wspólnej klasy bazowej jako klasy wirtualnej w klasach "siostrzanych", a przywołuje się go, pisząc słowo kluczowe **virtual** przed lub po specyfikatorze dostępu, a przed nazwą klasy bazowej. Wirtualność wspólnej klasy bazowej jest własnością stosowanego schematu dziedziczenia, a nie samej klasy, która poza tym niczym się nie różni od klasy niewirtualnej. Jeżeli przy dziedziczeniu mnogim klasa pochodna dziedziczy tę samą klasę bazową jako wirtualną i – idąc po innej gałęzi – jako niewirtualną, to oczywiście niejednoznaczności nie usuniemy. Ilustracją tego jest rysunek 8-1, który pokazuje schemat dziedziczenia z wirtualnymi i niewirtualnymi klasami bazowymi.



Rys. 8-1 Dziedziczenie mnogie z wirtualnymi klasami bazowymi

W prezentowanym grafie dziedziczenia leżąca najniżej w hierarchii klasa pochodna `Z` dziedziczy cechy sześciu swoich klas bazowych, przy czym klasy F, C, D i E są jej bezpośrednimi klasami bazowymi, zaś A i B – pośrednimi. Klasy B i E współdzielą jeden obiekt klasy A, ponieważ klasa A

jest w każdej z nich deklarowana jako wirtualna klasa bazowa. Natomiast każdy obiekt klas C i D będzie zawierać własną kopię zmiennych składowych klasy A. W rezultacie każdy obiekt klasy Z będzie zawierać trzy kopie zmiennych składowych klasy A: jedną przez dwie gałęzie wirtualne (przez E i B/F) i po jednej z gałęzi C i D.
Pokazany schemat można opisać przykładowymi deklaracjami:

```
class A {
public:
 void f() { cout << "A::f()\n"; }
};
class B: virtual public A { };
class c: public A { };
class D: public A { };
class E: virtual public A { };
class F: public B { };
class Z: public F, public C, public D, public E { };
```

Gdyby zadeklarować obiekt klasy `Z`:

```
Z obiekt;
```

to każde bezpośrednie wywołanie funkcji `f()` z tego obiektu

```
Z.f();
```

będzie niejednoznaczne, a więc błędne.

Wywołania funkcji `f()` można uczynić jednoznacznymi, odwołując się do niej poprzez obiekty klas pośrednich, które zawierają dokładnie po jednej kopii obiektu klasy `A`:

```
obiekt.C::f();
obiekt.D::f();
obiekt.E::f();
obiekt.F::f();
```

Niejednoznaczne będzie również wywołanie za pomocą wskaźnika do klasy `Z`:

```
Z* wsk = new Z;
wsk->f();
```

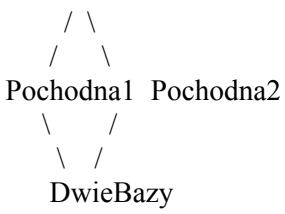
chociaż i w tym przypadku możemy wołać funkcję `f()` poprzez adresy obiektów klas pośrednich:

```
wsk->C::f();
wsk->D::f();
wsk->E::f();
wsk->F::f();
```

Wszystkie powyższe wywołania pośrednie mają składnię raczej mało zachęcającą. Gdyby w klasie `A` zadeklarować zmienne składowe, to odwołania do nich byłyby podobne.
Oczywistym sposobem usunięcia niejednoznaczności z dyskutowanego schematu byłoby zadeklarowanie klasy `A` jako wirtualnej klasy bazowej w pozostałych klasach pośrednich, tj. C i D. Takie właśnie założenie przyjęto w prezentowanym niżej programie, który korzysta ze znacznie prostszego schematu dziedziczenia.

Przykład 8.1.

Schemat dziedziczenia: Bazowa




```
#include <iostream.h>
class Bazowa {
public:
  Bazowa(): a(0) {}
  int a;
};
class Pochodna1: virtual public Bazowa {
public:
 Pochodna1(): b(0) {}
 int b;
};
class Pochodna2: virtual public Bazowa {
public:
 Pochodna2(): c(0) {}
 int c;
};
class DwieBazy: public Pochodna1, public Pochodna2 {
public:
  DwieBazy() {}
  int iloczyn() { return a*b*c; }
};
int   main() {
      DwieBazy obiekt;
      obiekt.a = 4; obiekt.b = 5; obiekt.c = 6;
      cout << "Iloczyn wynosi: "
           << obiekt.iloczyn() << endl;
      return 0;
}
```

Dyskusja. Instrukcja deklaracji `DwieBazy obiekt;` wywołuje konstruktor domyślny `DwieBazy() {}`. Konstruktor ten najpierw wywołuje konstruktor `Bazowa(){ a = 0; }`, a następnie konstruktory domyślne `Pochodna1()` i `Pochodna2()`. W rezultacie obiekt klasy `DwieBazy` będzie zawierał po jednym pod-obiekcie klas `Bazowa`, `Pochodna1` i `Pochodna2`.
Pozostała część programu nie wymaga obszerniejszego komentarza. Zauważmy jedynie, że w definicji funkcji `iloczyn()` wyrażenie `a*b*c` jest równoważne:
`Bazowa::a*Pochodna1::b*Pochodna2::c`.
Również poprawny byłby zapis
`obiekt.Bazowa::a`, ale dłuższy od `obiekt.a`.

•

Jeżeli wirtualna klasa bazowa zawiera konstruktory, to jeden z nich musi być konstruktorem domyślnym, albo konstruktorem z inicjalnymi wartościami domyślnymi dla wszystkich argumentów. Konstruktor domyślny będzie wołany bez argumentów, jeżeli żaden konstruktor klasy bazowej nie jest wywoływany jawnie z listy inicjującej konstruktora klasy pochodnej. Ponadto dla wirtualnej klasy bazowej obowiązują następujące reguły:

- Konstruktor wirtualnej klasy bazowej musi być wywoływany z tej klasy pochodnej, która faktycznie tworzy obiekt; wywołania z pośrednich klas bazowych będą ignorowane.
- Jeżeli deklaruje się wskaźnik do obiektów wirtualnej klasy bazowej, to nie można go przekształcić we wskaźnik do obiektów klasy pochodnej, ponieważ w klasie bazowej nie ma informacji o obiektach klas pochodnych. Natomiast konwersja wskaźnika w kierunku odwrotnym, tj. z klasy

pochodnej do klasy bazowej jest dopuszczalna, gdyż każdy obiekt klasy pochodnej zawiera wskaźnik do wirtualnej klasy bazowej. Ta własność wskaźników jest bardzo ważna, ponieważ odgrywa ona kluczową rolę przy definiowaniu i wykorzystaniu funkcji polimorficznych, nazywanych w języku C++ funkcjami wirtualnymi.

•

Podany niżej przykład ilustruje wymienione cechy wirtualnych klas bazowych. Klasa `Bazowa` jest teraz wyposażona w konstruktor z domyślną wartością argumentu, zaś wszystkie klasy pochodne mają konstruktory domyślne. Konstrukcja obiektu klasy `DwieBazy` zaczyna się od wywołania konstruktora `DwieBazy():Bazowa(300){}`, który najpierw wywołuje konstruktor klasy Bazowa, a następnie konstruktory klas `Pochodna1` i `Pochodna2`. Żaden z tych konstruktorów nie wywołuje konstruktora klasy `Bazowa`, ponieważ podobiekt tej klasy został już utworzony po wywołaniu konstruktora klasy `Bazowa` z bloku `DwieBazy()`. Sprawdzeniu tego faktu służy instrukcja `cout << obiekt.a << endl;`, która wydrukuje wartość 300. Deklaracja wskaźnika `wskb` służy do ilustracji konwersji z `Pochodna1*` do `Bazowa*`, zaś potraktowana jako komentarz instrukcja `wskp = (Pochodna1*)wskb;` ilustruje brak konwersji z typu `Bazowa*` do `Pochodna1*`. Wynika to bezpośrednio z arytmetyki wskaźników: wartością wyrażenia `wskb++` byłby adres następnego obiektu klasy `Bazowa`, zaś `wskp++` powinno się odnosić do następnego obiektu klasy `Pochodna`.
Zauważmy, że gdyby dodać deklaracje:

```
Pochodna1 obiekt1;
Pochodna2 obiekt2;
```

to każdy z tych obiektów zawierałby podobiekt klasy `Bazowa` z `a==100` i `a==200`, ponieważ za każdym razem z bloku konstruktora klasy pochodnej byłby wywołany konstruktor klasy `Bazowa`.

Przykład 8.2.

```
#include <iostream.h>
class Bazowa {
public:
  Bazowa(int i = 0): a(i) {}
  int a;
};

class Pochodna1: virtual public Bazowa {
public:
  Pochodna1(): Bazowa(100) {}
  int b;
};

class Pochodna2: virtual public Bazowa {
public:
  Pochodna2(): Bazowa(200) {}
  int c;
};
class DwieBazy: public Pochodna1, public Pochodna2 {
public:
  DwieBazy(): Bazowa(300) {}
};

int  main() {
     DwieBazy obiekt;
     obiekt.b = 5; obiekt.c = 6;
     cout << obiekt.a << endl;
     Bazowa* wskb;// wskb jest typu Bazowa*
     Pochodna1* wskp;//wskp jest typu Pochodna*
     wskb = (Bazowa*)wskp;
```

```
//Brak konwersji z Bazowa* do Pochodna1*
//    wskp = (Pochodna1*)wskb;
      return 0;
}
```

8.2. Funkcje wirtualne

Omawiając przeciążanie funkcji oraz operatorów stwierdziliśmy, że mechanizm ten realizuje polimorfizm, rozumiany jako "jeden interfejs (operacja), wiele metod (funkcji)". Jest to polimorfizm z wiązaniem wczesnym (nazywanym także statycznym), ponieważ rozpoznanie właściwego wywołania i ustalenie adresu funkcji przeciążonej następuje w fazie kompilacji programu. Dzięki temu wywołania funkcji z wiązaniem wczesnym należą do najszybszych. Wiązanie wczesne zachodzi również dla "zwykłych" funkcji oraz nie-wirtualnych funkcji składowych klasy i klas zaprzyjaźnionych.
W języku C++ istnieje ponadto bardziej finezyjny i giętki mechanizm, znany pod nazwą funkcji wirtualnych. Mechanizm ten odnosi się do tzw. wiązania późnego, czyli sytuacji, gdy adres wywoływanej funkcji nie jest znany w fazie kompilacji, lecz jest ustalany dopiero w fazie wykonania.
Wiązanie wywołania z definicją funkcji wirtualnej nie jest możliwe w fazie kompilacji, ponieważ funkcje wirtualne mają dokładnie takie same prototypy w całej hierarchii klas, a różnią się jedynie ciałem funkcji.
W schemacie dziedziczenia wiązane statycznie zwykłe funkcje składowe klasy również mogą mieć takie same prototypy w klasach bazowych i pochodnych, a różnić się tylko zawartością swoich bloków. W takich przypadkach funkcja klasy pochodnej nie zastępuje funkcji klasy bazowej, lecz ją ukrywa. Sytuacja ta jest podobna do ukrywania nazw zmiennych w blokach zagnieżdżonych. Ponieważ funkcje wirtualne nie spełniają kryteriów wymaganych dla funkcji przeciążonych, nie mogą być rozróżnione w omawianym w rozdziale 5 procesie rozpoznawania i dopasowania. Tym niemniej kompilator pozwala je rozróżnić dzięki omawianej dalej regule dominacji; ponadto programista może użyć w tym celu operatora zasięgu "::" dla klasy. Technika ta sprawdza się w przypadku dziedziczenia pojedynczego. Jednak dla dziedziczenia mnogiego, jak pokazano na początku tego rozdziału (nawet dla wirtualnych klas bazowych), kontrola wywołań staje się kłopotliwa, a dla złożonych grafów dziedziczenia zawodzi.

Mechanizm funkcji wirtualnych można więę określić jako polimorfizm z wiązaniem późnym, a programowanie, oparte o hierarchię klas i funkcje wirtualne jest często utożsamiane z obiektowym stylem programowania.

Dogodnym narzędziem dla operowania funkcjami wirtualnymi są wskaźniki i referencje do obiektów. Im też poświęcimy obecnie więcej uwagi.

8.2.1. Wskaźniki i referencje w hierarchii klas

Wskaźniki i referencje używaliśmy wielokrotnie i w różnych kontekstach. Nie zwracaliśmy natomiast uwagi na szczególne własności wskaźników w hierarchii klas. Tymczasem dla efektywnego posługiwania się funkcjami wirtualnymi potrzebujemy takiego sposobu odwoływania się do obiektów różnych klas, który nie wymaga faktycznej zmiany obiektu, do którego się odwołujemy. Sposób taki istnieje i opiera się na następującej własności wskaźników i referencji: zmienną wskaźnikową (referencyjną) zadeklarowaną jako wskaźnik do klasy bazowej można użyć dla wskazania na dowolną klasę pochodną od tej klasy bazowej bez używania jawnej konwersji typu. Jeżeli więc weźmiemy deklaracje:

```
class Bazowa { /* ... */ };
class Pochodna: public Bazowa { /* ... */ };
Bazowa baz; Pochodna po;
```

to możemy zapisać następujące poprawne instrukcje:

```
Bazowa* wskb = &baz;
wskb = &po;
Bazowa& refb = po;
```

```
Przy tych samych deklaracjach poprawne będą również instrukcje:
Bazowa* wskb = new Bazowa;
wskb = &po;
```

Przykład 8.3.

```
#include <iostream.h>
class Bazowa {
public:
   void ustawx(int n) { x = n; }
   void podajx() { cout << x << '\t'; }
private:
   int x;
};

class Pochodna : public Bazowa {
public:
   void ustawy(int m) { y = m; }
   void podajy() { cout << y << '\t'; }
 private:
   int y;
};
int  main() {
      Bazowa *wsk; //wskaz do klasy Bazowa
      Bazowa obiekt1; // Obiekt klasy Bazowa
      Pochodna obiekt2; //Obiekt klasy Pochodna
      wsk = &obiekt1; //Przypisz do wsk adres obiekt1
      wsk->ustawx(10); //Ustaw x w obiekt1
      obiekt1.podajx(); //Alternatywa: wsk->podajx()
      wsk = &obiekt2; //Przypisz do wsk adres obiekt2
      wsk->ustawx(20); //Ustaw x w podobiekcie obiekt2
      wsk->podajx();
//    wsk->ustawy(30); Nielegalna
      obiekt2.ustawy(30);
//    wsk->podajy(); Nielegalna
      obiekt2.podajy();
      return 0;
}
```

Wydruk z programu będzie miał postać:

10 20 30

Komentarz. Instrukcje `wsk->ustawy(30);` oraz `wsk->podajy();` byłyby nielegalne, mimo że wskaźnik `wsk` jest ustawiony na adres obiektu klasy pochodnej. Jest to oczywiste, ponieważ `wsk` pozwala na dostęp tylko do tych składowych obiektu klasy pochodnej, które są dziedziczone z klasy bazowej.

Zanotujmy w tym miejscu kilka uwag.

- ☞*Wskaźniki i referencje odwołują się do obiektów poprzez ich adresy, które mają ten sam rozmiar bez względu na klasę, do której należy wskazywany obiekt.*
- ☞*Z arytmetyki wskaźników wynika, że zwiększenie o 1 wartości wskaźnika brane jest w odniesieniu do zadeklarowanego typu danych. Tak więc, jeżeli wskaźnik* `wskb` *typu* `Bazowa*` *wskazuje na obiekt klasy* `Pochodna`*, to wartość* `wskb++` *będzie się odnosić do następnego obiektu klasy* `Bazowa`*, a nie klasy* `Pochodna`*.*

☞*Mimo, że możemy używać wskaźnika* `wskb` *do wskazywania obiektu klasy pochodnej, to jednak w tym przypadku uzyskamy dostęp jedynie do tych elementów (zmiennych i funkcji) klasy pochodnej, które odziedziczyła od klasy bazowej (zarówno bezpośredniej, jak i pośredniej). Powodem jest to, że wskaźnik do klasy bazowej dysponuje jedynie informacją o tej klasie, a nie "wie" nic o elementach, dodanych przez klasę pochodną.*

Wprawdzie jest dopuszczalne, aby wskaźnik do klasy bazowej wskazywał obiekt klasy pochodnej, to jednak twierdzenie odwrotne nie jest prawdziwe. Jest to spowodowane faktem, że kompilatory nie mają mechanizmu sprawdzania – dla fazy wykonania – czy konwersje w instrukcjach przypisania: `wskb = wskp;` gdzie `wskp` jest wskaźnikiem do klasy pochodnej, pozostawiają wynik, wskazujący na obiekt oczekiwanego typu.

8.2.2. Deklaracje funkcji wirtualnych

Funkcja wirtualna jest to funkcja składowa klasy, zadeklarowana w klasie bazowej i redefiniowana w klasach pochodnych. Deklaracja funkcji wirtualnej odróżnia się od deklaracji zwykłej funkcji składowej jedynie tym, że przed nazwą typu zwracanego umieszcza się słowo kluczowe **virtual**, np.

```
virtual void f();
virtual char* g() const;
```

Nazwy funkcji wirtualnych, wywoływanych za pośrednictwem wskaźników lub referencji do obiektów, wiązane są z ich adresami w fazie wykonania. Jest to omawiane wcześniej wiązanie późne (dynamiczne). Natomiast funkcje zadeklarowane jako wirtualne, a wywoływane dla obiektów, są wiązane w fazie kompilacji (wiązanie wczesne, albo statyczne). Przyczyną tego jest fakt, że typ obiektu jest już znany po kompilacji. Np. dla deklaracji:

```
class Test {
public:
 virtual void ff();
};
Test t1;
```

wywołanie:

```
t1.ff();
```

będzie wiązane statycznie, a więę funkcja `ff()` dostanie adres w fazie kompilacji i straci cechę wirtualności.

Prezentowany niżej program wydrukuje wartoćć x: 10. Zwróćmy uwagę na definicję funkcji `podaj()`: słowo kluczowe **virtual** występuje tylko w jej deklaracji; błędem syntaktycznym byłoby umieszczenie go w definicji funkcji, umieszczonej poza ciałem klasy.

Przykład 8.4.

```
#include <iostream.h>
class Bazowa {
public:
  int x;
  Bazowa(int i): x(i) {}
  virtual void podaj();
};
void Bazowa::podaj()
{ cout << "x: " << x << endl; }
int  main() {
     Bazowa *wsk;
     Bazowa obiekt(10);
```

```
      wsk = &obiekt;
      wsk->podaj();
      return 0;
}
```

Przykład 8.5.

```
#include <iostream.h>
class Bazowa {
public:
  int x;
  Bazowa(int i): x(i)  {}
  virtual void podaj();
};
void Bazowa::podaj()
{ cout << "x = "<< x << endl; }
class Pochodna1 : public Bazowa {
public:
  Pochodna1(int x): Bazowa(x) {}
  void podaj();
};
void Pochodna1::podaj()
{ cout << "x + x = "<< x + x << endl; }
class Pochodna2 : public Bazowa {
public:
  Pochodna2(int x): Bazowa(x) {}
};

int  main() {
      Bazowa* wsk;
      Bazowa obiekt(10);
      Pochodna1  obiekt1(20);
      Pochodna2  obiekt2(30);
      wsk = &obiekt;
      wsk->podaj();
      wsk = &obiekt1;
      wsk->podaj();
      wsk = &obiekt2;
      wsk->podaj();
      return 0;
}
```

Dyskusja. Wirtualna funkcja składowa `podaj()` jest redefiniowana jedynie w klasie pochodnej `Pochodna1`; w klasie `Pochodna2` musi być używana definicja z klasy `Bazowa`. W rezultacie wydruk z programu będzie miał postać:

x = 10
x + x = 40
x = 30

W programie wszystkie wiązania funkcji wirtualnej podaj() były wiązaniami dynamicznymi, realizowanymi w fazie wykonania programu. Gdyby wywołania funkcji podaj() związać z tworzonymi obiektami, a nie ze wskaźnikami do tych obiektów, np. obiekt1.podaj(), to wiązania miałyby miejsce w fazie kompilacji, a więę byłyby wiązaniami wczesnymi (statycznymi).

•

Dokonamy teraz przeglądu podstawowych własności funkcji wirtualnych.

•

- Funkcje wirtualne mogą występować jedynie jako funkcje składowe klasy. Zatem, podobnie jak funkcje składowe rozwijalne i statyczne, nie mogą być deklarowane ze specyfikatorem **extern**.
- Specyfikatora **virtual** nie wolno używać w deklaracjach statycznych funkcji składowych. Nie wolno, ponieważ wywołanie funkcji wirtualnej odnosi się do konkretnego obiektu, który używa zredefiniowanej treści funkcji. Inaczej mówiąc, funkcja wirtualna wykorzystuje niejawny wskaźnik this, którego nie ma funkcja statyczna.
- Funkcja wirtualna może być zadeklarowana jako zaprzyjaźniona (friend) w innej klasie.
- Funkcja wirtualna, zdefiniowana w klasie bazowej, nie musi być redefiniowana w klasie pochodnej. W takim przypadku wywołanie z obiektu klasy pochodnej będzie korzystać z definicji, zamieszczonej w klasie bazowej.
- Jeżeli definicja funkcji wirtualnej z klasy bazowej zostanie przesłonięta nową definicją w klasie pochodnej, to funkcja przesłaniająca jest również uważana za wirtualną.
- Specyfikator **virtual** może być użyty dla funkcji przesłaniającej w klasie pochodnej, ale będzie to redundancja. Jednak taka rozwlekłość może być korzystna dla celów dokumentacyjnych. Jeżeli w trakcie czytania tekstu programu chcemy sprawdzić, czy dana definicja odnosi się do funkcji wirtualnej, czy też nie, specyfikator **virtual** zaoszczędzi nam czas na poszukiwanie.
- Język C++ wymaga dokładnej zgodności deklaracji pomiędzy funkcją wirtualną w klasie bazowej a funkcją, która ją przesłania w klasie pochodnej. Inaczej mówiąc, funkcje te muszą mieć identyczne sygnatury, tj. liczbę, kolejność i typy argumentów oraz ten sam typ zwracany. Ewentualne naruszenie tej zgodności spowoduje, że kompilator nie będzie traktować funkcji zredefiniowanej jako wirtualnej, lecz jako funkcję przeciążoną, bądź nową funkcję składową.
- Ostatnio komitet normalizacyjny ANSI X3J16 rozluźnić powyższe ograniczenie. Teraz dopuszcza się niezgodność typów funkcji w następującym przypadku: jeżeli oba typy zwracane są wskaźnikami lub referencjami do klas i jeżeli klasa w typie zwracanym przez funkcję wirtualną klasy pochodnej jest publicznie dziedziczona od klasy w typie zwracanym przez funkcję wirtualną klasy bazowej. Dla tej nowej wykładni podane niżej deklaracje (błędne przy poprzedniej) będą poprawne:

```
class X { };
class Y : public X { };
class A {
public:
  virtual X& fvirt();
};
class B : public A {
public:
  virtual Y& fvirt();
};
```

Podsumujmy powyższe uwagi. Składnia wywołania funkcji wirtualnej jest taka sama, jak składnia wywołania zwykłej funkcji składowej. Interpretacja wywołania funkcji wirtualnej zależy od typu obiektu, dla którego jest wołana; jeżeli np. mamy klasę `Test` z zadeklarowaną w niej funkcją wirtualną

```
virtual void ff();
```

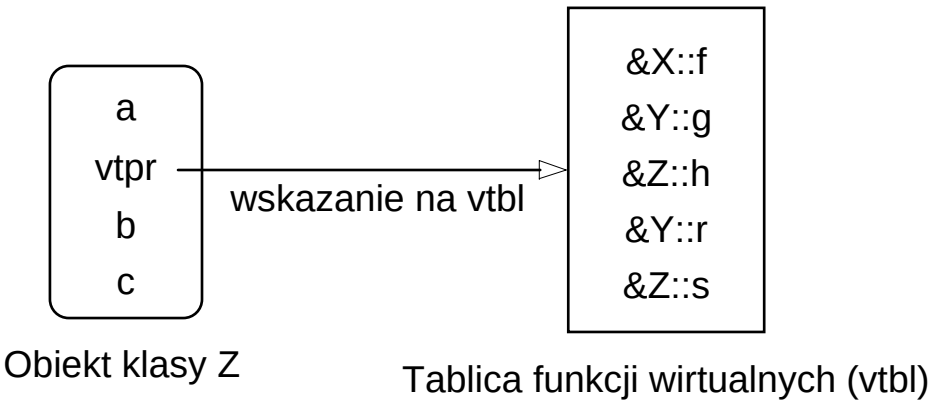
to ciąg instrukcji

```
Test* wsk = new Test;
Test t1;
Test& tref = t1;
wsk->ff();
tref.ff();
```

mówi: “hej, adresowany obiekcie, wybierz swoją własną funkcję `ff()` i wykonaj ją.”
Inaczej mówiąc: ponieważ obiekty są powoływane do życia w fazie wykonania, zatem wywołanie funkcji wirtualnej musi być wiązane z jedną z jej definicji (metod) dopiero w fazie wykonania.
Można w tym miejscu postawić pytanie: w jaki sposób informacja o typie obiektu dla wywołania funkcji wirtualnej jest przekazywana przez kompilator do środowiska wykonawczego? Odpowiedź na to pytanie nie może abstrahować od implementacji języka C++.
Przyjętym w języku C++ rozwiązaniem jest implementacja funkcji wirtualnych za pomocą tablicy wskaźników do funkcji wirtualnych. Np. przy dziedziczeniu pojedynczym każda klasa, w której zadeklarowano funkcje wirtualne, utrzymuje tablicę wskaźników do funkcji wirtualnych, a każdy obiekt takiej klasy będzie zawierać wskaźnik do tej tablicy. Weźmy dla ilustracji następujący schemat dziedziczenia:

```
class X {
public:
 virtual void f();
 virtual void g(int);
 virtual void h(char*);
private:
 int a;
};
class Y {
public:
 void g(int);
 virtual void r(Y*);
private:
 int b:
};
class Z {
public:
 void h(char*);
 virtual void s(Z*)
private:
 int c;
};
```

Przy powyższych deklaracjach struktura obiektu klasy `Z` będzie podobna do pokazanej na rysunku 8-2.



Rys. 8-2 Wskaźnik do tablicy funkcji wirtualnych

Każdy obiekt klasy `Z` będzie zawierać ukryty wskaźnik do tablicy funkcji wirtualnych, nazywany w implementacjach `vptr`. Wywołanie funkcji wirtualnej jest transformowane przez kompilator w wywołanie pośrednie. Np. wywołanie funkcji `g()` z bloku funkcji `f`

```
void f(Z* wsk) { wsk->g(10); }
```

wygeneruje kod w rodzaju:

```
(*(wsk->vptr[1]))(wsk,10);
```

Zasada jest taka, że przy dziedziczeniu pojedynczym dla każdej klasy z funkcjami wirtualnymi utrzymywana jest dokładnie jedna tablica funkcji wirtualnych. Przy dziedziczeniu mnogim klasa z funkcjami wirtualnymi, pochodna np. od dwóch klas bazowych będzie miała dwie takie tablice; obiekt tej
klasy będzie miał odpowiednio dwa ukryte wskaźniki, po jednym do każdej z tablic. Powód jest oczywisty: każdy obiekt klasy pochodnej od kilku klas bazowych będzie zawierał podobiekty tych klas, a z każdym podobiektem będzie skojarzona jego własna tablica funkcji wirtualnych.

8.2.3. Zasięg i reguła dominacji

Jak pamiętamy, każda klasa wyznacza swój własny zasięg, a jej zmienne i funkcje składowe mieszczą się w zasięgu swojej klasy. Nazwy w zasięgu klasy mogą być dostępne dla kodu zewnętrznego w stosunku do klasy, jeżeli użyjemy do tego celu operatora zasięgu "::".
Zasięg odgrywa kluczową rolę w mechanizmie dziedziczenia. Z punktu widzenia klasy pochodnej dziedziczenie wprowadza nazwy z zasięgu klasy bazowej w zasięg klasy pochodnej. Inaczej mówiąc, nazwy zadeklarowane w klasach bazowych są dziedziczone przez klasy pochodne.
Nazwy w zasięgu klasy pochodnej są w podobnej relacji do nazw w klasie bazowej, jak nazwy w bloku wewnętrznym do nazw w bloku go otaczającym. W bloku wewnętrznym zawsze możemy użyć nazwy zadeklarowanej w bloku zewnętrznym. Jeżeli w bloku wewnętrznym zdefiniujemy taką samą nazwę, jak zdefiniowana w bloku zewnętrznym, to nazwa w bloku wewnętrznym ukryje nazwę z bloku zewnętrznego.
Przy dziedziczeniu mnogim tworzenie obiektu klasy pochodnej zaczyna się zawsze od utworzenia podobiektów wcześniej zadeklarowanych klas bazowych. W tym przypadku zasięg klasy pochodnej jest zagnieżdżony w zasięgach wszystkich jej klas bazowych. Jeżeli dwie klasy bazowe zawierają tę samą nazwę, wtedy albo jedna nazwa musi dominować nad drugą, albo klasa pochodna musi usunąć niejednoznaczność przez ukrycie deklaracji z klas bazowych. Dopuszczalnoćć przesłaniania nazw z różnych gałęzi drzewa lub grafu dziedziczenia wymaga sformułowania zasady określającej, jakie kombinacje mogą być akceptowane, a jakie należy odrzucić jako błędne.
Zasada taka, nazywana regułą dominacji, została sformułowana przez B.Stroustrupa i A.Koeniga; brzmi ona następująco:

"Nazwa `B::f` dominuje nad nazwą `A::f`, jeżeli klasa `B`, w której `f` jest składową, jest klasą pochodną od klasy `A`. Jeżeli pewna nazwa dominuje nad inną, to nie ma między nimi kolizji. Nazwa dominująca zostanie użyta wtedy, gdy istnieje wybór."

Zauważmy, że reguła dominacji stosuje się zarówno do funkcji, jak i zmiennych składowych. Prezentowany niżej przykład ilustruje działanie tej zasady w dziedziczeniu pojedynczym.

Przykład 8.6.

```
// Dominacja w dziedziczeniu pojedynczym
// bez klas i funkcji wirtualnych
#include <iostream.h>
class Bazowa {
public:
 void f() { cout << "Bazowa::f()\n"; }
 void g() { cout << "Bazowa::g()\n" << endl; }
};
class Pochodna1: public Bazowa {
public:
```

```
 void f() { cout << "Pochodna1::f()\n"; }
};
class Pochodna2: public Bazowa {
public:
 void g() { cout << "Pochodna2::g()\n" << endl; }
};
int  main() {
     Pochodna1 po;
     po.f();
     po.g();
     return 0;
}
```

Z programu otrzymuje się wydruk o postaci:

Pochodna1::f()
Bazowa::g()

Dyskusja. W instrukcji `po.f();` wywoływana jest funkcja `f()` klasy `Pochodna1`, ponieważ nazwa `Pochodna1::f` dominuje nad nazwą `Bazowa::f`. Podobny efekt dałoby wywołanie funkcji `g()` dla obiektu klasy `Pochodna2`. Natomiast instrukcja `po.g();` wywołuje `Bazowa::g()`, ponieważ w klasie `Pochodna1` nazwa `g` nie występuje, ale klasa ta ma dostęp do funkcji składowej `g()` klasy Bazowa.

Przykład 8.7.

```
/* Dominacja w dziedziczeniu mnogim z klasami
wirtualnymi, lecz bez funkcji wirtualnych
*/
#include <iostream.h>

class Bazowa {
public:
   void f() { cout << "Bazowa::f()\n"; }
};

class Pochodna2: virtual public Bazowa {
public:
   void f() { cout << "Pochodna2::f()\n"; }
};

class Pochodna1:
virtual public Bazowa,
virtual public Pochodna2 { };

int  main() {
     Pochodna1 po1;
     po1.f();
     return 0;
}
```

Dyskusja. Wydruk z programu ma postać: **Pochodna2::f()**. Nazwa `f` nie występuje w deklaracji klasy `Pochodna1`, natomiast występuje w klasach `Bazowa` i `Pochodna2`, które są publicznymi wirtualnymi klasami bazowymi klasy `Pochodna1`. Z reguły dominacji wynika, że nazwa `f` w klasie `Pochodna2` dominuje nad tą samą nazwą w jej wirtualnej klasie bazowej.
Następny przykład ilustruje w pełni korzyści, wynikające z reguły dominacji. Schemat dziedziczenia jest tutaj grafem acyklicznym, w którym dwie klasy dziedziczą od wspólnej publicznej, wirtualnej

klasy bazowej. Klasy te są bezpośrednimi publicznymi klasami bazowymi dla najniższej w hierarchii klasy pochodnej. Przyjęty schemat dziedziczenia, wraz z deklaracjami funkcji wirtualnych we wspólnej klasie bazowej i wywołaniami tych funkcji przez wskaźniki zapewnia jednoznaczność odwołań.

Przykład 8.8.

```
//Dominacja: klasy i funkcje wirtualne
#include <iostream.h>
class Bazowa {
public:
   virtual void f() { cout << "Bazowa::f()\n"; }
   virtual void g() { cout << "Bazowa::g()\n"; }
};
class Pochodna1: virtual public Bazowa {
public:
   void g() { cout << "Pochodna1::g()\n"; }
};
class Pochodna2: virtual public Bazowa {
public:
   void f() { cout << "Pochodna2::f()\n"; }
};
class Pochodna12:
public Pochodna1, public Pochodna2 { };
int  main() {
     Pochodna12 obiekt;
     Pochodna12* wsk12 = &obiekt;
     wsk12->f();
     wsk12->g();
     Pochodna1* wsk1 = wsk12;
     wsk1->f();
     Pochodna2* wsk2 = wsk12;
     wsk2->g();
     return 0;
}
```

Wykonanie programu da wydruk o postaci:

Pochodna2::f
Pochodna1::g()
Pochodna2::f()
Pochodna1::g()

Analiza programu. Wywołanie `wsk12->f()` zostanie rozpoznane jako odnoszące się do nazwy `f` w klasie `Pochodna2`, co wynika z reguły dominacji i z faktu, że funkcja `f()` została zadeklarowana jako wirtualna. To samo odnosi się do pozostałych wywołań.

8.2.4. Wirtualne destruktory

Destruktor jest, podobnie jak konstruktor, specjalną funkcją składową klasy. Jak wiadomo, zadaniem konstruktora jest inicjowanie zmiennych składowych przy tworzeniu obiektu; destruktor wykonuje wszelkie czynności związane z usuwaniem obiektu, jak dealokacja uprzednio przydzielonej pamięci operatorem **new**, itp.
Jeżeli w klasie nie zadeklarowano destruktora, to będzie niejawnie wywoływany konstruktor domyślny generowany przez kompilator. Wywołanie to ma miejsce, gdy kończy się okres życia obiektu, np. gdy sterowanie opuszcza blok, w którym zadeklarowano obiekt lokalny lub – dla obiektów statycznych – gdy kończy się cały program.

Destruktor może być również zdefiniowany w klasie; wówczas będzie on wywoływany niejawnie zamiast destruktora generowanego przez kompilator. Taki destruktor można też wywoływać jawnie; np. dla deklaracji:

```
class Test {
public:
 Test() { };
 ~Test() { };
};
```

destruktor `~Test()` może być wywołany dla obiektu klasy `Test` z kwalifikatorem zawierającym nazwę klasy i operator zasięgu lub bez, zależnie od kontekstu:

```
Test t1;
t1.~Test();
t1.Test::~Test();
```

Konieczność definiowania własnych destruktorów zachodzi wtedy, gdy przy tworzeniu obiektu są alokowane jakieć oddzielnie zarządzane zasoby, np. gdy wewnątrz obiektu jest tworzony w pamięci swobodnej podobiekt. Wtedy zadaniem destruktora będzie zwolnienie tego obszaru pamięci.
Jawne wywołanie destruktora na rzecz jakiegoś obiektu powoduje jego wykonanie, ale niekoniecznie zwalnia pamięć przydzieloną zmiennym obiektu i nie zawsze oznacza całkowite zakończenie życia obiektu; co więcej, niewłaściwie zaprojektowany destruktor może próbować zwalniać już uprzednio zwolnione zasoby. Podany niżej przykład ilustruje taką własnie sytuację.

Przykład 8.9.

```
#include <iostream.h>
class Test {
public:
 enum { n = 10 };
 double* wsk;
 Test() { wsk = new double[n]; }
 ~Test()
  {
   if(wsk)
    {
     delete []  wsk;
     cout << "Zniszczenie wsk\n";
  // wsk = NULL;
    }
   else cout << "wsk==NULL\n";
  }
};
int   main() {
      Test t1;
      t1.~Test();
      return 0;
}
```

Dyskusja. Jeżeli instrukcja `wsk = NULL;` będzie traktowana jako komentarz, to wykonanie programu da wydruk:

Zniszczenie wsk
Zniszczenie wsk

W tym przypadku destruktor być wywoływany dwukrotnie: raz jawnie instrukcją `t1.~Test();` i drugi raz niejawnie przy ostatecznym usuwaniu obiektu `t1` tuż przed zakończeniem programu. Dopuszczenie do takiej sytuacji jest oczywistym błędem programistycznym.
Jeżeli z programu usuniemy symbol komentarza przed instrukcją `wsk = NULL;` to zostanie ona wykonana, a wydruk będzie miać postać:

```
Zniszczenie wsk
wsk==NULL
```

Teraz destrukcja obiektu `t1` przebiega poprawnie: przy pierwszym wywołaniu destruktor zwalnia pamięć zajmowaną przez tablicę dziesięciu liczb typu **double** i następnie przypisuje wskaźnikowi `wsk` adres pusty. Przy drugim wywołaniu drukuje `napis wsk==NULL` i usuwa to, co jeszcze pozostało z obiektu `t1` (w tę czynność nie wchodzimy, ponieważ zależy ona od implementacji, a więc mieści się na niższym poziomie abstrakcji w stosunku do klas i obiektów).

Problemy z destrukcją obiektów uzyskują dodatkowy wymiar, gdy uwzględnimy dziedziczenie. Ilustruje to następny przykład.

Przykład 8.10.

```
#include <iostream.h>
class Bazowa {
public:
 Bazowa() { cout << "Konstruktor klasy bazowej\n"; }
 ~Bazowa() { cout << "Destruktor klasy bazowej\n"; }
};
class Pochodna: public Bazowa {
public:
 Pochodna()
 { cout << "Konstruktor klasy pochodnej\n"; }
 ~Pochodna()
 { cout << "Destruktor klasy pochodnej\n"; }
};
int  main() {
     Bazowa* wskb = new Pochodna;
     delete wskb;
     return 0;
}
```

Analiza programu. W przykładzie posłużono się wskaźnikiem `wskb` do klasy bazowej, któremu przypisano obiekt klasy pochodnej. Wydruk z programu ma postać nieoczekiwaną:

Konstruktor klasy bazowej
Konstruktor klasy pochodnej
Destruktor klasy bazowej

Konstrukcja obiektu klasy pochodnej przebiega prawidłowo (najpierw jest tworzony obiekt klasy bazowej, a następnie pochodnej). Natomiast po wykonaniu instrukcji `delete wskb;` w sytuacji gdy wskaźnik był ustawiony na adres obiektu klasy pochodnej można się było spodziewać wywołania destruktora klasy
pochodnej, a nie bazowej. Rzecz w tym, że wywołanie delete “nic nie wie” o tym, iż obiekt jest klasy `Pochodna`, a w klasie `Bazowa` nie ma żadnej wskazówki, że wywołania destruktora mają przeszukiwać hierarchię klas.

•

Konsekwencje opisanej wyżej sytuacji zostały już zasygnalizowane wcześniej. Jeżeli konstruktor klasy pochodnej przydzielił obiektowi tej klasy pewne zasoby, które miał zwolnić jej destruktor, to działanie takie nie zostanie wykonane i zasoby te staną się tzw. "nieużytkami" (ang. garbage).
Rozwiązaniem tego problemu jest wprowadzenie destruktorów wirtualnych. Następny przykład ilustruje sposób definiowania destruktora wirtualnego w klasach bazowej i pochodnej oraz składnię jego wywołania.

Przykład 8.11.

```
// Destruktor wirtualny
#include <iostream.h>
class Bazowa {
public:
  Bazowa()
  { cout << "Konstruktor klasy bazowej\n"; }
  virtual ~Bazowa()
  { cout << "Destruktor klasy bazowej\n"; }
};
class Pochodna: public Bazowa {
public:
  Pochodna()
  { cout << "Konstruktor klasy pochodnej\n"; }
  ~Pochodna()
  { cout << "Destruktor klasy pochodnej\n"; }
};
int   main() {
      Bazowa* wskb = new Pochodna;
      delete wskb;
      return 0;
}
```

Dyskusja. Program wydrukuje następujące napisy:

Konstruktor klasy bazowej
Konstruktor klasy pochodnej
Destruktor klasy pochodnej
Destruktor klasy bazowej

Teraz konstrukcja i destrukcja obiektu przebiega poprawnie. Dzieje się to za sprawą destruktora klasy bazowej, zadeklarowanego ze słowem kluczowym **virtual**. Wirtualny destruktor klasy bazowej został zredefiniowany w klasie pochodnej; musieliśmy użyć w tym celu nazwy klasy pochodnej, czyli zrobić odstępstwo od zasad definiowania funkcji wirtualnych. Jest to (na szczęście) odstępstwo dopuszczalne, jeśli chcemy zachować zasady nazewnictwa konstruktorów i destruktorów. Funkcja wirtualna, jaką jest wirtualny destruktor, jest wołana dla wskaźnika do obiektu; zatem jej definicja będzie wiązana z wywołaniem dynamicznie, w fazie wykonania. Ponieważ wskaźnik `wskb` adresuje obiekt klasy `Pochodna`, zatem instrukcja `delete wskb;` wywoła destruktor tej klasy. Oczywiście natychmiast po tym zostanie wywołany destruktor klasy `Bazowa`, zgodnie z obowiązującą kolejnością wywołania destruktorów dla klas pochodnych.

•

W przykładach destrukcji obiektów w drzewie dziedziczenia pominęliśmy sprawę zwalniania dodatkowych zasobów, alokowanych dla obiektu przez konstruktor. Uczyniono tak w celu uproszczenia zapisu, aby pokazać inny aspekt problemu destrukcji. Oczywiście i w przypadku dziedziczenia kłopoty ze zwalnianiem zasobów, czy też usuwaniem zasobów już usuniętych, są podobne, a często zwielokrotnione wskutek zastosowania mechanizmu dziedziczenia. Innym pominiętym aspektem, o którym warto wspomnieć, jest problem rozpoznania obiektu, dla którego jest wywoływany destruktor klasy bazowej. Jeżeli usuwamy

obiekt klasy pochodnej, to najpierw jest wołany destruktor tej klasy, a następnie destruktor klasy bazowej.
Tak więc w chwili wywołania destruktora klasy bazowej obiekt klasy pochodnej już nie istnieje; istnieją tylko jego składowe, odziedziczone z klasy bazowej. W tej fazie destruktor nie może się dowiedzieć, czy należą one do obiektu klasy pochodnej, czy bazowej...

8.2.5. Klasy abstrakcyjne i funkcje czysto wirtualne

Funkcje wirtualne, definiowane w klasach bazowych "na szczycie" hierarchii klas, bardzo często są jedynie makietami, umieszczonymi z myślą o napisaniu dla nich sensownych definicji w klasach pochodnych. Jest to sytuacja typowa, ponieważ klasa bazowa często jest projektowana jako pewien prototyp dla klas pochodnych. Zwykłe funkcje składowe mogą mieć w takiej klasie puste bloki definicji. Jeżeli funkcja wirtualna w klasie bazowej nie wykonuje żadnego działania, to każda klasa pochodna musi przesłonić tę funkcję. Dla takiego przypadku przewidziano w języku C++ funkcje czysto wirtualne. W klasie bazowej wymagana jest jedynie deklaracja, wprowadzająca prototyp funkcji czysto wirtualnej (nie ma definicji). Ma ona postać:

```
virtual  typ  nazwa(wykaz parametrów) = 0;
```

Istotną częścią tej deklaracji jest związanie ciała (bloku) funkcji ze wskaźnikiem zerowym. Jest to informacja dla kompilatora, że nie istnieje ciało tej funkcji dla klasy bazowej. Po otrzymaniu takiej informacji kompilator będzie wymuszać redefinicje funkcji czysto wirtualnej w każdej klasie pochodnej, która ma mieć wystąpienia (obiekty), ponieważ funkcja czysto wirtualna jest dziedziczona jako czysto wirtualna.
Klasa bazowa, która zawiera conajmniej jedną funkcję czysto wirtualną, jest nazywana abstrakcyjną klasą bazową. Nazwa jest uzasadniona tym, że taka klasa nie może mieć swoich wystąpień, tj. obiektów; natomiast można jej używać dla tworzenia klas pochodnych. Klasa abstrakcyjna nie może mieć swoich wystąpień, ponieważ – w sensie technicznym – nie jest kompletnym typem ze względu na brak definicji funkcji czysto wirtualnej. Zauważmy także, że jeśli w klasie pochodnej od klasy abstrakcyjnej nie redefiniuje się odziedziczonej funkcji czysto wirtualnej, to taka klasa pochodna będzie również klasą abstrakcyjną, pozbawioną możliwości tworzenia własnych obiektów. Ponadto można wymienić następujące własności klas abstrakcyjnych:

- Dopuszcza się deklarowanie wskaźników i referencji do klas abstrakcyjnych.
- Klasa abstrakcyjna nie może być użyta jako typ argumentu, jako typ zwracany oraz jako typ w jawnej konwersji (zmiennymi, bądź stałymi tych typów muszą być obiekty).

Funkcja czysto wirtualna w deklaracji abstrakcyjnej klasy bazowej występuje jedynie w postaci specyficznego prototypu. Tym niemniej można zdefiniować ciało tej funkcji na zewnątrz deklaracji tej klasy i wywoływać ją za pomocą operatora zasięgu.

Przykład 8.12.

```
#include <iostream.h>
class Bazowa  {
public:
   virtual void czysta() = 0;
};
class Pochodna: public Bazowa {
public:
   Pochodna() { }
   void czysta()
   {
    cout << "Pochodna::czysta" << endl;
   }
   void ff() { Bazowa::czysta(); }
};
```

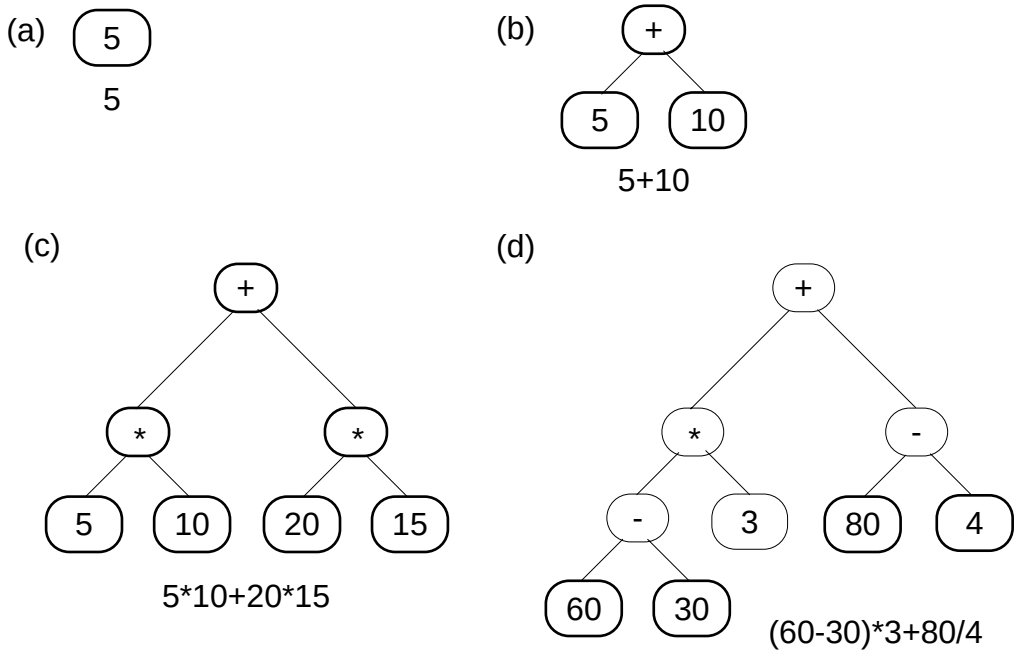
```
void Bazowa::czysta()
{ cout << "Bazowa::czysta" << endl; }
int  main() {
     Pochodna obiekt;
     obiekt.Bazowa::czysta();
     obiekt.ff();
     obiekt.czysta();
     Bazowa* wsk = &obiekt;
     wsk->czysta();
     return 0;
}
```

Wydruk z programu ma postać:

Bazowa::czysta
Bazowa::czysta
Pochodna::czysta
Pochodna::czysta

•

Podany niżej rysunek i przykład ilustruje konstrukcję i wartościowanie wyrażeń arytmetycznych, reprezentowanych przez tzw. drzewa wyrażeń.
W informatyce drzewem nazywa się strukturę złożoną z węzłów i gałęzi. Takie "informatyczne" drzewo przedstawia się graficznie w taki sposób, że najwyższym węzłem u góry jest korzeń drzewa, zaś węzły u samego dołu to jego liście, a więc odwrotnie, niż to czynimy przy rysowaniu drzew, występujących w przyrodzie. Drzewa wyrażeń należą do podstawowych struktur, stosowanych przy konstrukcji kompilatorów i interpretatorów. Jeżeli odniesiemy tę strukturę do terminologii obiektowej, to stwierdzimy, że korzeń drzewa odpowiada pewnej pierwotnej klasie bazowej, a kolejne węzły będą klasami pochodnymi, powiązanymi w hierarchię dziedziczenia. Ponieważ żywe drzewo rozgałęzia się w ten sposób, że zawsze z gałęzi grubszej wyrasta jedna lub więcej gałęzi cieńszych, zatem w naszym "drzewie" będziemy mieć schemat dziedziczenia pojedynczego.
Na rysunku 8-3 pokazano kilka przykładowych drzew wyrażeń.



Rysunek 8-3 Przykłady drzew wyrażeń

Najprostszym wyrażeniem jest stała, np. 5. Jej reprezentacją w drzewie wyrażenia jest część (a) rysunku 8-3. Drzewo stałej składa się z jednego węzła, który jest zarazem korzeniem i liściem. Proste wyrażenie arytmetyczne, 5 + 10, jest reprezentowane przez drzewo 8-3(b), w którym operator "+" odpowiada obiektowi klasy `Suma`, a stałe 5 i 10 – obiektom klasy `constant`. Części (c) i (d) rysunku prezentują wyrażenia o rosnącym stopniu złożoności.

Przykład 8.13.

```
#include <iostream.h>
class Wrn {
public:
  virtual int wart() = 0;
};

class Constant: public Wrn {
public:
  Constant(int k): x(k) {}
  int wart()  { return x;  }
private:
  int x;
};

class Suma: public Wrn {
public:
  Suma(Wrn* l, Wrn* p) { lewy = l;  prawy = p; }
  int wart() { return lewy->wart()+prawy->wart(); }
private:
  Wrn* lewy;
  Wrn* prawy;
};

class Odejm: public Wrn {
public:
  Odejm(Wrn* l, Wrn* p) { lewy = l; prawy = p; }
  int wart() { return lewy->wart()-prawy->wart(); }
private:
  Wrn* lewy;
  Wrn* prawy;
};

class Iloczyn: public Wrn {
public:
  Iloczyn(Wrn* l, Wrn* p) { lewy = l; prawy = p; }
  int wart() { return lewy->wart() * prawy->wart(); }
private:
  Wrn* lewy;
  Wrn* prawy;
};

class Iloraz: public Wrn {
public:
  Iloraz(Wrn* l, Wrn* p) { lewy = l; prawy = p; }
  int wart() { return lewy->wart() / prawy->wart(); }
private:
  Wrn* lewy;
  Wrn* prawy;
};

int  main() {
```

```
      Constant a(5);
      Constant b(10);
      Constant c(15);
      Constant d(20);
      Iloczyn  e(&a, &b);
      Iloczyn  f(&c, &d);
      Suma     g(&e, &f);
      Iloraz   h(&f, &e);
      cout << "a*b + c*d = " << g.wart() << endl;
      cout << "c*d / a*b = " << h.wart() << endl;
      return 0;
}
```

Wydruk z programu ma postać:

a*b + c*d = 350
c*d / a*b = 6

Dyskusja. W powyższym przykładzie starano się pokazać:

- Zastosowanie abstrakcyjnej klasy bazowej (tutaj klasa `Wrn`) z czysto wirtualną funkcją składową (funkcja `wart()`) do konstruowania drzewa dziedziczenia.
- Zastosowanie wskaźników do tak określonej klasy bazowej; zauważmy, że wskaźniki do klasy `Wrn` w klasach pochodnych są składowymi tych klas.
- Konstrukcję drzewa wyrażenia za pomocą struktury powiązanej wskaźnikami, w której węzłami są obiekty, a gałęziami wskaźniki. Dla tworzenia różnych wyrażeń zadeklarowano klasy pochodne `Constant`, `Suma`, `Odejm`, `Iloczyn`, `Iloraz`.
- Wartościowanie wyrażenia arytmetycznego za pomocą przekazywania komunikatów; komunikat (wywołanie funkcji) przesłany do obiektu-korzenia drzewa wyzwala przesyłanie kolejnych komunikatów do odpowiednich węzłów drzewa. Po zakończeniu wszystkich wywołań obiekt-korzeń jest w stanie odpowiedzieć na początkowy komunikat, podając wartość wyrażenia.

Klasa `Wrn` zawiera funkcję czysto wirtualną `wart()`, która jest redefiniowana w każdej z klas pochodnych. Dla dowolnego obiektu klasy pochodnej funkcja `wart()` zwraca wartość wyrażenia reprezentowanego przez swój obiekt i jego obiekty potomne. Oznacza to, że jeżeli wywołamy `wart()` dla korzenia drzewa lub poddrzewa, to funkcja zwróci wartość wyrażenia, reprezentowanego przez całe drzewo lub poddrzewo.

9. Strumienie i pliki

Strumień jest pewną abstrakcją, opisującą urządzenie logiczne, które albo "produkuje" albo "konsumuje" informację. W języku C++ operujemy na strumieniach danych, tzn. sekwencjach wartości tego samego typu, dostępnych w porządku sekwencyjnym. Oznacza to, że dostęp do n-tej wartości w strumieniu danych jest możliwy po uzyskaniu dostępu do poprzednich (n-1) wartości. Przez dostęp rozumiemy zarówno czytanie wartości, jak i wpisywanie
wartości do strumienia. Strumień może być dołączony do urządzenia fizycznego przez system wejścia/wyjścia dzięki odpowiednio zdefinowanym funkcjom czytania i zapisu. Dołączenie strumienia do urządzenia fizycznego jest realizowane w ten sposób, że strumień jest kojarzony z systemowym urządzeniem logicznym, w którym są zdefiniowane wymienione wyżej funkcje czytania i zapisu.
W języku C++ wszystkie strumienie zachowują się w ten sam sposób, co pozwala na dołączanie ich do urządzeń fizycznych o różnych własnościach. Tak więc możemy wykorzystać tę samą metodę do wyprowadzenia informacji na ekran, na plik dyskowy, czy drukarkę. Np. wejście jest sekwencją zdarzeń, które pojawiają się w systemie: znaki pisane na klawiaturze, wciśnięcie klawisza myszki, etc. Taka sekwencja zdarzeń może być wprowadzona do strumienia wejściowego.

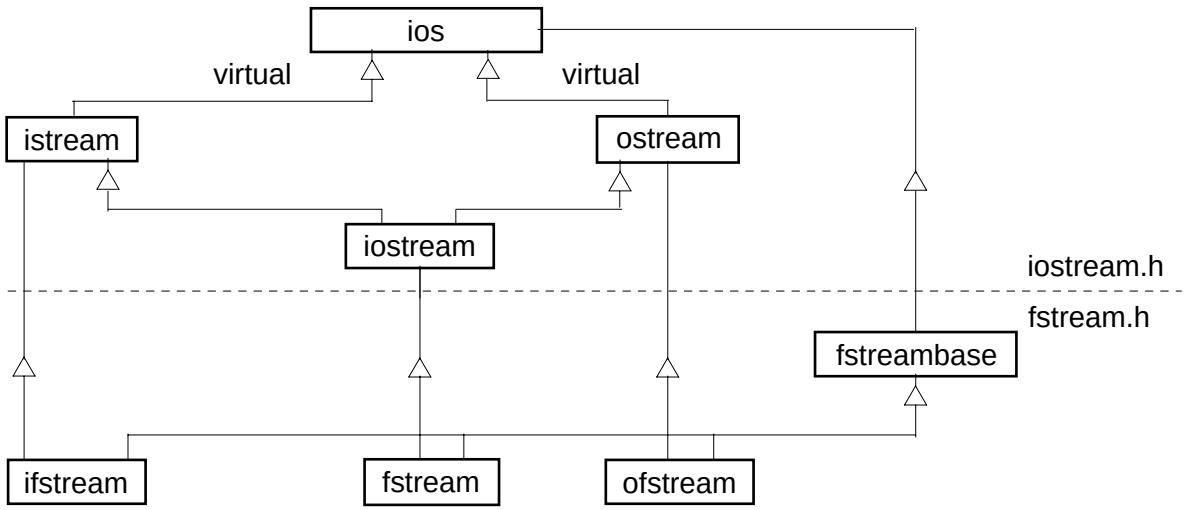
Uruchomienie programu w języku C++ powoduje automatyczne otwarcie czterech strumieni:

cin standardowe wejście (domyślnym urządzeniem fizycznym jest klawiatura)
cout standardowe wyjście (domyślnym urządzeniem fizycznym jest ekran monitora)
cerr standardowy błąd (ekran)
clog buforowana wersja cerr (ekran)

Ich deklaracje zawarte są w pliku **iostream.h**. Jeżeli użytkownik ma zamiar wprowadzać dane do programu z klawiatury i wyprowadzać wyniki na ekran, to musi włączyć ten plik do swojego programu.

9.1. Klasy strumieni wejścia/wyjścia

Strumienie języka C++ są niczym więcej, niż ciągami bajtów. Sposób interpretacji kolejnych bajtów w ciągu dla typów wbudowanych jest zawarty w definicjach klas strumieni. Dla typu (klasy) definiowanego w programie użytkownik może wykorzystać operacje dostępne w klasach strumieni, bądź przeciążyć te operacje na rzecz własnej klasy.
Podstawowe klasy strumieni wejścia/wyjścia są zdefiniowane w dwóch plikach nagłówkowych: `iostream.h` oraz `fstream.h`. Uproszczony schemat hierarchii tych klas pokazano na rysunku 9-1.



Rys. 9-1 Klasy strumieni we/wy

W pliku nagłówkowym `iostream.h` zawarte są deklaracje czterech podstawowych klas we/wy: `ios`, `istream`, `ostream` i `iostream`. Klasa `ios` jest klasą bazową dla `istream` i `ostream`, które z kolei są klasami bazowymi dla `iostream`. Klasa `ios` musi być wirtualną klasą bazową dla klas `istream` i `ostream`, aby tylko jedna kopia jej składowych była dziedziczona przez `iostream`:

```
class istream : virtual public ios { //... }
class ostream : virtual public ios { //... }
class iostream:public istream,public ostream { //... }
```

W klasie `ios` jest zadeklarowany wskaźnik do klasy `streambuf`, która jest abstrakcyjną klasą bazową dla całej rodziny klas buforów strumieni. Bufory te służą jako chwilowa pamięć dla danych z wejścia i wyjścia, a także jako sprzęgi łączące strumienie z urządzeniami fizycznymi.
Ponieważ klasy `istream` i `ostream` zawierają wskaźniki do innych klas, każda z nich (bądź klasa od niej pochodna) ma zdefiniowany własny operator przypisania.
Obiektem klasy `istream` jest wymieniony uprzednio strumień **cin**, zaś obiektami klasy `ostream` są strumienie **cout**, **cerr** i **clog**.
W klasie `istream` deklaruje się funkcje operatorowe `operator>>()`. Przeciążony operator pobrania '>>' służy do wprowadzania danych do programu ze strumienia **cin**, standardowo związanego z klawiaturą. Przykładowe prototypy tych funkcji mają postać:

```
istream& operator>>(signed char*);
istream& operator>>(int&);
istream& operator>>(double&);
```

Instrukcję wprowadzania danej ze strumienia **cin** zapisuje się w postaci:

```
cin >> zmienna;
```

gdzie zmienna zadeklarowanego typu określa wywołanie odpowiedniego przeciążonego operatora '>>'.

☞ *Uwaga. Operator "**>>**" pomija (przeskakuje) przy czytaniu tzw. białe znaki, czyli spacje, znaki tabulacji i znaki nowego wiersza. Należy o tym pamiętać przy wczytywaniu danych do zmiennych typu **char** i **char***.*

Przeciążony operator wstawiania "<<" jest skojarzony z buforowanym strumieniem **cout** i służy do wyprowadzania danych na ekran monitora (lub drukarkę).

Przykładowe prototypy funkcji operatorowych "<<" mają postać:

```
ostream& operator<<(short int);
ostream& operator<<(unsigned char);
ostream& operator<<(long double);
```

Instrukcję wyprowadzania (wstawiania do strumienia **cout**) wartości wyrażenia zapisuje się w postaci:

```
cout << wyrażenie;
```

Zwróćmy uwagę na fakt, że funkcje operatorowe dla operatorów “<<” i “<<” zwracają referencje do obiektów klas `istream` i `ostream`, dla których są wywoływane; dzięki temu możliwa jest konkatenacja operacji strumieniowych.

W pliku nagłówkowym `fstream.h` zadeklarowano klasy strumieni, kierowane do/z plików: `fstreambase`, `ifstream`, `fstream` i `ofstream`. Deklaracje tych klas i sposoby korzystania z ich obiektów omówimy w osobnym podrozdziale.

9.1.1. Funkcje składowe

W klasach `ios`, `istream` i `ostream` znajdujemy deklaracje szeregu funkcji składowych. W praktyce używa się kilku do kilkunastu z nich. W podanym niżej przykładzie wykorzystano funkcje o następujących prototypach:

```
int get();
```

zadeklarowaną w klasie `istream`
oraz

```
ostream& put(char);
```

zadeklarowaną w klasie `ostream`.
Funkcja `get()` pobiera i przekazuje następny znak ze strumienia wejściowego, zaś funkcja `put(char)` wstawia znak do strumienia wyjściowego. Są to funkcje niższego poziomu niż funkcje operatorowe “<<” i “>>”, bardzo przydatne w przypadku, gdy strumienie są traktowane jako ciągi bajtów, bez dodatkowych interpretacji określonych podciągów bajtów.

Przykład 9.1.

```
#include <iostream.h>
int   main() {
      char znak;
      while((znak = cin.get()) != '$')
      cout.put(znak);
      return 0;
}
```

Jeżeli z klawiatury wprowadzimy łańcuch znaków "abcd$" to wygląd ekranu będzie następujący:

abcd$
abcd

Jeżeli w skład łańcucha znaków wchodzą spacje, to będą one również wczytywane do zmiennej znak, co pokazuje następny wydruk:

a b c d$
a b c d

Przykład 9.2.

```
#include <iostream.h>
int   main() {
      char z, bufor[5];
      cin.get(bufor, 5, '\n');
      cin.getline(bufor, 5);//ten sam efekt
      cin >> bufor; // brak kontroli rozmiaru bufora
      cin.putback(bufor[2]);
      z = cin.peek();
      for(int i = 0; i < 5; i++) cout.put(bufor[i]);
```

```
      cout.put('\n');
      cout << z << endl;
      return 0;
}
```

Jeżeli wprowadzimy łańcuch znaków "abcdef", to wygląd ekranu będzie:

abcdef
abcd
c

Dla łańcucha zawierającego spacje: "a b c d e f" otrzymamy wydruk:

a b c d e f
a b
b

Dyskusja. W programie wykorzystano inną, przeciążoną wersję funkcji składowej `get()` klasy `istream` o prototypie:
`istream& get(char* buf, int num, char delim='\n');`
która czyta znaki do tablicy wskazywanej przez `buf` dotąd, dopóki nie wczyta `num` znaków lub dopóki nie napotka znaku, podanego jako `delim`. Ciąg znaków w `buf` zostanie zakończony przez funkcję znakiem zerowym. Jeżeli nie podamy wartości `delim`, to domyślnym znakiem będzie '`\n`'. Jeżeli w strumieniu wejściowym znajdzie się taki znak, to nie zostanie on z niego pobrany, lecz pozostanie w strumieniu aż do następnej operacji wprowadzania.
Funkcja `getline()` ma podobny prototyp:
`istream& getline(char* buf, int num, char delim='\n');`
i działa analogicznie za wyjątkiem tego, że pobiera i usuwa znak kończący wprowadzanie ze strumienia wejściowego.
Funkcja `putback()` o prototypie:
`istream& putback(char z);`
zwraca ostatnio pobrany (lub dowolnie wybrany z tablicy, jak w podanym przykładzie) znak do tego samego strumienia, z którego został pobrany.
Funkcja `int istream::peek()`, która pozwala "zaglądać" do wnętrza strumienia klasy `istream`, przekazuje następny znak (lub znak końca pliku EOF) bez usuwania go ze strumienia.
Zwróćmy uwagę na postać wydruków. Jeżeli wprowadzamy ciąg sześciu znaków "abcdef", to funkcja `get()` wczyta do tablicy `bufor[5]` tylko pierwsze cztery z nich (piątym będzie znak zerowy '`\0`'). Jeżeli zaś podamy znaki ze spacjami, jak w "a b c d e f", to również zostaną wczytane do tablicy cztery pierwsze znaki, a więc "a b ". Teraz `bufor[0] == 'a'`, zaś `bufor[2]== 'b'` i instrukcja `cin.putback(bufor[2]);` zwróci 'b' do strumienia cin.

Oferowane przez funkcje `get()` i `put()` możliwości można rozszerzyć, stosując funkcje `read()` i `write()` o prototypach:

```
istream& read(char* buf, int num);
ostream& write(char* buf, int num);
```

Funkcja `read()` czyta `num` bajtów ze skojarzonego z nią strumienia i wstawia je do bufora, wskazywanego przez `buf`. Funkcja `write()` zapisuje `num` bajtów z bufora wskazywanego przez `buf` do skojarzonego z nią strumienia.
Jeżeli funkcja `read()` napotka znak końca pliku (EOF) zanim przeczyta `num` bajtów, to skończy działanie, a bufor będzie zawierał tyle znaków, ile zostało wczytane. Do kontroli wczytywania można

wykorzystać funkcję klasy `istream` o prototypie `int gcount();`, która przekazuje liczbę znaków przeczytanych przez ostatnią operację wprowadzania danych.

9.2. Formatowanie wejścia i wyjścia

Klasa `ios` zawiera szereg dwuwartościowych *sygnalizatorów formatu* (ang. flags), które mogą być albo włączone (on) albo wyłączone (off). Wartości te decydują o sposobie interpretacji danych pobieranych ze strumienia wejściowego lub wysyłanych do strumienia wyjściowego. Zestawione niżej sygnalizatory związane są z każdym strumieniem (cin, cout, cerr, clog, strumienie plikowe).

ios::skipws	przeskocz białe znaki na wejściu
ios::left	justuj wyjście do lewej
ios::right	justuj wyjście do prawej
ios::internal	uzupełnij pole liczby spacjami
ios::dec	konwersja na system dziesiętny
ios::oct	konwersja na system ósemkowy
ios::hex	konwersja na system szesnastkowy
ios::showbase	wyświetl podstawę systemu liczenia
ios::showpoint	wyświetl kropkę dziesiętną
ios::uppercase	wyświetl 'X' dla notacji szesnastkowej
ios::showpos	dodaj '+' przed dodatnią liczbą dziesiętną
ios::scientific	notacja wykładnicza
ios::fixed	zastosuj notację z kropką dziesiętną
ios::unitbuf	opróżniaj każdy strumień po wstawieniu danych
ios::stdio	opróżniaj stdout i stderr po każdym wstawieniu danych

Wszystkie wartości sygnalizatorów są przechowywane w postaci określonego układu bitów danej typu **long int**. Gdy zaczyna się wykonanie programu, z każdym ze strumieni zostaje związany oddzielny zbiór sygnalizatorów z określonymi wartościami domyślnymi. Np. dla strumienia **cout** sygnalizatory `skips` i `unitbuf` są ustawione na "on", zaś pozostałe na "off". Użytkownik może sprawdzić ich ustawienie, wywołując funkcję `long int flags()` dla danego strumienia, np. w instrukcji: `long int li = cout.flags();` może też ustawić określone sygnalizatory na "on", korzystając z alternatywnej postaci funkcji `long int flags(long int)`, np. instrukcją:

```
cout.flags(ios::dec | ios::showpos);
```

Prześledźmy tę instrukcję. Funkcja składowa `flags()` klasy `ios` jest wywoływana z argumentem, będącym bitową alternatywą. Dzięki temu zostaną ustawione na "on" obydwa sygnalizatory, tj. `dec` i `showpos`.

Zastosowanie funkcji `flags()` do ustawiania sygnalizatorów bywa niezbyt wygodne, ponieważ włączając jeden lub kilka z nich, jednocześnie wyłącza pozostałe, których nie podano w jej argumencie. W takich razach należy raczej korzystać z funkcji składowej `long int setf(long)` i komplementarnej do niej funkcji `long int unsetf(long int)`, które nie dają wymienionego wyżej efektu ubocznego. Tak więc np. dla włączenia w strumieniu `cout` sygnalizatorów `showbase` i showpos, nie zmieniając ustawienia pozostałych, wystarczy napisać

```
cout.setf(ios::showbase | ios::showpos);
```

Sygnalizatory te można następnie wyłączyć instrukcją:

```
cout.unsetf(ios::showbase | ios::showpos);
```

Podobnie jak `flags()`, funkcje `setf()` i `unsetf()` mogą przekazać aktualne ustawienia sygnalizatorów. Np. wykonanie instrukcji

```
long int li = cout.setf(ios::showbase);
```

zachowa bieżące ustawienia sygnalizatorów w zmiennej `li`, po czym ustawi na "on" sygnalizator `showbase`.

☞ *Uwaga. Funkcje flags(), setf() i unsetf() są funkcjami składowymi klasy ios, a zatem oddziaływują na strumienie tworzone przez tę klasę. Dlatego wszelkie wywołania tych funkcji należy wykonywać dla konkretnego strumienia.*

Funkcja `ios::setf()` występuje również w alternatywnej postaci z dwoma argumentami: `long int setf(long int, long int)`. Tę postać funkcji wykorzystuje się do ustawiania sygnalizatorów, które są skojarzone z tzw. *polami bitowymi*. W klasie `ios` zdefiniowano trzy takie pola bitowe typu **static const long int**:

dla sygnalizatorów `left`, `right` i `internal` jest to pole `adjustfield`
dla sygnalizatorów `dec`, `oct` i `hex` jest to pole `basefield`
dla sygnalizatorów `scientific` i `fixed` jest to pole `floatfield`.

Sygnalizatory skojarzone z polami bitowymi wykluczają się wzajemnie – tylko jeden z nich może być włączony, a pozostałe wyłączone. Tak więc instrukcja

```
cout.setf(ios::oct, ios::basefield);
```

włączy `oct` i wyłączy pozostałe sygnalizatory (`dec` i `hex`) w tym polu, pozostawiając bez zmiany wszystkie inne sygnalizatory. Podobnie instrukcja

```
cout.setf(ios::left, ios::adjustfield);
```

włączy `left`, wyłączy `right` oraz `internal` i pozostawi bez zmiany pozostałe.

Jeżeli funkcję `setf(long int, long int)` wywołamy z pierwszym argumentem równym zeru, to wyłączy ona wszystkie sygnalizatory w podanym polu. Np. instrukcja

```
cout.setf(0, ios::floatfield);
```

wyłączy wszystkie sygnalizatory w `ios::floatfield`, a pozostawi bez zmiany wszystkie pozostałe.

W klasie `ios` znajdujemy szereg dalszych funkcji formatujących. Trzy z nich: `fill()`, `precision()` i `width()`, wykorzystano w poniższym przykładzie.

Przykład 9.3.

```
#include <iostream.h>
const double PI = 3.14159265353;
int  main()  {
     cout.fill('.');
     cout.setf(ios::left, ios::adjustfield);
     cout.width(12);
     cout << "Wyraz" << '\n';
     cout.setf(ios::right, ios::adjustfield);
     cout.width(12);
```

```
      cout << "Wyraz" << '\n';
      cout.width(10);
      cout << cout.width() << '\n';
      cout.setf(ios::showpos);
      cout.precision(9);
      cout << PI << '\n';
      return 0;
}
```

Wydruk z programu ma postać:

Wyraz.....
.....Wyraz
.....10
+3.141592654

Funkcje `ios::fill()`, `ios::precision()` i `ios::width()` służą do formatowania wyjścia. Każda z nich występuje również w postaci przeciążonej.
Przy wyprowadzaniu dowolnej wartości, zajmuje ona na ekranie tyle miejsca, ile potrzeba na wyświetlenie wszystkich jej znaków. Możemy jednak ustalić minimalną szerokość w pola wydruku, wywołując funkcję o prototypie

```
int width(int w);
```

która ustala nową szerokość pola w i przekazuje do funkcji wołającej dotychczasową szerokość. Wywołanie przeciążonej wersji tej funkcji

```
int width() const;
```

przekazuje jedynie aktualną szerokość pola wydruku.

Jeżeli ustawimy szerokość pola wydruku na `w`, to przy wyprowadzaniu wartości, która zajmuje mniej niż `w` znaków, pozostałe pozycje znakowe zostaną uzupełnione aktualnie ustawionym znakiem wypełniającym. Domyślnym znakiem wypełniającym jest spacja. Jeżeli jednak wyprowadzana wartość zajmuje więcej niż `w` znaków, to będą wyprowadzone wszystkie znaki, a więc w tym przypadku ustawiona szerokość pola zostanie zignorowana.
Znak wypełniający wolne miejsca w polu wydruku można ustalić za pomocą funkcji

```
char fill(char z);
```

która ustala nowy znak na `z` i przekazuje do funkcji wołającej znak dotychczasowy. Wersja bezparametrowa tej funkcji

```
char fill() const;
```

przekazuje jedynie aktualny znak wypełniający.

Przy wyprowadzaniu wartości zmiennopozycyjnych są one drukowane z domyślną dokładnością sześciu miejsc po kropce dziesiętnej. Jeżeli chcemy mieć inną dokładność wydruku, wywołujemy funkcję składową

```
int precision(int p);
```

która ustala dokładność na p miejsc po kropce dziesiętnej i przekazuje do funkcji wołającej dotychczasową liczbę miejsc. W wersji bezparametrowej

```
int precision() const;
```

funkcja ta przekazuje jedynie aktualną liczbę miejsc po kropce dziesiętnej.

9.2.1. Manipulatory

Formatowanie wejścia i wyjścia, tj. wprowadzanie zmian stanu strumieni **cin** i **cout** za pomocą sygnalizatorów jest w praktyce dość kłopotliwe. Weźmy dla ilustracji następujący przykład.

Przykład 9.4.

```
#include <iostream.h>
int   main() {
      int ii;
      cin.setf(ios::hex, ios::basefield);
      cin >> ii;
      cout << ii << endl;
      cout.setf(ios::oct, ios::basefield);
      cout << ii << endl;
      return 0;
}
```

Wydruk z programu po wprowadzeniu ii==10 ma postać:

10
16
20

Dyskusja. W momencie startu programu jest włączony (ustawienie domyślne) sygnalizator dec podstawy liczenia, a pozostałe sygnalizatory w polu basefield (oct i hex) są wyłączone. Pierwsza instrukcja wywołująca funkcję setf() włącza sygnalizator hex, a wyłącza pozostałe w tym polu. Dlatego wartość wczytana do zmiennej ii będzie traktowana jako liczba szesnastkowa, co widać w drugim wierszu wydruku (strumień **cout** ma nadal stan domyślny, z włączonym sygnalizatorem dec). Po zmianie stanu strumienia **cout** wprowadzonej wykonaniem instrukcji
`cout.setf(ios::oct, ios::basefield);`
wstawiana do strumienia wartość ii będzie interpretowana jako liczba ósemkowa, tj. liczba 20.

Dla uproszczenia notacji przy formatowaniu wejścia i wyjścia wprowadzono w języku C++ alternatywną metodę zmiany stanu strumieni. Metoda ta wykorzystuje specjalne funkcje, nazywane *manipulatorami strumieniowymi* lub *manipulatorami wejścia/wyjścia*. Manipulatory dzielą się na bezargumentowe, zadeklarowane w pliku iostream.h oraz jednoargumentowe, zadeklarowane w pliku iomanip.h.

Tablica 9.1 Manipulatory strumieniowe

dec	Konwersja na liczbę dziesiętną
hex	Konwersja na liczbę szesnastkową
oct	Konwersja na liczbę ósemkową
endl	Prześlij znak NL i opróżnij strumień
flush	Opróżnij strumień
ws	Pomiń spacje
setbase(int b)	Ustal typ konwersji na b
setfill(int z)	Ustal znak dopełniający pole na z
setprecision(int p)	Ustal liczbę miejsc po kropce dziesiętnej
setw(int w)	Ustal szerokość pola na w
setiosflags(long int f)	Włącz sygnalizatory podane w f
resetioflags(long int f)	Wyłącz sygnalizatory podane w f

Manipulatory strumieniowe wywołuje się w ten sposób, że po prostu wstawia się ich nazwy (ewent. z parametrem) w łańcuch operacji wejścia/wyjścia, np.

```
cout << oct << 127 << hex << 127;
cout << setw(4) << 100 << endl;
```

Zauważmy przy okazji, że wcześniej poznaliśmy już manipulator `endl`, który wstawia znak nowego wiersza i opróżnia bufor wyjściowy oraz manipulator z parametrem `setw(int)`.

Manipulator setw(int) jest szczególnie użyteczny przy wczytywaniu łańcuchów znaków. Np. sekwencja instrukcji:

```
char buffer[8];
cin >> setw(8) >> buffer;
```

powoduje wczytanie łańcucha znaków do tablicy znaków `buffer`. Manipulator `setw(8)`, który wyznacza rozmiar tej tablicy znaków, zapobiega przepełnieniu bufora. Inaczej mówiąc, do `buffer` zostanie wczytane co najwyżej 7 znaków, dzięki czemu pozostanie miejsce na terminalny znak zerowy ('\0'), który występuje na końcu każdego łańcucha znaków.

Przykład 9.5.

```
#include <iostream.h>
#include <iomanip.h>
int  main() {
     int ii;
     cout << setiosflags(0x200);
     cout << hex;
     cout << 15 << endl;
     cin >> ii;
     cout << ii << endl;
     cout << dec << ii << endl;
     cout << 127 << setw(4) << hex << 127
```

```
            << oct << setw(4) << 127 << endl;
      return 0;
}
```

Wydruk z programu ma postać:

F
10
A
10
127 7F 177

☞ ***Komentarz****. Sygnalizatory w klasie ios są zadeklarowane w postaci wyliczenia (enum), w którym np. ios::uppercase ma przypisaną wartość 0x0200 (dziesiętnie 512, oktalnie 01000); stąd wartość argumentu funkcji setiosflags(0x200), która ustawia duże litery dla notacji szesnastkowej.*

9.3. Pliki

We wprowadzeniu do tego rozdziału stwierdzono, że strumienie, czyli obiekty klas strumieniowych, można kojarzyć z predefiniowanymi urządzeniami logicznymi. Urządzenia te, nazywane niekiedy plikami specjalnymi, służą do komunikacji programu z otoczeniem, tj. z reprezentowanymi przez nie urządzeniami fizycznymi. Zauważmy przy okazji, że jedynymi plikami specjalnymi, bezpośrednio dostępnymi z obiektów klas zadeklarowanych w `iostream.h` są nienazwane urządzenia, dołączane automatycznie do strumieni `cin`, `cout`, `cerr` i `clog`. Dla dostępu do plików nazwanych, takich jak pliki dyskowe, musimy korzystać z klas strumieni zadeklarowanych w pliku nagłówkowym `fstream.h` (rys. 9-1):

```
class fstreambase : virtual public ios { };
class ifstream: public fstreambase,public istream {};
class ofstream: public fstreambase,public ostream {};
class fstream: public fstreambase,public iostream {};
```

Ponieważ klasy te są klasami pochodnymi od `ios`, `istream`, `ostream` i `iostream`, zatem mają one dostęp do wszystkich elementów publicznych i chronionych swoich klas bazowych. Strumienie wejściowe muszą być obiektami klasy `ifstream`; strumienie wyjściowe – obiektami klasy `ofstream`. Strumienie, które mogą wykonywać zarówno operacje wejściowe, jak i wyjściowe, muszą być obiektami klasy `fstream`.
Jeżeli zadeklarujemy jakiś obiekt (strumień) jednej z klas, np.

```
ifstream iss;
```

to możemy go skojarzyć z konkretnym plikiem za pomocą funkcji składowej tej klasy, w tym przypadku `ifstream::open()`, np.

```
iss.open("plikwe.doc", ios::in);
```

Funkcja `open()` wykonuje szereg operacji, określanych jako *otwarcie pliku*. Prototyp funkcji `open()` ma następującą postać:

```
void open(char* nazwa, int tryb, int dostęp);
```

gdzie: zmienna `nazwa` jest nazwą otwieranego pliku,
stała `tryb` określa sposób otwarcia pliku,

zmienna `dostęp` określa prawa dostępu do pliku.

Wartości argumentów funkcji `open()` mogą być następujące.

- Wartościami zmiennej nazwa mogą być łańcuchy znaków, zapisywane zgodnie z zasadami obowiązującymi w danym systemie operacyjnym.

- Stała tryb może być jedną ze stałych, zdefiniowanych w klasie `ios`:

`ios::app`
Dopisuj nowe dane na końcu istniejącego pliku. Utwórz plik, jeżeli nie istnieje. Może wystąpić tylko dla obiektów klas `ofstream` i `fstream`.

`ios::ate`
Wymusza, po otwarciu, przejście na koniec pliku. Może wystąpić dla obiektów wszystkich trzech klas.

`ios::in`
Otwórz plik do odczytu. Może wystąpić dla obiektów klas `ifstream` i `fstream`.

`ios::nocreate`
Powoduje nieudane wykonanie funkcji `open()`, jeżeli plik nie istnieje.

`ios::noreplace`
Powoduje nieudane wykonanie funkcji `open()`, jeżeli plik już istnieje, chyba że podano również `app` lub `ate`.

`ios::out`
Otwórz plik do zapisu. Jeżeli plik już istnieje, wyzeruj jego zawartość; utwórz plik, jeżeli nie istnieje. Może wystąpić dla obiektów klas `ofstream` i `fstream`.

`ios::trunc`
Wyzeruj zawartość istniejącego pliku o takiej samej nazwie, jak podana dla zmiennej nazwa.

Z wyliczonych wyżej stałych można tworzyć alternatywy za pomocą bitowego operatora "`|`". Np. tryb

```
ios::in | ios::out
```

pozwala zarówno na odczyt, jak i zapis (tylko dla obiektu klasy `fstream`).
Jeżeli chcemy zachować dane w istniejącym już pliku, to ustawimy tryb:

```
ios::in | ios::out | ios::ate
```

- Zmienna `dostęp` ma wartość domyślną `filebuf::openprot`, gdzie `static const int openprot` jest liczbą, określającą prawa dostępu. Dla systemu Unix `openprot==0644` (read i write dla właściciela pliku i tylko read dla pozostałych); zgodnie z regułami dostępu wartość ta może być dowolną liczbą z zakresu 0000-0777. Dla systemu MS-DOS wartość domyślna `openprot==0`; może ona wynosić: 0 – dla swobodnego dostępu do pliku, 1 – dla pliku tylko do czytania, 2 – dla pliku ukrytego, 4 – dla pliku systemowego i 8 – dla ustawienia bitu archiwizacji.

Deklarację strumienia można połączyć z instrukcją otwarcia pliku podając nazwę pliku i tryb dostępu jako argumenty konstruktora odpowiedniej klasy. Np. instrukcja

```
ifstream obin("plikwe.doc",ios::in, filebuf::openprot);
```

deklaruje obiekt `obin` klasy `ifstream`, wiąże go z plikiem o nazwie `plikwe.doc`, ustala tryb na `ios::in`, a dostęp na `filebuf::openprot`.

Ponieważ konstruktory omawianych klas są zdefiniowane z domyślnymi wartościami argumentów (stałej `tryb` i zmiennej `dostęp`), to podaną wyżej deklarację wystarczy napisać w postaci:
`ifstream obin("plikwe.doc");`
(`tryb==ios::in, dostęp==filebuf::openprot`)

a deklarację otwarcia pliku na pisanie np. w postaci:
`ofstream obout("plikwy.txt");`
(`tryb==ios::out, dostęp==filebuf::openprot`)

W deklaracji strumienia nazwę pliku można poprzedzić nazwą katalogu, np.
"`c:\borlandc\plikwe.doc`" (MS-DOS),
czy "`/home/mike/plikwe.doc`" (Unix).

Podobnie jak dla zwykłych plików, strumienie można kojarzyć z plikami specjalnymi, reprezentującymi inne urządzenia fizyczne. Np. deklaracja (MS-DOS: `ofstream druk("lpt1");` kieruje dane, wstawiane do strumienia druk na drukarkę.

Przykład 9.6.

```
#include <fstream.h>
#include <stdlib.h>
int   main()  {
      ofstream ofs;
      ofs.open("plik1.doc",ios::out,filebuf::openprot);
      if ( !ofs )
      {
       cerr << "Nieudane otwarcie pliku do zapisu\n";
       exit( 1 );
      }
      ofs << "To jest pierwszy wiersz tekstu, \n";
      ofs << "a to drugi.\n";  */
      ofs.close();
      return 0;
}
```

Dyskusja. Program otwiera do zapisu plik `plik1.doc`. Jeżeli nie było pliku o takiej nazwie na dysku, to zostanie założony. Zwróćmy uwagę na kilka szczegółów.
Mimo że nie dołączyliśmy pliku `iostream.h`, używany jest operator wstawiania "<<" i to nie do strumienia" `cout`, lecz do zadeklarowanego przez nas strumienia `os`. Mogliśmy tak zrobić, ponieważ klasa `ofstream` odziedziczyła ten operator od klasy `ostream`. W programie umieszczono wywołanie funkcji składowej `close()` zamknięcia pliku `os`. Jest to funkcja składowa klasy `fstreambase`, odziedziczona od niej przez klasę `ofstream`. Wywołanie to nie było konieczne, ponieważ jest ono wykonywane automatycznie przy zakończeniu programu. W instrukcji `if` wykorzystano przeciążony na rzecz klasy `ios` logiczny operator "`!`" do sprawdzenia, czy otwarcie pliku zakończyło się powodzeniem. Zauważmy też, że utworzony (lub na nowo zapisany) plik ma zawartość zero (0). Gdyby wymazać znaki komentarza (/* i */), to dwie ostatnie instrukcje wpisałyby do pliku podane dwa wiersze tekstu.

Przykład 9.7.

```
#include <fstream.h>
int   main() {
      char znak;
      char* tekst1 = "Tekst w pliku plik1.txt";
      char* tekst2 = "Tekst w pliku plik2.txt\n";
      char* tekst3 = "Tekst dodawany";
      ofstream ofs1("plik1.txt");
      ofstream ofs2("plik2.txt");
      ofs1 << tekst1; ofs1.close();
      ofs2 << tekst2 << tekst3;
      ofs2.close();
      ifstream ifs1("plik2.txt");
      ofstream ofs3("plik3.txt");
      while (ofs3&&ifs1.get(znak)) ofs3.put(znak);
      ifs1.close(); ofs3.close();
      return 0;
}
```

Dyskusja. Program tworzy pliki `plik1.txt` i `plik2.txt`. Do pierwszego z nich wpisuje łańcuch `tekst1`, zaś do drugiego najpierw łańcuch `tekst2`, a następnie (konkatenacja) łańcuch `tekst3`. Zauważmy, że łańcuch `tekst3` jest dopisywany po znaku nowego wiersza, którym kończy się łańcuch `tekst2`. Obydwa pliki są jawnie zamykane, po czym plik `plik2.txt` zostaje otwarty do odczytu, a plik `plik3.txt` (chwilowo pusty) do zapisu. W instrukcji `while` wykonywane jest kopiowanie zawartości pliku `plik2.txt` do pliku `plik3.txt`; funkcja `get()` czyta kolejne znaki z `ifs1`, a funkcja `put()` wpisuje je do `ofs3`. W wyrażeniu instrukcji `while` wykonywana jest konwersja strumienia do wartości *prawda* (wartość różna od zera), jeżeli nie zdarzył się błąd zapisu. Wartość `ifs.get(znak)` jest referencją do `ifs`, która jest przekształcana na wartość *prawda*, jeżeli nie wystąpił błąd podczas czytania znaku ze strumienia `ifs`. Te dwie wartości są w logicznej koniunkcji, a zatem kopiowanie będzie biegło tak długo, jak długo obydwie będą różne od zera. Gdy `get()` napotka koniec pliku wejściowego, to wystąpi błąd w operacji czytania, wartość `ifs.get(znak)` zmieni się na *fałsz* (zero) i program wyjdzie z pętli `while`. Pętlę `while` można też zapisać w postaci:

```
while(!ifs1.eof()&&ifs1.get(znak)) ofs3.put(znak);
```

w której funkcja `eof()` przekazuje wartość niezerową (prawda) tylko wtedy, gdy zostanie napotkany koniec pliku.

9.3.1. Plik jako parametr funkcji main

W rozdziale 5 przedyskutowano komunikację funkcji `main()` z otoczeniem, tj. z systemem operacyjnym. Przypomnijmy, że wykonanie każdego programu zaczyna się od wykonania pierwszej instrukcji funkcji `main()`, a ostatnią wykonywaną instrukcją jest instrukcja `return` tej funkcji. Prawie we wszystkich naszych programach funkcja `main()` występowała z pustym wykazem argumentów; wiadomo jednak, że może ona mieć wiele argumentów, ponieważ jej prototyp ma postać:

```
int main(int argc, char* argv[]);
```

gdzie argument `argv` jest tablicą łańcuchów znaków, a `argc` jest w chwili uruchomienia programu inicjowany liczbą tych łańcuchów. Ponieważ nazwy plików są łańcuchami znaków, zatem nic nie stoi

na przeszkodzie, aby nazwy te były argumentami aktualnymi funkcji `main()`. Ilustrują to pokazane niżej dwa przykłady.

Przykład 9.8.

```
#include <fstream.h>
int   main(int argc, char* argv[]) {
      char znak;
      if(argc != 2)
      {
       cerr << "Napisz: czytaj <nazwa-pliku>\n";
       return 1;
      }
      ifstream ifs(argv[1]);
      if(!ifs)
      {
       cerr << "Nieudane otwarcie pliku do odczytu\n";
       return 1;
      }
      while(!ifs.eof())
      {
       ifs.get(znak);
       cout << znak;
      }
      return 0;
}
```

Dyskusja. Program wyświetla zawartość dowolnego pliku na ekranie. Jeżeli nazwa skompilowanego pliku ładowalnego (po konsolidacji) z naszym programem jest `czytaj` (lub `czytaj.exe` pod MS-DOS), to program wywołamy z wiersza rozkazowego systemu operacyjnego pisząc:

czytaj nazwa-pliku

Zauważmy, że czytanie zawartości pliku odbywa się w pętli `while`, a warunkiem zakończenia jest wystąpienie znaku końca pliku, gdy wartość przekazywana z funkcji `int ios::eof()` stanie się różna od zera (prawda). Znaki z otwartego do czytania pliku pobierane są ze strumienia `ifs` do zmiennej `znak` za pomocą funkcji `get()`, a nie operatora “>>” ponieważ ten ostatni pomija znaki spacji.

Przykład 9.9.

```
#include <fstream.h>
#include <stdlib.h>
int   main(int argc, char* argv[]) {
      char znak;
      if(argc != 3)
      {
       cerr << "Niepoprawna liczba parametrow\n";
       exit (1);
      }
      ifstream ifs(argv[1]);
      if(!ifs)
      {
       cerr << "Nieudane otwarcie pliku do odczytu\n";
       exit(1);
      }
```

```
      ofstream ofs(argv[2], ios::noreplace);
      if(!ofs)
      {
       cerr << "Nieudane otwarcie pliku do zapisu\n";
       exit(1);
      }
      while(ofs && ifs.get(znak)) ofs.put(znak);
      return 0;
}
```

Dyskusja. Program kopiuje zawartość pliku wejściowego do pliku wyjściowego. Wywołujemy go z trzema parametrami w wierszu rozkazowym; jeżeli np. plik wykonalny z programem ma nazwę "`kopiuj`", plik do skopiowania "`plik.we`", a plik-kopia "`plik.wy`", to wywołanie ma postać:
`kopiuj plik.we plik.wy`
Zauważmy, że plik wyjściowy otwarto w trybie `noreplace`, a więc nie jest możliwe skopiowanie pliku "`plik.we`" na już istniejący plik "`plik.wy`".

9.3.2. Dostęp swobodny

W podanych do tej chwili przykładach plikowych operacji wejścia/wyjścia wykorzystywaliśmy dostęp sekwencyjny: np. odczytanie n-tej danej w pliku było możliwe po odczytaniu (n-1) poprzednich danych.
W zastosowaniach plików, szczególnie w bazach danych, bardzo przydatna byłaby możliwość "zajrzenia" w dowolne miejsce pliku bez konieczności przeglądania pliku od początku. Język C++ stwarza taką możliwość dzięki wprowadzeniu w systemie wejścia/wyjścia dwóch wskaźników skojarzonych z plikiem. Pierwszym z tych wskaźników jest *wskaźnik pobierania* (ang. get pointer), który wskazuje miejsce następnej operacji wejściowej. Drugim jest *wskaźnik wstawiania* (ang. put pointer), który wskazuje miejsce następnej operacji wyjściowej. Wskaźniki te są przesuwane automatycznie o jedną pozycję w kierunku końca pliku po każdej operacji wejścia lub wyjścia. Jednakże użytkownik może przejąć kontrolę nad jednym lub obydwoma wskaźnikami za pomocą zadeklarowanych w klasach `istream` i `ostream` (plik nagłówkowy `iostream.h`) funkcji składowych o prototypach:

```
istream& seekg(streamoff offset, ios::seek_dir origin);
streampos tellg();

ostream& seekp(streamoff offset, ios::seek_dir origin);
streampos tellp();
```

- Parametr `offset` podajemy dla określenia, o ile bajtów w stosunku do pozycji `origin` chcemy przesunąć jeden lub obydwa wskaźniki pliku. Typy parametrów są następujące:
- typ streamoff jest wprowadzony deklaracją
  `typedef long int streamoff;`
- typ streampos jest wprowadzony deklaracją
- `typedef long int streampos;`
- typ `ios::seek_dir` jest zdefiniowanym w klasie `ios` wyliczeniem
  `enum seek_dir { beg=0, cur=1, end=2 };`

Dla obiektów klasy `ifstream` możemy wywoływać funkcję `seekg()`, np.

```
ifstream obin("plik.we");
obin.seekg(7, ios::beg);
```

oznacza ustawienie wskaźnika pobierania o 7 bajtów w prawo od początku pliku `plik.we`. Podobnie dla obiektów klasy `ofstream` wywołujemy funkcję `seekp()`, np.

```
ofstream obout("plik.wy");
obout.seekp(-7, ios::cur);
```

oznacza ustawienie wskaźnika wstawiania o 7 bajtów w lewo od bieżącej pozycji w pliku `plik.wy`.
Dla obiektów klasy `fstream` możemy wywoływać obie funkcje, w zależności od kontekstu.
Funkcje `tellg()` i `tellp()` typu **streampos** (synonim **long int**) służą do odczytu bieżącego położenia każdego ze wskaźników. Przykładowe wywołania:

```
long int l1 = obin.tellg();
long int l2 = obout.tellp();
```

Przykład 9.10.

```
#include <fstream.h>
int  main() {
     char znak;
     ifstream ifs("plik.we");
     if(!ifs)
     {
      cerr<<"Nieudane otwarcie pliku do odczytu\n";
      return 1;
     }
     ifs.seekg( 0, ios::beg);
     streampos poz1 = ifs.tellg();
     cout << poz1 << endl;
     do
     {
      ifs.get(znak);
      if(znak != EOF)
      {
       cout << znak << ' ';
       poz1 = ifs.tellg();
       cout << poz1 << ' ';
      }

     } while(!ifs.eof());
     cout << endl;
     ifs.seekg(0, ios::end);
     int i = ifs.tellg() ;
     cout << i << endl;
     ifs.close();
     return 0;
}
```

Dyskusja. Po otwarciu pliku `plik.we` do odczytu, ustawiamy wskaźnik pobierania na pierwszy bajt tego pliku, a jego położenie zapisujemy w zmiennej `poz1`. W pętli **do** przesuwamy wskaźnik pobrania instrukcją `ifs.get(znak);` notując w zmiennej `poz1` jego kolejne położenia. Pętla kończy się testem na wartość funkcji składowej `eof()`; wartość ta przed osiągnięciem końca pliku wynosi cały czas zero; na końcu pliku funkcja `eof()` przekazuje wartość niezerową. Końcowa sekwencja instrukcji służy do pokazania, że wskaźnik pobrania po odczytaniu zawartości ustawił się na końcu pliku. Jeżeli w utworzonym wcześniej pliku `plik.we` umieścimy ciąg znaków "abcdefghij", to wygląd ekranu po wykonaniu programu będzie następujący (liczby po literach są kolejnymi odległościami – w bajtach – wskaźnika pobrania od początku pliku):

0
a 1 b 2 c 3 d 4 e 5 f 6 g 7 h 8 i 9 j 10
10

Następny program ilustruje dostęp swobodny na przykładzie pliku klasy `fstream`, otwieranego w trybie `nocreate`. Oznacza to, że próba otwarcia do zapisu pliku nieistniejącego nie spowoduje jego utworzenia, lecz spowoduje przerwanie wykonania programu.

Przykład 9.11.

```
#include <fstream.h>
int  main() {
     char    tekst[] = "Tekst w pliku";
     fstream plik;
     char* nazwa;
     int i = 0;
     char znak;
     cout << "Podaj nazwe pliku: ";
     cin  >> nazwa;
 //Otwieramy plik do odczytu i zapisu.
     plik.open(nazwa,ios::in|ios::out|ios::nocreate);
     if (!plik)
     {
      cerr << "\nNieudane otwarcie pliku "
           << nazwa << endl;
     return 1;
     }
     cout << "Tekst wejsciowy: " << tekst << endl;
//Teraz zapisujemy tekst do pliku.
     while (znak = tekst[i++]) plik.put (znak);
//A teraz wypisujemy tekst od konca.
     cout << "Tekst odwrotny: ";
     plik.seekg (-1, ios::end);
     long int l;
     do {
     if ((znak = plik.get())!= EOF) cout << znak;
     plik.seekg (-2, ios::cur);
     l = plik.tellg();
     } while (l != -1);
     cout << endl;
     plik.close();
     return 0;
}
```

Dyskusja. Program najpierw zastępuje zawartość istniejącego pliku ciągiem znaków "Tekst w pliku" (instrukcja `while`). Następnie ustawia wskaźnik pobrania na ostatnim znaku w pliku (`plik.seekg (-1, ios::end);`) i odczytuje tę nową zawartość w odwrotnym kierunku. Jeżeli istniejącym plikiem był plik o nazwie "plik1.we", to wygląd ekranu będzie następujący:

Podaj nazwe pliku: plik1.we

Tekst wejsciowy: Tekst w pliku
Tekst odwrotny: ukilp w tskeT

10. Obsługa wyjątków

W języku potocznym przyjęło się mówić, że “wyjątek potwierdza regułę”. O ile maksyma ta raczej nie ma odniesienia do języka programowania, o tyle może się odnosić do osób piszących programy oraz do ograniczonych zasobów systemu.
W ogólności *wyjątkiem* (ang. exception) nazywamy zdarzenie, spowodowane przez anormalną sytuację, wymagającą przywrócenia normalnego stanu. Dobrze zaprojektowany system obsługi wyjątków powoduje wówczas zawieszenie normalnego wykonywania programu i przekazanie sterowania do odpowiedniej procedury obsługi wyjątku (ang. exception handler).

Wyjątki w środowisku programowym mogą pochodzić z różnych źródeł i występować na różnych poziomach: sprzętowym, programu i systemu.

- Na najniższym poziomie, nazwijmy go sprzętowym, mogą występować różne tego typu zdarzenia, w tym:
- błędy parzystości pamięci, generujące niemaskowalne przerwanie niskiego poziomu;
- niesprawność urządzenia zewnętrznego, np. wyłączenie drukarki lub brak papieru, otwarte drzwiczki napędu dysków, brak dyskietki w napędzie;
- uszkodzenie urządzenia zewnętrznego.

Błędy te są asynchroniczne względem programu i nie mają związku z tym, co akurat wykonuje program, a więc nie będą przechwytywane przez mechanizm obsługi wyjątków.

- Zdarzenia wymagające reakcji na poziomie programu. Występujące tutaj błędy mogą mieć różne przyczyny: błędny format danych wprowadzanych przez użytkownika, próba usunięcia nieistniejącego pliku, próba obliczenia logarytmu lub pierwiastka z liczby ujemnej, próba dzielenia przez zero, próba pobrania elementu z pustego stosu, próba wywołania nieistniejącej funkcji wirtualnej, etc.

- Na poziomie systemowym do najczęściej występujących błędów możemy zaliczyć brak pamięci przy próbie utworzenia nowego obiektu, albo brak miejsca na dysku.

Przy tradycyjnym podejściu do programowania reakcje na błędy można pogrupować w następujące kategorie.
1. Zaniechanie wykonania programu i wysłanie komunikatu o błędzie.
2. Przekazanie do programu wartości reprezentującej błąd.
3. Zignorowanie błędu.
4. Przekazanie do programu poprawnej wartości i przesłanie informacji o błędzie przez specjalną zmienną.
5. Wywołanie procedury obsługi błędu, napisanej przez programistę.

Każde z tych rozwiązań ma trudne do zaakceptowania wady.

Pierwsze z nich może być stosowalne w takich programach jak edytory, kompilatory, gry, etc. Z reguły wystarcza wtedy wyczyszczenie pamięci, zwolnienie wykorzystywanych zasobów systemu (np. zamknięcie plików), wydruk komunikatu i wyjście z programu. Jednak jest ono nie do przyjęcia w takich programach interakcyjnych, w których program reagujący na najmniejsze potknięcie operatora zakończeniem działania mógłby go pozbawić wyników długotrwałej pracy.

Drugie rozwiązanie jest na ogół niełatwe w implementacji, ponieważ w wielu przypadkach trudno jest odróżnić wartość poprawn¹ od b³Ydnej. £atwym przypadkiem jest odróænienie b³Ydnego wskaźnika. Np. funkcja czytająca dane z pliku może albo zwrócić wskaźnik do następnej pozycji, albo wskaźnik zerowy; podobnie funkcja typu `char*` może zwracać pusty łańcuch dla sygnalizacji niepowodzenia.

Nie ma natomiast sposobu określenia błędnego kodu dla funkcji typu int, ponieważ każda wartość zwracana może być uważana za poprawną. Zresztą nawet w przypadkach, gdy takie rozwiązanie jest dopuszczalne, może się okazać nieopłacalne, ponieważ sprawdzanie poprawności wyniku przy każdym wywołaniu funkcji pociąga za sobą duże narzuty czasowe i pamięciowe.

Trzecie rozwiązanie jest trudne w implementacji i na ogół niebezpieczne. Nie jest łatwym brak reakcji na błąd, szczególnie w odniesieniu do funkcji, która zwraca wartość inną niż `void`, czy `void*`. Zdarzają się także sytuacje, w których brak reakcji jest niedopuszczalny, np. gdy konstruktor kopiujący ustali, że nie ma dość pamięci dla utworzenia kopii obiektu (w tym przypadku nie będzie to, rzecz jasna, błąd użytkownika).

Rozwiązanie czwarte jest stosowane w standardowych bibliotekach języka C. Wiele funkcji z tych bibliotek (np. funkcje matematyczne) sygnalizuje błąd, ustawiając wartość EDOM (błąd dziedziny) lub ERANGE (błąd zakresu) w zmiennej **errno**, np.

```
double sqrt(double x)
{ if(x < 0) { errno = EDOM; return 0; } //... }
```

Jest to mechanizm niezbyt pomocny dla użytkownika. Komunikat o błędzie zawiera w tym przypadku jedynie typ błędu, a nie nazwę błędnie wywołanej funkcji. Co więcej, jeśli zdarzą się dwa kolejne błędy, to drugi może przysłonić komunikat o pierwszym. Oczywiście takim sytuacjom można zapobiec, ale znowu kosztem sporego narzutu pamięciowego i czasowego.

Rozwiązanie piąte spotyka się w dwóch wariantach. Używane w programie klasy można wyposażyć w domyślne funkcje obsługi błędów. Zwykle są to funkcje, które drukują komunikat o błędzie i powodują zakończenie programu, tak jak to czyniliśmy w wielu przykładowych programach. Możliwe jest także zadeklarowanie funkcji, która np. zapisuje komunikat o błędzie do pliku i pozwala na kontynuację wykonywania programu. Jednak takie rozwiązanie nie będzie mieć cech ogólności, ponieważ w zasadzie każdą klasę należałoby wyposażyć w jej własny mechanizm obsługi błędów.

10.1. Model obsługi wyjątków w języku C++

Przeprowadzona wyżej krytyka tradycyjnych sposobów reakcji na wyjątki sugeruje, że optymalnym rozwiązaniem byłoby rozszerzenie składni języka o następujące konstrukcje:

- Przekazanie od procedury obsługi wyjątku do funkcji, wywołującej tę procedurę, informacji o rodzaju błędu oraz ewentualnych dodatkowych informacji.
- Generalizację wyjątków w postaci hierachii klas (np. wyjątki “błąd dziedziny” i “błąd zakresu” są szczególnymi przypadkami wyjątku “błąd matematyczny”).
- Dołączenie do funkcji wywołującej niezależnego kodu obsługi dla poszczególnych błędów.
- Automatyczne przekazanie sterowania do odpowiedniego fragmentu kodu obsługi błędu w przypadku zgłoszenia wyjątku.

Zastosowany w języku C++ mechanizm obsługi wyjątków spełnia powyższe postulaty. Zaakceptowany przez komitety ANSI X3J16/ISO WG-21 w roku 1990 stał się po raz pierwszy dostępny we wzorcowym kompilatorze AT&T wersji 3.0 we wrześniu 1991, a pierwsze implementacje przemysłowe firm DEC i IBM weszły na rynek na początku 1992. Dane te przytaczamy nie bez powodu: język, który zapewnia skuteczną obsługę wyjątków, może służyć do budowy systemów odpornych na błędy (ang. fault-tolerant systems), a więc ma szansę stać się standardem przemysłowym.
W języku C++ dla obsługi wyjątków zastosowano *model z terminacją*. Oznacza to, że procesy obsługi wyjątków przebiegają w sekwencji: zgłoszenie wyjątku przez funkcję – wyjście z jej bloku – przechwycenie przez procedurę obsługi – obsługa – zakończenie programu (lub przejście do następnej instrukcji w bloku funkcji zawierającej procedurę obsługi).
Nie jest to jedyne możliwe rozwiązanie: wielokrotnie w innych językach próbowano zastosować bardziej ogólny *model ze wznowieniem*. Jest to bardzo atrakcyjna alternatywa: zakłada ona, że

procedura obsługi wyjątku powinna być tak zaprojektowana, aby mogła żądać wznowienia programu od punktu, w którym został zgłoszony wyjątek. Model taki mógłby być szczególnie obiecujący dla unifikacji obsługi wyjątków na poziomie programu z obsługą wyjątków na poziomie systemu (wyczerpanie zasobów). Jednak wieloletnia praktyka pokazała, że model z terminacją jest prostszy, bardziej przejrzysty oraz tańszy prowadzi do łatwiejszego zarządzania systemami.

10.1.1. Deklaracje wyjątków

Mechanizm obsługi wyjątków języka C++ wprowadza trzy nowe słowa kluczowe. Pierwsze z nich, **try** oznacza blok kodu, w którym mogą wystąpić sytuacje wyjątkowe. Ich zgłoszenie następuje za pomocą instrukcji **throw**, a są one obsługiwane w blokach poprzedzonych słowem kluczowym **catch**. Bloki `catch`, które muszą występować bezpośrednio za blokiem `try`, mogą występować wielokrotnie. Ciąg bloków `catch`, występujących bezpośrednio za blokiem `try`, zawiera procedury obsługi wyjątków. Konstrukcję tę zapisuje się w postaci:

```
try{ }catch() { } ... catch() { }
```

Wyjątki mogą być zgłaszane *wyłącznie* wewnątrz bloku `try`, który, podobnie jak bloki instrukcji złożonych lub funkcji, może zawierać deklaracje, definicje i instrukcje.
Najprostsza składniowo instrukcja `throw` ma postać:

```
throw;
```

i oznacza ponowne zgłoszenie wyjątku aktualnie obsługiwanego w bloku `catch`. Wywołanie `throw` bez parametru w chwili gdy żaden wyjątek nie jest obsługiwany powoduje (domyślnie) zakończenie programu.
Instrukcja `throw` najczęściej występuje z parametrem:

```
throw wrn;
```

gdzie `wrn` może być dowolnym wyrażeniem traktowanym przez kompilator tak, jak wyrażenia będące argumentem wywołania funkcji lub instrukcji `return`, np. `throw 10; throw "abc"; throw obiekt;` przy czym `obiekt` jest wystąpieniem wcześniej zdefiniowanej klasy.
Typ obiektu będącego wynikiem obliczenia wyrażenia `wrn` określa rodzaj wyjątku, zaś sam obiekt jest przekazywany do tego bloku `catch`, który występuje za ostatnio napotkanym blokiem `try`. Jeżeli zgłoszony wyjątek nie jest obsługiwany przez daną procedurę w bloku `catch`, to jest on przekazywany do następnej. Jeżeli dla danego wyjątku nie została znaleziona procedura jego obsługi, to wykonanie programu zostanie zakończone. W procesie zakończenia programu wywoływana jest wówczas funkcja `terminate()`, która z kolei wywołuje funkcję `abort()`.

Przykład 10.1.

```
#include <iostream.h>
#include <excpt.h>
int  main() {
     char znak = '\0';
     while (znak != '*') {
     try
     {
      cout << "Znak '*'- koniec. "
           << "Podaj dowolny znak: ";
      cin >> znak;
      switch (znak) {
      case 'a': throw 1;
      case 'b': throw "tekst";
      case 'c': throw 2.0;
      default : throw 'x';
      } //Koniec switch
     } // Koniec try
```

```
      catch(int) { cout << "Przypadek 1\n"; }
      catch(char*)  { cout << "Przypadek 2\n"; }
      catch(double) { cout << "Przypadek 3\n"; }
      catch(...) {
      cout << "Wymagana kolejna procedura catch! ";
      return 1;
                } // Koniec catch(...)
      } // Koniec while
      return 0;
}
```

Przykładowy wydruk z programu ma postać:

Znak '*' - koniec. Podaj dowolny znak: a
Przypadek 1
Znak '*' - koniec. Podaj dowolny znak: c
Przypadek 3
Znak '*' - koniec. Podaj dowolny znak: *
Wymagana kolejna procedura catch!

Dyskusja. W powyższym programie wyjątki są zgłaszane w bloku `try`, umieszczonym w funkcji `main()`. Plik nagłówkowy `<excpt.h>` zawiera niezbędne deklaracje, pozwalające na dostęp do mechanizmu obsługi wyjątków. Procedury obsługi przechwytują wyjątki typu **int**, **char*** i **double**. Blok oznaczony `catch(...) {}` obsługuje wszystkie nieobsłużone wyjątki dowolnego typu. W sekwencji
`try{ }catch() { } ... catch() { }`
blok `catch(...) { }`, jeżeli występuje, musi być umieszczony jako ostatni.

•

Sekwencję `try-catch` można przenieść do oddzielnej funkcji, wywoływanej następnie z bloku funkcji `main()`, jak pokazano w kolejnym przykładzie.

Przykład 10.2.

```
#include <iostream.h>
#include <excpt.h>
void fun() {
 char znak = '\0';
 while (znak != '*') {
  try {
       cout << "Znak '*'- koniec. ";
       cout << "Podaj dowolny znak: ";
       cin >> znak;
       switch (znak) {
         case 'a': throw 1;
         case 'b': throw "tekst";
         case 'c': throw 2.0;
         default : throw 'x';
                   } //Koniec switch
      } // Koniec try

  catch(int)    { cout << "Przypadek 1\n"; }
  catch(char*)  { cout << "Przypadek 2\n"; }
  catch(double) { cout << "Przypadek 3\n"; }
```

```
  catch(...)
  { cout << "Wymagana kolejna procedura catch!\n"; }
 } // Koniec while
}// Koniec fun
int  main() {
     fun();
     return 0;
}
```

Jeżeli wprowadzimy tę samą sekwencję znaków co poprzednio, to otrzymamy identyczny obraz interakcji użytkownika z programem.

10.2. Wyjątek jako obiekt

W praktyce programy mogą zawierać wiele możliwych błędów w fazie wykonania. Błędy takie mogą być odwzorowane na wyjątki o rozróżnialnych nazwach. Ponadto wskazane jest, aby w przypadku wystąpienia błędu zgłoszony wyjątek zawierał maksimum informacji o przyczynie błędu. Jeżeli typ zgłaszanego instrukcją throw wyjątku jest typem wbudowanym, to możliwości są stosunkowo niewielkie. Rozwiązaniem jest zdefiniowanie klasy wyjątków z odpowiednim publicznym interfejsem i traktowanie wyjątku jako obiektu. Ilustruje to poniższy przykład.

Przykład 10.3.

```
#include <iostream.h>
class Liczba  {
public:
 class Zakres { };
 Liczba(int);
};
Liczba::Liczba(int i)
{  if (i > 10) throw Zakres();  }
int  main()  {
     int x;
     char znak;
     try  {
     cout << "Podaj liczbe typu int: ";
     cin >> x;
     Liczba num(x);
          } //Koniec try
     catch (Liczba::Zakres)
     {
      cout << endl << "Przechwycony wyjatek!\n";
     }; // Koniec catch
     cout << "Kontynuacja programu.\n"
          << "Wcisnij klawisz litery lub cyfry: ";
     cin >> znak;
     return 0;
}
```

Przykładowa interakcja z użytkownikiem:

Podaj liczbe typu int: 19

Przechwycony wyjatek!
Kontynuacja programu.
Wcisnij klawisz litery lub cyfry: a

Możliwość traktowania wyjątku jako obiektu typu zdefiniowanego przez użytkownika prowadzi do koncepcji hierarchii wyjątków, w której pewne wyjątki mogą być typami pochodnymi od wyjątków ogólniejszych. Np. dla biblioteki matematycznej można zdefiniować klasę bazową *BłądMat* i klasy od niej pochodne *Nadmiar*, *Niedomiar*, *DzielZero*. W takich przypadkach istotna jest kolejność, w jakiej występują bloki `catch`. Wiadomo, że próby przechwycenia wyjątków zgłaszanych z bloku `try` odbywają się w takiej kolejności, w jakiej występują kolejne bloki `catch`. Zatem procedurę obsługi dla klasy bazowej należy umieszczać jako ostatnią (albo przedostatnią, jeżeli występuje `catch(...){}`); w przeciwnym przypadku procedura dla klasy pochodnej nie zostałaby nigdy wywołana. Zwróćmy jeszcze uwagę na następujący moment. Jeżeli weźmiemy ciąg deklaracji:

```
class WyjOgólny {
public:
 virtual void ff() { /* instrukcje */ }
};
class WyjSzczególny: public WyjOgolny {
public:
  void ff() { /* instrukcje */ }
};
void funkcja()
{
 try
  {
   // Wywołanie funkcji, która zgłasza
   // wyjątek typu WyjSzczególny
  }
 catch(WyjOgólny wo) { wo.ff(); }
}
```

to w tym przypadku zostanie wykonana funkcja `WyjOgólny::ff()`, pomimo że zgłoszony wyjątek był typu `WyjSzczególny`, a funkcja `ff()` jest funkcją wirtualną. Wynika to stąd, że obiekt typu `WyjSzczególny` jest przekazywany przez wartość (za pomocą konstruktora kopiującego klasy `WyjSzczególny`) jako parametr aktualny procedury `catch()`. Ponieważ parametrem formalnym jest obiekt typu `WyjOgólny`, to obiekt przesłany z bloku `try` zostanie "obcięty na wymiar `wo`". W obiekcie `wo` będzie więc dostępny jedynie wskaźnik do funkcji wirtualnej `ff()` klasy `WyjOgólny`. Można temu zapobiec, stosując wskaźniki lub referencje, np.

```
catch(WyjOgólny& wo) { wo.ff(); }
```

10.3. Sygnalizacja wyjątków w deklaracji funkcji

Konstrukcje `throw-try-catch` zwykle występują w bloku oddzielnej funkcji, wywoływanej w ciele innej funkcji. Interakcję takiej funkcji z innymi funkcjami można uczynić bardziej czytelną, podając jawnie w jej nagłówku możliwe do zgłoszenia wyjątki, np.

```
void ff(int i) throw(A, B);
```

Powyższa deklaracja mówi, że funkcja `ff` może zgłosić wyjątki tylko dwóch podanych typów. Taki sposób deklarowania stosuje się również, gdy podajemy definicję funkcji, a nie tylko jej prototyp. W obu przypadkach w bloku funkcji mogą (ale nie muszą) wystąpić odpowiednie instrukcje `throw` lub wywołania funkcji generujących wyjątki z bloku `try`.
Gdyby z bloku tej funkcji został zgłoszony wyjątek różny od `A` lub `B`, to funkcja nie będzie w stanie obsłużyć wyjątku samodzielnie, ani też przekazać go do funkcji wołającej. Po wystąpieniu takiego nieoczekiwanego wyjątku zostanie automatycznie wywołana funkcja `void unexpected()`. Funkcja ta wywołuje opisaną uprzednio funkcję `terminate()`, która z kolei wywołuje `abort()` i kończy program. Jednak wywołaniem domyślnym dla funkcji `unexpected()` jest wywołanie funkcji, zdefiniowanej przez użytkownika, a "rejestrowanej" jako argument funkcji

`set_unexpected()`. Funkcja ta jest wprowadzona w pliku nagłówkowym `<except.h>` deklaracjami:

```
typedef void (*PFV)();
PFV set_unexpected(PFV);
```

Stwarza ona użytkownikowi pewną możliwość wpływania na obsługę wyjątku nieoczekiwanego. Jak widać z deklaracji, wskaźnik `PFV` do bezargumentowej funkcji typu **void** jest typem zwracanym przez funkcję `set_unexpected()`, tj. typem funkcji, która była parametrem aktualnym w ostatnim wywołaniu funkcji `set_unexpected()`.

W deklaracji (definicji) funkcji można jej "zakazać" zgłaszania wyjątków, dodając w jej nagłówku `throw()`, np.

```
void ff(int i) throw();
```

Jeżeli, mimo zakazu, powyższa funkcja zgłosi wyjątek, to musi on zostać przechwycony i obsłużony w jej bloku. W przeciwnym przypadku zostanie wywołana funkcja `unexpected()` i dalszy bieg zdarzeń będzie analogiczny, jak w poprzednim przypadku.

Przykład 10.4.

```
#include <iostream.h>
#include <excpt.h>
class Nowa { };

Nowa obiekt;
void f3(void) throw (Nowa)
{
 cout << "Wywolana f3()" << endl;
 throw(obiekt);
}

void f2(void) throw()
{
 try {
      cout << "Wywolana f2()" << endl;
      f3();
     }
 catch ( ... )
    {
     cout << "Przechwycony wyjatek w f2()!" << endl;
    }
 }
int   main() {
      try   {
      f2();
      return 0;
            }
      catch ( ... )    {
      cout << "Potrzebna kolejna procedura catch! ";
      return 1;
                       }
}
```

Wydruk z programu będzie miał postać:

Wywolana f2()
Wywolana f3()
Przechwycony wyjatek w f2()!

Dyskusja. W przykładzie pokazano wpływ specyfikacji wyjątków w nagłówku funkcji na działanie programu. Zdefiniowano w nim klasę wyjątków `Nowa` i jej wystąpienie o nazwie `obiekt`. Prototyp funkcji `f3()`

```
void f3(void) throw (Nowa);
```

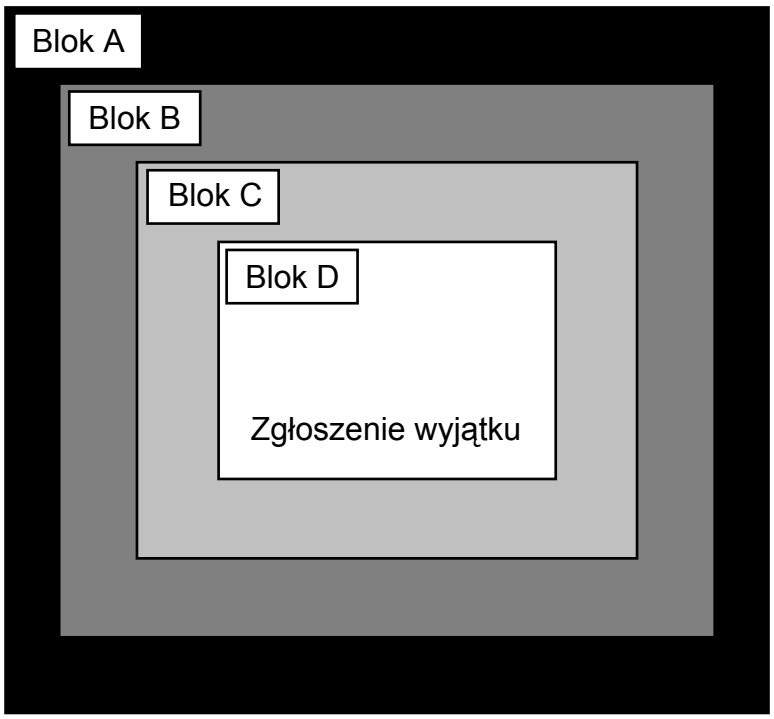
mówi, że jedynymi wyjątkami, które może ona zgłaszać, są obiekty klasy `Nowa`. Natomiast funkcja `f2()`, co wynika z postaci jej prototypu

```
void f2(void) throw();
```

nie powinna zgłaszać żadnych wyjątków. Jednak z jej bloku jest wywoływana funkcja `f3()`, która może i zgłasza wyjątek. Tak więc wykonanie programu po wywołaniu `f2()` z bloku `main()` nie kończy się wykonaniem instrukcji `return 0;` lecz `return 1;` po przechwyceniu wyjątku zgłoszonego z bloku funkcji `f3()`.

10.4. Propagacja wyjątków

Funkcje, wywoływane w bloku `try`, mogą również zawierać bloki `try`; pozwala to tworzyć hierarchie obsługi wyjątków.
Jeżeli funkcja zgłaszająca wyjątek jest wywoływana z bloku innej, nadrzędnej funkcji, to proces obsługi wyjątku może przebiegać w sposób, zilustrowany rysunkiem 10-1. Schemat wywołań jest tutaj następujący: z bloku funkcji `A` została wywołana funkcja `B`, z jej bloku została wywołana funkcja `C`, a z jej bloku funkcja `D`, która zgłosiła wyjątek.



Rys. 10-1 Obsługa wyjątku przy zagnieżdżonych wywołaniach funkcji

W chwili zgłoszenia wyjątku zamykany jest blok funkcji `D`, to znaczy usuwany jest ze stosu jego rekord aktywacyjny i usuwane są wszystkie zmienne lokalne (automatyczne) utworzone w tym bloku. Jeżeli w bloku `D` istnieje odpowiednia procedura obsługi zgłoszonego wyjątku, to sterowanie zostanie przekazane do tej procedury. Załóżmy, że tak nie jest, i że odpowiedni blok `catch` znajduje się w

funkcji nadrzędnej `B`. Wobec tego, po zakończeniu bloku `D`, zostaną zakończone w taki sam sposób bloki `C` i `B`, po czym sterowanie zostanie przekazane do procedury obsługi wyjątku z bloku `B`. Po zakończeniu obsługi zostanie wznowione wykonanie bloku funkcji `A` od następnej po wywołaniu funkcji `B` instrukcji.
Gdyby w żadnym bloku z łańcucha wywołań nie został znaleziony odpowiedni blok `catch`, to zostałyby zakończone wszystkie bloki i sterowanie zostałoby przekazane do funkcji `terminate()`. Standardowo funkcja ta powoduje zakończenie programu. Dokładniej mówiąc, funkcja `void terminate()` wykonuje ostatnią funkcję, przekazaną jako parametr aktualny (wskaźnik) do funkcji `set_terminate()`, wprowadzonej deklaracjami:

```
typedef void (*PFV) ();
PFV set_terminate(PFV);
```

Jak widać z deklaracji, wskaźnik `PFV` do bezargumentowej funkcji typu **void** jest i argumentem, i typem zwracanym przez funkcję `set_terminate()`.

•

Podany niżej przykład ilustruje opisany mechanizm.

Przykład 10.5.

```
//Propagacja wyjatkow
#include <iostream.h>
class Nowa { };//Deklaracja wyjatku
Nowa obiekt;
void B() throw();
void C() throw(Nowa); void D() throw (Nowa);
void A() throw() {
try { cout << "Blok try funkcji A()\n";
B(); }
catch(...) { cout << "catch() w A()"; }
cout << "Kontynuacja A()\n";
}
void B() throw() {
try  { cout << "Blok try funkcji B()\n";
       C();
     }
catch(Nowa)
{ cout << "Przechwycony wyjatek z D()!\n"; }
cout << "Zamykany blok B()\n";
}
void C() throw(Nowa) {
try  {
      cout << "Blok try funkcji C()\n";
      D();
     }
catch(int) { cout << "catch w C()\n"; throw; }
cout << "Kontynuacja C()\n";
}

void D() throw (Nowa) {
try {
     cout << "Blok try funkcji D()\n";
     throw(obiekt);
    }
```

```
catch(int) { cout << "catch w D()\n"; }
cout << "Kontynuacja D()\n";
}

int  main() {
     try  {
     cout << "Wywolana A()\n";
     A();
     cout << "Po A()\n";
     return 0;
          }
     catch(...) { cout<<"Potrzebny kolejny blok catch\n"; }
     cout << "Kontynuacja main()\n";
     return 0;
}
```

Wydruk z programu ma postać:

Blok try funkcji A()
Blok try funkcji B()
Blok try funkcji C()
Blok try funkcji D()
Przechwycony wyjatek z D()!
Zamykany blok B()
Kontynuacja A()
Po A()

10.5. Wyjątki i zasoby systemowe

Możliwość wystąpienia wyjątków wymaga starannej uwagi programisty, ponieważ burzy ona liniowy przebieg wykonania programu. Jeżeli np. funkcja rezerwuje pewne zasoby (otwiera plik, przydziela pamięć z kopca, itp.), to powinna je w odpowiedni sposób zwolnić, gdyż w przeciwnym przypadku może to spowodować nieoczekiwany przebieg wykonania programu. Zazwyczaj funkcja zwalnia zasoby przy wyjściu ze swojego bloku, tuż przed przekazaniem sterowania do funkcji wołającej. Jednakże odnosi się to jedynie do zmiennych lokalnych; jeżeli funkcja operuje na zmiennych globalnych, to nie mamy takiej gwarancji. Weźmy dla przykładu sekwencję instrukcji:

```
ifs.open("we.doc");
fun(ifs);
ifs.close();
```

Jeżeli `ifs` jest zmienną globalną (obiektem) klasy `ifstream`, to funkcja `fun` (lub funkcja przez nią wywoływana) może zgłosić wyjątek i instrukcja `ifs.close();` nie zostanie nigdy wykonana!
Opanowanie takiej sytuacji jest technicznie możliwe przez przechwycenie dowolnego z możliwych wyjątków, zamknięcie pliku i ponowne zgłoszenie przechwyconego wyjątku:

```
ifs.open("we.doc");
try {
 fun(ifs);  }
catch(...)  {
 ifs.close();
 throw;     }
ifs.close();
```

Jednak stosowanie takiej strategii byłoby nadzwyczaj kłopotliwe. Zamiast takiego podejścia należy wykorzystać fakt, że w C++ przy wyjściu z funkcji następuje automatyczne wywołanie destruktorów

dla wszystkich obiektów lokalnych. Dotyczy to również tych obiektów, dla których zostały zarezerwowane zasoby w funkcjach, wywoływanych przez funkcję `fun`. W pokazanym wyżej przypadku wystarczy zadeklarować `ifs` jako obiekt lokalny klasy `ifstream`:

```
ifstream ifs("we.doc");
fun(ifs);
```

ponieważ destruktor klasy bibliotecznej `ifstream` automatycznie zamknie plik `we.doc` w momencie, gdy wykonanie dojdzie do końca otaczającego bloku, lub gdy wyjątek jest obsługiwany w bloku zewnętrznym.
Powyższe uwagi odnoszą się w równym stopniu do alokacji pamięci.

Przykład 10.6.

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

class GetMemory {
public:
  int* wskmem;
  GetMemory(int m) { wskmem = new int[m];}
  ~GetMemory()     { delete[] wskmem;}
};

class MojaKlasa {
 public:
  class Rozmiar { };
  MojaKlasa(const char* filename, int sizemem);
};
MojaKlasa::MojaKlasa(const char* filename, int sizemem)
{
  ofstream os(filename);
  if (sizemem < 0 || 30 < sizemem) throw Rozmiar();
  GetMemory obiekt1(sizemem);
  cout << "Przydzielona zadana pamiec "
       <<  sizemem << " bajtow "
       << "i otwarty plik\n";
}

int  main() {
     int x;
     try
     {
      cout << "Podaj rozmiar pamieci w bajtach: ";
      cin >> x;
      MojaKlasa obiekt2("zasob.txt",x);
     }
     catch ( MojaKlasa::Rozmiar )
     {
      cout << " Niepoprawny rozmiar zadanej pamieci \n";
     }
     return 0;
}
```

Dyskusja. Dwukrotne uruchomienie programu może dać następujące wydruki:

Podaj rozmiar pamieci w bajtach: 25

Przydzielona pamiec 25 bajtow i otwarty plik

Podaj rozmiar pamieci w bajtach: -100
Niepoprawny rozmiar zadanej pamieci

W przykładzie pokazano przechwytywanie błędów powstających w konstruktorze obiektu klasy `MojaKlasa`. Jeżeli z klawiatury podamy rozmiar alokowanej pamięci w granicach od 0 do 30, to wykonanie programu przebiega liniowo: program przydzieli zadaną pamięć i utworzy (otworzy i zamknie) plik `zasob.txt` o zerowej zawartości. W drugim przypadku mamy następującą sekwencję czynności:

- utworzenie obiektu `os` i otwarcie pliku `zasob.txt`
- zgłoszenie wyjątku typu `MojaKlasa::Rozmiar`
- zamknięcie pliku `zasob.txt` przez niejawnie wywołany destruktor klasy `ofstream` (destrukcja obiektu `os`)
- przechwycenie wyjątku przez blok `catch`
- obsługę wyjątku
- zakończenie programu.

10.6. Nadużywanie wyjątków

Wyjątki powinny być wyjątkowe. To oczywiste stwierdzenie nie zawsze jest respektowane: programiści dość często ulegają pokusie wykorzystania mechanizmu wyjątków do przekazywania sterowania z jednego punktu programu do innego. Takie postępowanie może w najlepszym razie świadczyć o złym stylu programowania. Wyjątki powinny być zarezerwowane dla takich przypadków, które nie mogą się zdarzyć w normalnym przebiegu obliczeń i których wystąpienie tworzy sytuację, z której nie ma wyjścia w aktualnym zasięgu. Dobrym przykładem konieczności użycia mechanizmu wyjątków jest wyczerpanie się pamięci. Nikt nie może z góry przewidzieć kiedy zabraknie pamięci, a w punkcie detekcji tego faktu rzadko jest możliwe zrobić coś więcej.
Przeciwieństwem dla tego przypadku może być wykrycie końca pliku, z którego właśnie czytamy dane. Wiadomo, że w każdym pliku dojdziemy do jego końca, a zatem kod dla czytania z pliku musi być na to przygotowany. To samo dotyczy pobierania danych z kolejki: przed każdą próbą usunięcia elementu należy się upewnić, czy kolejka nie jest pusta.
Podany niżej przykład ilustruje nadużywanie mechanizmu wyjątków do przkazywania sterowania w sytuacjach, w których całkowicie wystarczające byłoby warunkowe wywoływanie funkcji.

Przykład 10.7.

```
#include <iostream.h>
#include <string.h>
void wykonanie1()
{ cout << "Wykonanie a\n"; }
void wykonanie2()
{ cout << "Wykonanie b\n"; }
int  main() {
     char rozkaz[80];
     cout << "Napisz znak lub sekwencje znakow.\n";
     cout << "Zacznij od znakow 'a' i' 'b': ";
     while(cin >> rozkaz) {
     try
     {
      if (strcmp(rozkaz, "a") == 0) wykonanie1();
      else if(strcmp(rozkaz, "b") == 0) wykonanie2();
      else cout << "Nieznany rozkaz: "
                << rozkaz << endl;
     } // Koniec try
```

```
      catch (char* komunikat)
      {
       cout << komunikat << endl;
      } // Koniec catch
      catch(...) { cout << "Koniec\n"; }
      } // Koniec while
      return 0;
}
```

Wydruk dla przykładowego wykonania programu:

Napisz znak lub sekwencje znakow.
Zacznij od znakow 'a' i 'b': a
Wykonanie a
b
Wykonanie b
abc
Nieznany rozkaz: abc

11. Dynamiczna i statyczna kontrola typów

Dynamiczna kontrola typów, znana pod akronimem RTTI (ang. Run-Time Type Information), została wprowadzona do standardu języka w roku 1993. W skład mechanizmu RTTI wchodzą:

- Operator **dynamic_cast** (słowo kluczowe), który służy do otrzymania wskaźnika do obiektu klasy pochodnej przy danym wskaźniku do klasy bazowej tego obiektu. Operator **dynamic_cast** daje ten wskaźnik jedynie wtedy, gdy wskazywany obiekt jest rzeczywiście wystąpieniem podanej klasy pochodnej; w przeciwnym przypadku przekazuje 0.
- Operator **typeid** (słowo kluczowe), który pozwala zidentyfikować dokładny typ obiektu na podstawie wskaźnika do jego klasy bazowej.
- Klasę biblioteczną `type_info`, dostarczającą dalszych informacji o typie dla fazy wykonania programu. Deklaracja tej klasy znajduje się w pliku nagłówkowym `<typeinfo.h>`.

Mechanizm RTTI uzupełniają trzy dalsze operatory: **static_cast**, **const_cast** i **reinterpret_cast**, służące do konwersji statycznej.

Wprowadzenie dynamicznej kontroli typów wynikło z naturalnej potrzeby. Podstawowym sposobem kontroli typów jest w języku C++ kontrola statyczna (“silna”), wykonywana w fazie kompilacji. Jest to mechanizm bardzo efektywny, ponieważ nie wprowadza żadnych narzutów czasowych w fazie wykonania. Np. kontrola statyczna wywołania funkcji składowej klasy obejmuje jej pełny typ: typy argumentów i typ zwracany. Silna kontrola typów ma również miejsce w przypadku funkcji przeciążonych i funkcji z argumentami domyślnymi. Jedynym odstępstwem (nie zalecanym do stosowania) jest możliwość deklarowania funkcji z wielokropkiem ('...') podanym zamiast typu argumentu.
Silna typizacja statyczna jest korzystna z wielu względów. Jeżeli tworzymy obiekt konkretnego typu, np. `double`, `char*`, czy `Test`, to próba użycia tego obiektu w sposób niezgodny z jego typem oznacza naruszenie systemu typów. Język, w którym takie naruszenie nie może się nigdy zdarzyć, jest językiem o typizacji silnej. Przykładem takiego języka jest Pascal.
Silna typizacja nie mogła być wbudowana w język C++ ze względu na jego cechy, odziedziczone z języka C. Konstrukcje programowe takie jak unie, konwersje i tablice nie pozwalają na wykrycie każdego naruszenia systemu typów w fazie kompilacji. Postąpiono wobec tego inaczej.
Każde jawne naruszenie systemu typów generuje komunikat o błędzie i powoduje zaniechanie kompilacji.
Każde niejawne naruszenie (lub nawet podejrzenie o naruszenie) systemu typów powoduje wysłanie ostrzeżenia przez kompilator.
Po przejściu przez fazę kompilacji wykonywana jest następna kontrola typów w fazie konsolidacji (łączenia). Np. dla wywołań funkcji oznacza to, że program przejdzie konsolidację tylko wtedy, gdy każda wywołana funkcja ma swoją definicję i typy argumentów podane w jej deklaracji takie same (lub zgodne) jak typy podane w definicji. Jest to szczególnie ważne w programach wieloplikowych, gdy konsolidator musi sprawdzić na zgodność typy funkcji we wszystkich jednostkach kompilacji (ang. type-safe linkage).
Zamiast wykorzystywać niepewne cechy pochodzące z języka C, proponuje się użytkownikowi korzystanie z cech, podlegających ścisłej kontroli typów. Przykładami mogą być klasy pochodne, bezpieczne tablice, etc.

Mimo iż typizacja statyczna z towarzyszącą jej bezpieczną konsolidacją jest bardzo efektywna, pozbawia ona język giętkości, właściwej językom z typizacją dynamiczną, takim jak np. Smalltalk i Eiffel. W językach tych wiązanie obiektu z konkretnym typem i kontrola typów są odkładane do fazy wykonania. Pozwala to m.in. na zmianę typu obiektu w różnych momentach fazy wykonania. W języku C++ nie dopuszcza się takich “przełączeń”. Nawet w przypadku klas z funkcjami wirtualnymi kompilator i konsolidator gwarantują jednoznaczną odpowiedniość pomiędzy obiektami, a wywołanymi dla nich funkcjami, generując dla każdej takiej klasy tablicę funkcji wirtualnych.

Klasy z funkcjami wirtualnymi są często nazywane *klasami polimorficznymi*. Są to jedyne klasy, które pozwalają bezpiecznie operować na swoich obiektach za pomocą wskaźników do ich klasy bazowej. Słowo "bezpiecznie" jest rozumiane jako gwarancja ze strony języka, że obiekty mogą być używane jedynie zgodnie ze swoim zdefiniowanym typem.
Jednak nawet klasy polimorficzne mają pewne ułomności. Wiadomo przecież, że przypisując wskaźnikowi do klasy bazowej adres obiektu klasy pochodnej możemy operować tylko tymi składowymi obiektu klasy pochodnej, które odziedziczył z klasy bazowej. Wiadomo także, że nie jest dopuszczalna jawna konwersja wskaźnika do klasy bazowej na wskaźnik do klasy pochodnej.
Mechanizm RTTI wychodzi naprzeciw tym problemom, pozwalając wykonywać jawną kontrolę i konwersję typów w fazie wykonania programu.

11.1. Konwersja dynamiczna

Zwykłe konwersje są jednym z głównych źródeł błędów w języku C++. Ponadto mają one dość zagmatwaną składnię i w wielu przypadkach nie są bezpieczne; konwersja jest operacją na typach danych i na ogół nie zależy od wartości obiektów, na których operuje. Dlatego zwykła konwersja nie może się nie udać – po prostu wyprodukuje nową wartość.
Tę niekorzystną sytuację w znacznym stopniu zmieniło na lepsze wprowadzenie dwóch nowych operatorów: `dynamic_cast` i `typeid`. Pierwszy z nich można stosować tylko do klas z funkcjami wirtualnymi, które łatwo mogą dostarczyć informacji o swoim typie w fazie wykonania programu. Składnia tego operatora ma postać:
`dynamic_cast<T>(wsk)`
W wyrażeniu `dynamic_cast<T>(wsk)` `T` musi być wskaźnikiem lub referencją do wcześniej zdefiniowanej klasy lub `void*`. Argument `wsk` musi być wskaźnikiem lub referencją.
Jeżeli `T` jest typu `void*`, wówczas `wsk` musi także być wskaźnikiem. W tym przypadku konwersja daje wskaźnik, który może mieć dostęp do dowolnego elementu klasy leżącej najniżej w hierarchii klas.
Konwersja z klasy pochodnej do bazowej jest wiązana statycznie (w fazie kompilacji/konsolidacji). Jeżeli `T` jest wskaźnikiem i `wsk` jest wskaźnikiem do klasy pochodnej, to wynik konwersji jest wskaźnikiem do klasy pochodnej. Przy takim założeniu można dokonywać konwersji z klasy pochodnej do bazowej i z danej klasy pochodnej do innej klasy pochodnej. Analogiczna relacja zachodzi dla referencji, gdy `T` i `wsk` są referencjami.
Konwersja z klasy bazowej do pochodnej jest możliwa jedynie dla klas polimorficznych. W tym przypadku mamy wiązanie dynamiczne (w fazie wykonania).
Udana konwersja dynamiczna przekształca `wsk` do żądanego typu. Jeżeli `T` i `wsk` są wskaźnikami, to nieudana konwersja zwraca wskaźnik o wartości 0; w przypadku referencji niepowodzenie konwersji zgłasza wyjątek `Bad_cast` (klasa `Bad_cast` jest zdefiniowana w pliku nagłówkowym `<typeinfo.h>`).

W podanym niżej przykładzie klasa `Bazowa` zawiera wirtualną funkcję składową, a więc jest klasą polimorficzną. W prezentowanym programie konwersja wskaźnika do klasy bazowej we wskaźnik do klasy pochodnej jest bezpieczna; oznacza to, że po konwersji możemy bezpiecznie (w sensie typizacji) operować na elementach obiektów klasy pochodnej.

Przykład 11.1.

```
#include <iostream.h>
class Bazowa {
public:
 int x;
 virtual void podaj() { cout<<"Bazowa::podaj()\n"; }
};

class Pochodna: public Bazowa {
public:
```

```
  int y;
  void podaj() { cout << "Pochodna::podaj()\n"; }
};
int   main() {
      Pochodna po, *wskp;
      Bazowa* wskb = &po;
      if((wskp = dynamic_cast<Pochodna*>(wskb)) != 0)
      cout << "Konwersja udana\n";
      wskp->x = 10;
      wskp->y = 20;
      cout << "wskp->x = " << wskp->x << endl;
      cout << "wskp->y = " << wskp->y << endl;
      cout << "wskb->x = " << wskb->x << endl;
      wskp->podaj();
      wskp->Bazowa::podaj();
      return 0;
}
```

Wydruk z programu ma postać:

Konwersja udana
wskp->x = 10
wskp->y = 20
wskb->x = 10
Pochodna::podaj()
Bazowa::podaj()

Przykład 11.2.

```
#include <iostream.h>
class Bazwirt1 {
public:
  Bazwirt1() { }
  virtual void f() { cout << "Bazwirt1::f()\n"; }
};
class Bazwirt2 {
public:
   Bazwirt2() { }
   virtual void g() { cout << "Bazwirt2::g()\n"; }
};
class Bazowa3 {};

class Pochodna: public Bazwirt1, public virtual Bazwirt2, public
virtual Bazowa3
{};
void g(Pochodna& po)
{
 Bazwirt1* wskb1 = &po;
 wskb1->f();
 Pochodna* wskp1 = (Pochodna*)wskb1;
 wskp1->g();
 Pochodna* wskp2 = dynamic_cast<Pochodna*>(wskb1);
 Bazwirt2* wskw = &po;
//Nie ma konwersji z wirtualnej bazy:
// Pochodna* wskp3 = (Pochodna*)wskw;
 Pochodna* wskp4 = dynamic_cast<Pochodna*>(wskw);
 Bazowa3* wskb3 = &po;
```

```
//Nie ma konwersji z wirtualnej bazy:
// Pochodna* wskp5 = (Pochodna*)wskb3;
//Nie ma konwersji z klasy nie-polimorficznej:
// Pochodna* wsk6 = dynamic_cast<Pochodna*>(wskb3);
}
int  main() {
     Pochodna pdn;
     g(pdn);
     return 0;
}
```

Dyskusja. Wydruk z programu ma postać;

Bazwirt1::f()
Bazwirt2::g()

W powyższym przykładzie klasa `Pochodna` dziedziczy od dwóch klas polimorficznych `Bazwirt1` i `Bazwirt2` oraz od klasy `Bazowa3`, przy czym ostatnie dwie są dla niej wirtualnymi klasami bazowymi. W bloku funkcji `g()`, której argumentem jest referencja do klasy `Pochodna`, zadeklarowano szereg konwersji, przy czym zapisy niedopuszczalne składniowo są potraktowane jako komentarze. Zwróćmy uwagę na następujące:
`Pochodna* wskp1 = (Pochodna*)wskb1;`
jest konwersją dopuszczalną, ale bez gwarancji powodzenia operacji na obiekcie `*wskp1`. Dwie konwersje dynamiczne:
`Pochodna* wskp2 = dynamic_cast<Pochodna*>(wskb1);`
`Pochodna* wskp4 = dynamic_cast<Pochodna*>(wskw);`
są bezpieczne, ponieważ będą sygnalizowały niepowodzenie, a ponadto można sprawdzić ich typy wynikowe.

11.2. Dynamiczna identyfikacja typów

Drugą główną cechą RTTI jest możliwość wyznaczenia dokładnego typu obiektu. Do identyfikacji typu obiektu służy wbudowany operator `typeid()`.
Gdyby operator `typeid()` był funkcją, jego deklaracja wyglądałaby jak niżej:

```
class type_info;
const type_info& typeid(nazwa-typu);
const type_info& typeid(wyrażenie);
```

Operator `typeid()` można używać zarówno do typów wbudowanych, jak i typów definiowanych przez użytkownika. Zwraca on referencję do nieznanego typu nazwanego `type_info`.
Jeżeli jego operandem jest `nazwa-typu`, to zwraca on referencję do klasy `type_info`, która reprezentuje argument `nazwa-typu`.

Jeżeli operandem jest `wyrażenie`, to `typeid()` zwraca referencję do klasy `type_info`, która wtedy reprezentuje typ obiektu oznaczonego przez `wyrażenie`.
Jeżeli operandem jest referencja lub poprzedzony gwiazdką wskaźnik do klasy polimorficznej, to `typeid()` zwraca typ dynamiczny aktualnego obiektu tej klasy. Jeżeli operand nie jest polimorficzny, `typeid()` zwraca obiekt, który reprezentuje typ statyczny.
Jeżeli `wsk` jest wskaźnikiem do klasy, a w wyrażeniu `typeid(*wsk)` wartość `wsk==0`, wówczas zgłaszany jest wyjątek `Bad_typeid` (klasa `Bad_typeid` jest zdefiniowana w pliku nagłówkowym `<typeinfo.h>`).
Korzystanie z operatora `typeid()` wymaga włączenia do programu pliku nagłówkowego `<typeinfo.h>`.

Przykład 11.3.

```
// Identyfikacja typu obiektu
// dla klasy nie-polimorficznej
#include <iostream.h>
#include <typeinfo.h>
class Info { };
int  main() {
     Info* wsk = new Info;
     cout << "Typ klasy Info jest: "
          << typeid(Info).name() << endl;
     cout << "Typ *wsk jest: "
          << typeid(*wsk).name() << endl;
     if(typeid(Info) == typeid(*wsk))
     cout << "Ten sam typ 'Info' i '*wsk'.\n";
     else cout << "NIE te same typy 'Info' i '*wsk'\n";
     return 0;
}
```

Dyskusja. Wydruk z programu ma postać:

Typ klasy 'Info' jest: Info
Typ *wsk jest: Info
Ten sam typ 'Info' i '*wsk'

W programie wykorzystano funkcję składową `name()`, klasy `type_info` zadeklarowaną w standardowym pliku nagłówkowym `<typeinfo.h>`. Funkcja ta zwraca nazwę typu, a jej prototyp ma postać:

```
const char* name() const;
```

Przykład 11.4.

```
//Identyfikacja typu obiektu dla typow
//wbudowanych i klas nie-polimorficznych
#include <iostream.h>
#include <typeinfo.h>
class Bazowa { };
class Pochodna: public Bazowa { };
char* wsk1 = "True";
char* wsk2 = "False";

int  main() {
     char znak;
     float x;
     if(typeid(znak) == typeid(x))
     cout << "Ten sam typ 'znak' i 'x'.\n";
     else cout << "NIE te same typy 'znak' i 'x'\n";
     cout << typeid(int).name() << endl;
     cout << typeid(Bazowa).name();
     cout << " przed " << typeid(Pochodna).name() << ":"
          << (typeid(Bazowa).before(typeid(Pochodna))
     ? wsk1 : wsk2) << endl;
     return 0;
}
```

Dyskusja. Wydruk z programu ma postać:

NIE te same typy 'znak' i 'x'
int
Bazowa przed Pochodna: True

W programie wykorzystano funkcję składową `before()`, klasy `type_info` zadeklarowaną w standardowym pliku nagłówkowym `<typeinfo.h>`. Funkcja ta zwraca wartość typu **int**, a jej prototyp ma postać:

```
int before(const type_info&) const;
```

Funkcja `type_info::before()` pozwala porządkować obiekty klasy `type_info`. Należy zaznaczyć, że relacja porządku, wprowadzana przez tę funkcję, nie ma nic wspólnego z uporządkowaniem w drzewie czy w grafie dziedziczenia. Nie ma również gwarancji, że `before()` da te same wyniki dla różnych programów, czy też dla kolejnych wykonań tego samego programu.

Przykład 11.5.

```
//Polimorficzna klasa bazowa
#include <iostream.h>
#include <typeinfo.h>
class Bazowa {
 virtual void func() {};
};
class Pochodna: public Bazowa {};
int  main() {
      Pochodna obiekt;
      Pochodna* wskp;
      wskp = &obiekt;
      try { //Testy prowadzone w fazie wykonania
      if (typeid(*wskp) == typeid(Pochodna))
//Pytanie: jaki jest typ *wskp?
      cout << "Nazwa typu jest "
            << typeid(*wskp).name();
      if (typeid(wskp) != typeid(Bazowa))
      cout << "\nWskaz nie jest typu Bazowa. ";
      return 0;
      } //Koniec try
      catch (Bad_typeid) {
      cout << "Nieudana identyfikacja typeid().";
      return 1;
      } // Koniec catch
}
```

Wydruk z programu ma postać:

Nazwa typu jest Pochodna
Wskaz nie jest typu Bazowa.

11.3. Nowa notacja dla konwersji

Konwersja dynamiczna za pomocą operatora `dynamic_cast` jest stosowalna w omawianych wcześniej specyficznych przypadkach, w szczególności dla konwersji z polimorficznej klasy bazowej do jej klasy pochodnej. Trzy dalsze operatory: `static_cast`, `const_cast` i `reinterpret_cast` wprowadzono dla konwersji statycznych z dwóch powodów. Po pierwsze, nowa składnia zamiast stosowania notacji (typ)wyrażenie, która może niejednokrotnie sugerować, że chodzi o jakąś funkcję, wyraźnie pokazuje, że mamy do czynienia z konwersją. Po drugie, sama

operacja konwersji, korzystająca z nowych operatorów, jest bezpieczniejsza. Tym niemniej użytkownikowi pozostawiono możliwość korzystania ze starej notacji, wraz z czychającymi w niej pułapkami. Można się o tym przekonać nie tylko w przypadku konwersji, co ilustruje podany niżej przykład.

Przykład 11.6.

```
#include <iostream.h>
const int zmienna = 10;
int   main() {
      cout << "zmienna: " << zmienna << endl;
      int& z = zmienna;
      z = 20;
      cout << "z: " << z << endl;
      return 0;
}
```

Dyskusja. Program daje się skompilować i wykonać, chociaż kompilator wyśle najpierw ostrzeżenie: "Temporary used to initialize 'z' in function 'main'". Wbrew oczekiwaniom program wydrukuje:

zmienna: 10
z: 20

•

Wróćmy jednak do konwersji. Operator `static_cast` ma składnię:

```
static_cast<T>(arg)
```

gdzie `T` musi być wskaźnikiem, referencją, typem arytmetycznym, lub typem wyliczeniowym, zaś typ argumentu `arg` musi być zgodny z typem `T` i w pełni znany w fazie kompilacji. Konwersja statyczna może być w szczególności stosowana do przekształcenia wskaźnika do klasy bazowej we wskaźnik do klasy pochodnej i odwrotnie. W pierwszym przypadku wymaga się, aby klasa bazowa nie była klasą wirtualną. W drugim przypadku musi istnieć jednoznaczna konwersja z klasy pochodnej do bazowej. Podany niżej przykład ilustruje konwersje w obu kierunkach oraz przypomina stary styl konwersji.

Przykład 11.7.

```
#include <iostream.h>
class Bazowa { };
class Pochodna: public Bazowa { };
int   main() {
      Bazowa* wskb = new Bazowa;
      Pochodna* wskp = new Pochodna;
      Pochodna* wskp1 = (Pochodna*)wskb;//stary styl
      Pochodna* wskp2 = static_cast<Pochodna*>(wskb);
      Bazowa* wskb1 = static_cast<Bazowa*>(wskp);
      return 0;
}
```

Operator `const_cast`, podobnie jak pozostałe operatory konwersji, został pomyślany jako mechanizm, który respektuje stałość zdefiniowanego obiektu, a jednocześnie pozwala na jego "uzmiennienie", ale już pod inną nazwą i adresem. Składnia operatora jest następująca:

```
const_cast<T>(arg)
```

gdzie `T` i `arg` muszą być tego samego typu, za wyjątkiem modyfikatorów **const** i **volatile**. Wynik konwersji jest typu `T`.

Przykład 11.8.

```
//const_cast: wsk jest const, wsk1 nie.
#include <iostream.h>
const int z1 = 10;
int   main() {
      cout << "z1: " << z1 << endl;
      const int* wsk = &z1;
      int* wsk1;
      wsk1 = const_cast<int*>(wsk);
      *wsk1 = 30;
      cout << "*wsk1: " << *wsk1 << endl;
      return 0;
}
```

Wydruk z programu ma postać:

z1: 10
***wski: 30**

•

Operator `reinterpret_cast` ma składnię:

```
reinterpret_cast<T>(arg)
```

gdzie `T` musi być wskaźnikiem, referencją, typem arytmetycznym, wskaźnikiem do funkcji, lub wskaźnikiem do elementu klasy.
Zgodnie ze swoją nazwą, operator ten można wykorzystać np. do konwersji z typu `int*` do `int` i odwrotnie, uzyskując z powrotem typ `int*`, co pokazano w poniższym przykładzie.

Przykład 11.9.

```
//reinterpret_cast
#include <iostream.h>
#include <typeinfo.h>
int   main() {
      int i1 = 10;
      cout << typeid(i1).name() << endl;
      int* wski = &i1;
      cout << *wski << endl;
      cout << typeid(wski).name() << endl;
//Konwersja z int* do int:
      i1 = reinterpret_cast<int>(wski);
      cout << typeid(i1).name() << endl;
//Konwersja z int do int*:
      wski = reinterpret_cast<int*>(i1);
      cout << typeid(wski).name() << endl;
      return 0;
}
```

Wydruk z programu ma postać:

int
10
int*
int
int*

Dyskusja. Stosując dwukrotnie operator `reinterpret_cast` do tej samej pary zmiennych wróciliśmy do pierwotnego typu. Operator ten w ogólności można stosować do konwersji wskaźnika dowolnego typu we wskaźnik dowolnego typu. Jak łatwo przewidzieć, nie będą to konwersje bezpieczne i na ogół będą zależne od implementacji, nie dając gwarancji przenośności programu. Jedyną konwersją bezpieczną jest konwersja do pierwotnego typu.
Tak więc operator `reinterpret_cast` jest prawie tak samo mało pewny, jak "stary" `(T)arg`. Jednak ten nowy operator jest bardziej widoczny, nigdy nie pozwala na "nawigację" w hierarchii klas i nie łamie stałości obiektów z modyfikatorem **const**.

Dodatek A. Zbiór znaków ASCII

D	o	z	d	o	z	d	o	z	d	o	z
0	0	NUL	32	40	SP	64	100	@	96	140	`
1	1	SQH	33	41	!	65	101	A	97	141	a
2	2	STX	34	42	"	66	102	B	98	142	b
3	3	ETX	35	43	#	67	103	C	99	143	c
4	4	EOT	36	44	$	68	104	D	100	144	d
5	5	ENQ	37	45	%	69	105	E	101	145	e
6	6	ACK	38	46	&	70	106	F	102	146	f
7	7	BEL	39	47	'	71	107	G	103	147	g
8	10	BS	40	50	(	72	110	H	104	150	h
9	11	HT	41	51	)	73	111	I	105	151	i
10	12	LF	42	52	*	74	112	J	106	152	j
11	13	VT	43	53	+	75	113	K	107	153	k
12	14	FF	44	54	,	76	114	L	108	154	l
13	15	CR	45	55	-	77	115	M	109	155	m
14	16	SO	46	56	.	78	116	N	110	156	n
15	17	SI	47	57	/	79	117	O	111	157	o
16	20	DLE	48	60	0	80	120	P	112	160	p
17	21	DC1	49	61	1	81	121	Q	113	161	q
18	22	DC2	50	62	2	82	122	R	114	162	r
19	23	DC3	51	63	3	83	123	S	115	163	s
20	24	DC4	52	64	4	84	124	T	116	164	t
21	25	NAK	53	65	5	85	125	U	117	165	u
22	26	SYN	54	66	6	86	126	V	118	166	v
23	27	ETB	55	67	7	87	127	W	119	167	w
24	30	CAN	56	70	8	88	130	X	120	170	x
25	31	EM	57	71	9	89	131	Y	121	171	y
26	32	SUB	58	72	:	90	132	Z	122	172	z
27	33	ESC	59	73	;	91	133	[	123	173	{
28	34	FS	60	74	<	92	134	\	124	174	|
29	35	GS	61	75	=	93	135	]	125	175	}
30	36	RS	62	76	>	94	136	^	126	176	~
31	37	US	63	77	?	95	137	_	127	177	DEL

Nagłówki w tabeli oznaczają: d – dziesiętny kod znaku, o – oktalny kod znaku, z – znak.

Znaczenie znaków sterujących:

0.	NUL	- znak zerowy
1.	SOH (Start Of Heading)	- początek nagłówka = SOM
2.	STX (Start of Text)	-początek tekstu = EOA
3.	ETX (End of Text)	- koniec tekstu = EOM
4.	EOT (End of Transm.)	- koniec transmisji
5.	ENQ (Enquiry)	- wywołanie stacji
6.	ACK (Acknowledge)	- potwierdzenie
7.	BEL (Bell)	- dzwonek
8.	BS (Back Space)	- powrót o 1 pozycję
9.	HT (Horizontal Tab)	- tabulacja pozioma
10.	LF (Line Feed)	- przesuw o 1 wiersz
11.	VT (Vertical Tab)	- tabulacja pionowa
12.	FF (Form Feed)	- przesuw o 1 stronę
13.	CR (Carriage Return)	- powrót karetki
14.	SO (Switch Output)	- wyjście (przełączenie trwałe)
15.	SI (Switch Input)	- wejście (przełączenie powrotne)
16.	DLE (Data Link Escape)	- pominięcie znaków sterujących
17.	DC1 (Device Control 1)	- sterowanie urządz.1/ start transmisji=XON
18.	DC2 (Device Control 2)	- sterowanie urządzenia 2
19.	DC3 (Device Control 3)	- sterowanie urządz.3/ stop transmisji= XOFF
20.	DC4 (Device Control 4)	- sterowanie urządzenia 4
21.	NAK (Negative Acknowledge)	- potwierdz. negatywne (gdy błąd)
22.	SYN (Sync)	- synchronizacja
23.	ETB (End Transm. Block)	- koniec bloku
24.	CAN (Cancel)	- anulowanie
25.	EM (End of Medium)	- koniec nośnika (zapisu)
26.	SUB (Substitute)	- zastąpienie
27.	ESC (Escape)	- przełączenie
28.	FS (File Separator)	- poprzedza dane alfanumeryczne
29.	GS (Group Separator)	- poprzedza dane binarne
30.	RS (Record Separator)	- separator rekordów
31.	US (Unit Separator)	- separator pozycji
32.	SP (Space)	- spacja (odstęp)
127.	DEL (Delete)	- kasowanie

Dodatek B
Priorytety i łączność operatorów

Operator	Priorytet	Łączność	Działanie
`::`	17	L	Zasięg globalny
`::`		P	zasięg klasy
`.`	16	L	dostęp do składowej obiektu
`->`		L	dostęp do składowej obiektu
`[]`		L	indeksowanie
`()`		L	wywołanie funkcji
`()`		L	konstrukcja obiektu
`sizeof`		L	rozmiar obiektu/typu
`++`	15	P	przedrostkowe zwiększanie o 1
`++`		P	przyrostkowe zwiększanie o 1
`--`		P	przedrostkowe zmniejszanie o 1
`--`		P	przyrostkowe zmniejszanie o 1
`~`		P	negacja bitowa
`!`		P	negacja logiczna
`-`		P	minus jednoargumentowy
`+`		P	plus jednoargumentowy
`&`		P	adres argumentu/referencja
`*`		P	dostęp pośredni
`new`		P	tworzenie (przydział pamięci)
`delete`		P	usuwanie (zwalnianie pamięci)
`delete[]`		P	usuwanie tablicy
`()`		P	konwersja typu (rzutowanie)
`.*`	14	L	dostęp do składowej
`->*`		L	dostęp do składowej

Priorytet: im większa wartość, tym wyższy priorytet
Łączność : L – lewostronna, P – prawostronna

Operator	Priorytet	Łączność	Działanie
*	13	L	mnożenie
/		L	dzielenie
%		L	modulo (reszta z dzielenia)
+	12	L	dodawanie
-		L	odejmowanie
<<	11	L	przesuwanie w lewo
>>		L	przesuwanie w prawo
<	10	L	mniejsze
<=		L	mniejsze lub równe
>		L	większe
>=		L	większe lub równe
==	9	L	równe
!=		L	nierówne
&	8	L	koniunkcja bitowa
^	7	L	bitowa różnica symetryczna
|	6	L	alternatywa bitowa
&&	5	L	koniunkcja logiczna
||	4	L	alternatywa logiczna
? :	3	L	wyrażenie warunkowe
=	2	P	przypisanie
*=		P	mnożenie i przypisanie
/=		P	dzielenie i przypisanie
%=		P	modulo i przypisanie
+=		P	dodawanie i przypisanie
-=		P	odejmowanie i przypisanie
<<=		P	przesunięcie w lewo i przypisanie
>>=		P	przesunięcie w prawo i przypisanie
&=		P	koniunkcja bitowa i przypisanie
^=		P	różnica symetryczna i przypisanie
|=		P	alternatywa bitowa i przypisanie
,	1	L	ustalenie kolejności