

Programowanie obiektowe

dr inż. Sławomir Koczubiej

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego
Katedra Technologii Informatycznych

(29 października 2026)

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Kontakt

Budynek C, pokój 3.26
sk@tu.kielce.pl

Materiały do pobrania, aktualności, terminy zaliczeń

<http://staff.tu.kielce.pl/sk>

Organizacja wykładów

- Program wykładów obejmuje zagadnienia wykraczające poza ćwiczenia laboratoryjne.
- Uczestnictwo w wykładach pozwala zdobyć wiedzę fundamentalną, która nie jest omawiana szczegółowo na laboratoriach.
- Regularne uczestnictwo w wykładach jest najlepszym sposobem na zdanie egzaminu.
- Czasem sprawdzam listę obecności.

Warunki zaliczenia wykładu

Egzamin zaliczeniowy po zakończeniu wykładów.

Organizacja laboratoriów

- Zajęcia laboratoryjne są obowiązkowe.
- Dopuszcza się jedną nieobecność.
- Większa liczba nieobecności powoduje zmniejszenie oceny do niedostatecznej włącznie (30% lub więcej).
- W przypadku usprawiedliwionej nieobecności zajęcia można odrobić z inną grupą (jeśli istnieje taka możliwość).
- Zajęcia praktyczne stanowią jedynie wycinek kursu.

Warunki zaliczenia laboratoriów

Wykonanie ćwiczeń i zaliczenie sprawdzianów kontrolnych.

Tematyka wykładów

- Wprowadzenie do programowania w języku C++. Obiekty i klasy.
- Ochrona i kapsułkowanie. Interfejsy.
- Dziedziczenie, dziedziczenie wielobazowe. Polimorfizm.
- Obiekty i zarządzanie pamięcią. Tworzenie i niszczenie obiektów.
- Operatory przeciążone. Funkcje operatorowe.
- Strumienie i obsługa plików. Struktury i rekordy.
- Wyjątki i ich obsługa.
- Szablony.
- Biblioteka STL.

Tematyka laboratoriów

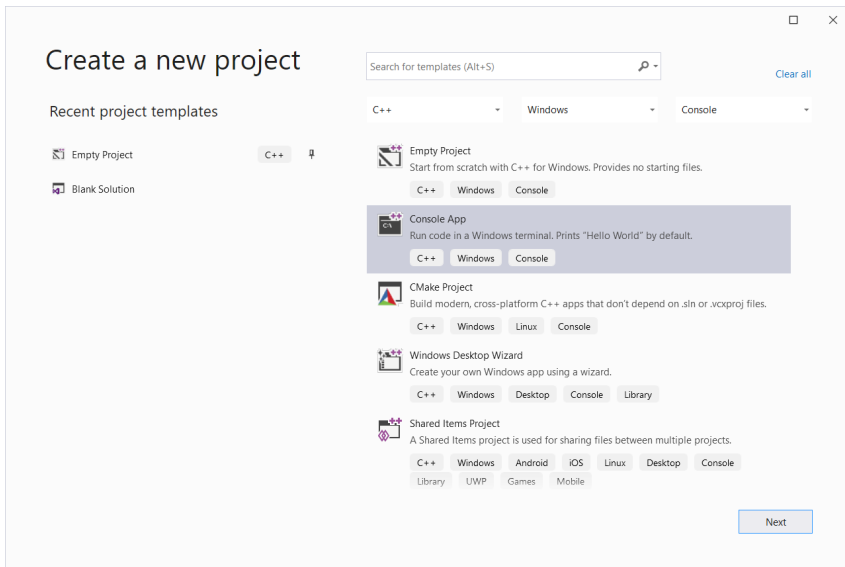
- Struktura programu w języku C++.
- Definiowanie klas. Składowe klasy, obiekty.
- Dziedziczenie i dziedziczenie wielobazowe.
- Polimorfizm i tablice wskaźników.
- Tworzenie i niszczenie obiektów. Konstruktor, destruktor i zarządzanie pamięcią.
- Redefiniowanie operatorów.
- Obsługa błędów.
- Obsługa plików.
- Kontenery i iteratory.

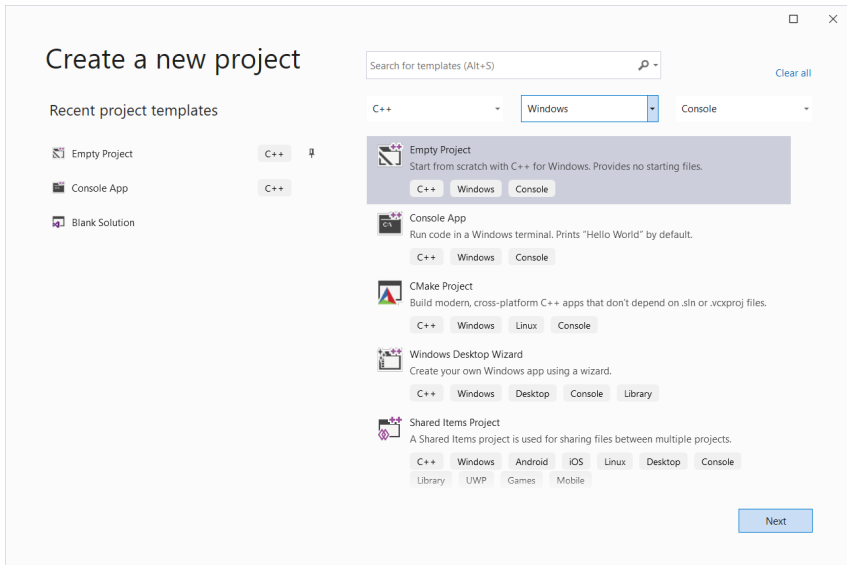
Literatura

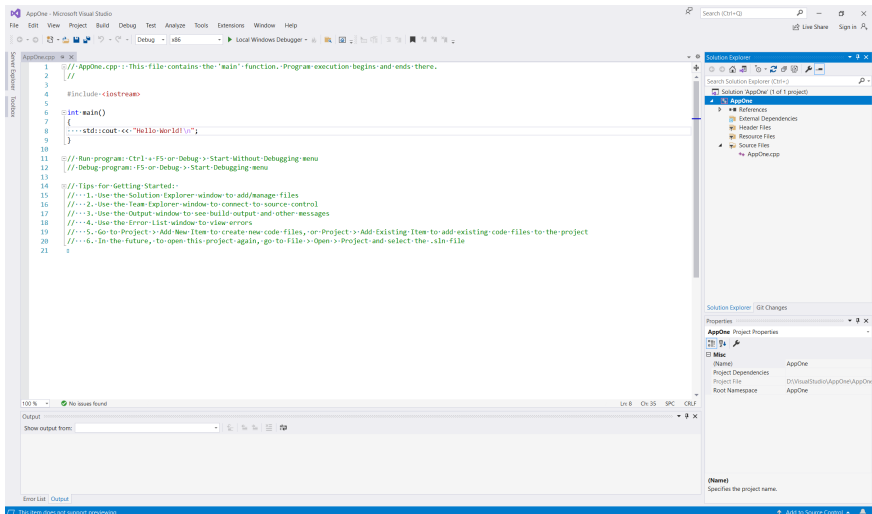
- Drozdek A. *C++. Algorytmy i struktury danych*. Wydawnictwo Helion, Gliwice 2004.
- Eckel B. *Thinking in C++*. Wydawnictwo Helion, Gliwice 2002.
- Grębosz J. *Symfonia C++ Standard*. Wydawnictwo Edition 2000, Kraków 2009.
- Prata S. *Szkoła programowania. Język C++*. Wydawnictwo Helion, Gliwice 2006.

- 1 Informacje ogólne
- 2 Środowisko programowania**
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Środowisko programowania Visual Studio – Windows



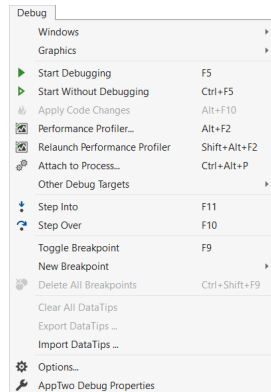




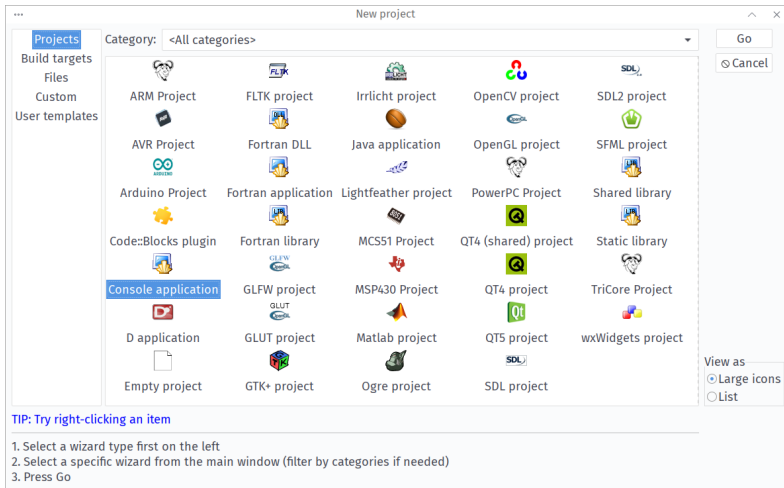
Kompilowanie, debugowanie, śledzenie

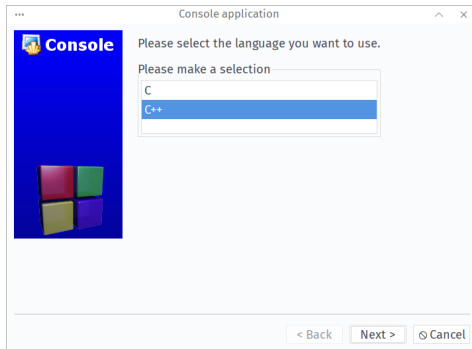
skróty klawiaturowe

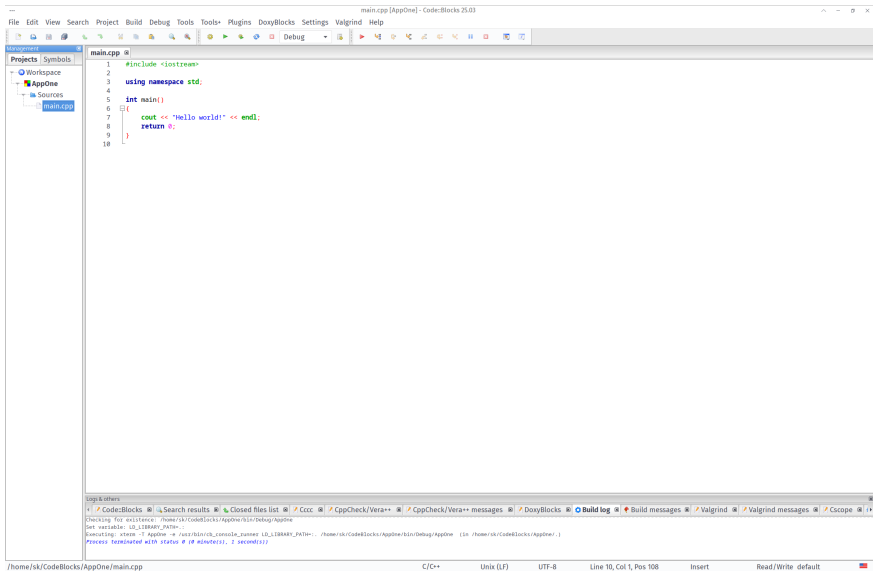
[F5]	start z nadzorowaniem (debugger) i kontynuacja wykonywania programu
[CTRL+F5]	start bez debuggera
[F11]	krokowe wykonywanie programu
[F10]	krokowe wykonywanie programu, procedury i funkcje w jednym kroku
[F9]	wstawianie i usuwanie punktów przerwania (<i>breakpoint</i>)



Środowisko programowania Code::Blocks – Linux



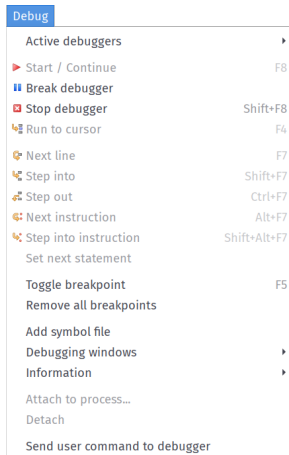




Kompilowanie, uruchamianie, debugowanie, śledzenie

skrótów klawiaturowe

[F9]	budowanie i uruchomienie programu
[CTRL+F9]	budowanie programu
[F7]	uruchomienie do następnej linii kodu
[F8]	uruchomienie i kontynuowanie do kolejnego punktu przerwania
[SHIFT+F7]	kontynuowanie z rozwijaniem funkcji
[CTRL+F7]	kontynuowanie do powrotu z funkcji
[F5]	wstawianie i usuwanie punktów przerwania (<i>breakpoint</i>)



- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start**
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Budowa programu (który nic nie robi)

```
int main()  
{  
    return 0;  
}
```

Każdy program C++ musi posiadać funkcję o nazwie `main()`, stanowiącą punkt wejściowy programu. Od niej rozpoczyna się wykonanie programu.

Budowa programu:

- typ rezultatu funkcji,
- nazwa funkcji,
- parametry funkcji,
- ciało funkcji,
- instrukcja powrotu z podprogramu,
- wartość przekazywana w miejscu wywołania.

Biblioteki i dyrektywa `#include`

Obiekty i funkcje wejścia/wyjścia, takie jak `cout`, nie są częścią języka C++, lecz jego **biblioteki standardowej**. Ich deklaracje znajdują się w pliku nagłówkowym `iostream`.

Dyrektywa `#include<>` powoduje włączenie do kodu źródłowego programu definicji i deklaracji zawartych w innych plikach źródłowych (często to dodatkowe biblioteki).

Dyrektywa `#include<>` jest realizowana przez **preprocesor** języka C++.

Pliki nagłówkowe w języku C++ mogą mieć rozszerzenia `*.h`, `*.hpp`, `*.hxx`.

Aktualnie w kodach źródłowych C++ nie pisze się rozszerzeń nazw plików nagłówkowych biblioteki standardowej. Własne nagłówki dalej wymagają rozszerzenia `*.h` lub `*.hpp`. Dodatkowo, nazwy plików nagłówkowych znanych z języka C pisze się z prefiksem `c`.

```
#include <stdio.h>           ← C  
#include "foo.h"             ← C, własny nagłówek
```

```
#include <iostream>          ← C++  
#include <cstdio>            ← C++, ten sam nagłówek C  
#include "foo.h"             ← C++, własny nagłówek
```

Nagłówki z rozszerzeniem `*.h` znane z języka C nadal są w C++ dostępne, ale w nowym kodzie należy używać wersji z prefiksem `c`.

Standardowe strumienie

Standardowe kanały komunikacji między komputerem a otoczeniem (zwykle terminalem). Występują w Uniksie i systemach uniksopodobnych, w środowisku uruchomieniowym C/C++ i ich pochodnych.

Trzy podstawowe połączenia I/O:

- **stdin** – **standardowe wejście** programu, zwykle kojarzone z klawiaturą (*standard input*),
- **stdout** – **standardowe wyjście** programu, zwykle kojarzone z ekranem monitora (*standard output*),
- **stderr** – **standardowe wyjście błędów**, zwykle kojarzone z ekranem monitora (*standard error*).

`stdin` i `stdout` reprezentują normalny kanał komunikacji programu z użytkownikiem.

`stderr` zarezerwowany jest do wyświetlania komunikatów diagnostycznych programu.

Przykład **przekierowania** (redyrekcji) strumieni z poziomu systemu operacyjnego:

```
C:\> foo.exe > output.txt
C:\> foo.exe < input.txt
C:\> foo.exe < input.txt > output.txt
```

```
user@linux:~$ ./foo > output.txt
user@linux:~$ ./foo < input.txt
user@linux:~$ ./foo < input.txt > output.txt
```

Strumienie w języku C++:

- `cin` – strumień reprezentujący standardowe wejście, odpowiada strumieniowi `stdin` z C,
- `cout` – strumień reprezentujący standardowe wyjście, odpowiada strumieniowi `stdout` z C,
- `cerr` – niebuforowany strumień wyjściowy błędów, odpowiada strumieniowi `stderr` z C,
- `clog` – buforowany strumień wyjściowy błędów, odpowiada strumieniowi `stderr` z C.

Aby skorzystać ze strumieni, należy włączyć plik nagłówkowy `iostream`.

W języku C++ strumienie są **obiektami**.

Poważny program w języku C++

```
#include <iostream>

int main()
{
    float cm, inch;

    std::cout << "Przeliczanie centymetrow na cale." << std::endl;
    std::cout << "Podaj wymiar w cm: ";

    std::cin >> cm;

    inch = 0.3937 * cm;

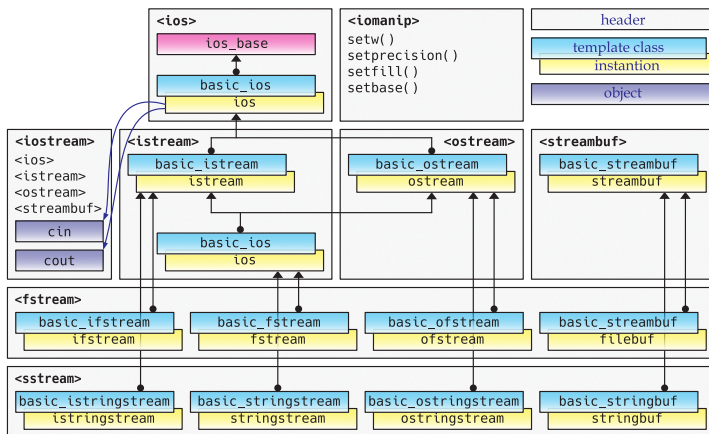
    std::cout << cm << "cm to " << inch << "\"." << std::endl;

    return 0;
}
```

```
Przeliczanie centymetrow na cale.  
Podaj wymiar w cm: 5  
5cm to 1.9685".
```

Obsługa wejścia/wyjścia

Język C++, podobnie jak C, nie posiada instrukcji do obsługi standardowego wejścia i wyjścia.



Obsługa wejścia-wyjścia jest realizowana z wykorzystaniem zaawansowanych możliwości tego języka. Należą do nich: **klasy, klasy pochodne, przeciążanie funkcji i operatorów, funkcje wirtualne, szablony i wielodziedziczenie.**

Zarządzanie strumieniami i buforami może być dość skomplikowane, dlatego biblioteka `iostream` udostępnia kilka klas, których zadaniem jest ułatwienie zarządzania tymi elementami:

- klasa `streambuf` – zapewnia obszar pamięci na bufor wraz z metodami służącymi do wypełniania bufora, odwoływania się do zawartości bufora, opróżniania bufora, zarządzania pamięcią bufora,
- klasa `ios_base` – reprezentuje ogólne właściwości strumienia informujące o tym, czy jest otwarty, czy jest strumieniem tekstowym czy też binarnym,
- klasa `ios` – jest pochodną klasy `ios_base` i zawiera wskaźnik na obiekty klasy `streambuf`,
- klasa `ostream` – dziedziczy z klasy `ios` i udostępnia metody do obsługi wyjścia,
- klasa `istream` – dziedziczy z klasy `ios` i udostępnia metody do obsługi wejścia,
- klasa `iostream` – wielodziedziczy po klasach `istream` i `ostream`, zawiera metody do obsługi wejścia jak i wyjścia.

Aby skorzystać z takich udogodnień, należy użyć obiektów odpowiednich klas. Na przykład do obsługi wyjścia użyjemy obiektu klasy `ostream`, jak choćby `cout`.

W C++ nie trzeba tworzyć obiektu `cout`, jest zdefiniowany jako **globalny obiekt strumienia** wyjściowego w bibliotece `<iostream>`.

Po dołączeniu nagłówka `<iostream>` kompilator zyskuje dostęp do deklaracji obiektu `cout` automatycznie inicjalizowanego przez bibliotekę.

Obiekt jest automatycznie powiązywany ze **standardowym strumieniem wyjściowym** (ekranem lub konsolą) jeszcze przed uruchomieniem funkcji `main()`.

`cout` jest zdefiniowany w **przestrzeni nazw biblioteki standardowej `std`**. Aby z niego skorzystać, należy wskazać jego pełną nazwę `std::cout` lub użyć deklaracji `using namespace std`.

Obiekt `cout` jest zwykle łączony z operatorem `<<` (operator **wstawiania**).

Operator wstawiania jest **przeciążony** i współpracuje ze wszystkimi podstawowymi typami C++:

- `bool`,
- `unsigned char`,
- `signed char`,
- `char`,
- `short`,
- `unsigned short`,
- `int`,
- `unsigned int`,
- `long`,
- `unsigned long`,
- `long long`,
- `unsigned long long`,
- `float`,
- `double`,
- `long double`.

Klasa `ostream` z obiektem `cout` zapewnia definicję funkcji operatorowej `operator << ( )` dla każdego z typów danych.

W klasie `ostream` zdefiniowano też funkcje wstawiania dla typów wskaźnikowych:

- `const unsigned char *`,
- `const signed char *`,
- `const char *`,
- `void *`.

```
const char *str = "Ala ma psa.";
char name[20] = "Wiktor Wektor";

cout << "Halo!";
cout << name;
cout << str;
```

Operacje z użyciem obiektów pochodnych z klasy `ostream`, czyli na przykład `cout`, są buforowane.

Mechanizm buforowania jest korzystny, gdy standardowe wyjście jest skojarzone z plikiem. Może być kłopotliwy, gdy jest skojarzone z ekranem – wypisany tekst może pojawić się później, niż się go spodziewamy, a przy awaryjnym zakończeniu programu zawartość bufora przepada.

Bufor `cout` jest opróżniany na pewno po `endl` i `flush`, przed każdą operacją odczytu z `cin` (strumienie są powiązane) oraz przy normalnym zakończeniu programu. Dlatego zachęta pojawi się przed oczekiwaniem na dane.

```
cout << "Podaj liczbę: ";  
cin >> n;
```


Tekst bez `\n` i bez `flush` może pokazać się dopiero po zakończeniu obliczeń.

```
cout << "Przetwarzanie..."; ← bufor nie musi zostać opróżniony
longComputations();         ← napis może się pokazać dopiero teraz
```

Przy awarii programu zawartość bufora przepada – ostatni komunikat **nie dochodzi**, więc błędu szuka się w złym miejscu.

```
cout << "doszedlem tutaj"; ← napis trafia do bufora
int *p = nullptr;
*p = 1;                     ← awaria, zawartość bufora przepada
```

Dlatego komunikaty diagnostyczne wypisuje się do `cerr`, który jest opróżniany po każdej operacji.

```
cerr << "doszedlem tutaj"; ← napis pojawi się na pewno
```

Manipulatory są w rzeczywistości funkcjami. Można opróżnić bufor obiektu `cout` wywołując funkcję bezpośrednio:

```
flush(cout);
```

Operator wstawiania klasy `ostream` konwertuje wartości na postać tekstową:

- wartość **typu znakowego**, jeśli reprezentuje znak drukowalny, jest wyświetlana jako znak w polu o szerokości jednego znaku,
- liczbowe **typy całkowite** są wyświetlane jako całkowite liczby dziesiętne w polu o szerokości odpowiedniej, aby pomieścić daną liczbę i ewentualnie znak minusa,
- **łańcuchy** są wyświetlane w polu o szerokości równej długości danego łańcucha,
- **typy zmiennoprzecinkowe** są wyświetlane przy użyciu sześciu cyfr, zera końcowe są pomijane, liczba jest wyświetlana w notacji stałoprzecinkowej lub notacji naukowej, notacja naukowa jest stosowana, gdy wykładnik jest większy od 5 lub mniejszy od -4.

```
cout << "12345678901234567890\n";
```

```
char z = 'K';
```

```
int t = 273;
```

```
cout << z << endl;
```

```
cout << t << endl;
```

```
cout << -t << endl;
```

```
double a = 1.200;
```

```
cout << a << endl;
```

```
cout << (a + 1.0 / 9.0) << endl;
```

```
double b = 1.67E2;
```

```
cout << b << endl;
```

```
b += 1.0 / 9.0;
```

```
cout << b << endl;
```

```
cout << (b * 1.0e4) << endl;
```

```
double c = 2.3e-4;
```

```
cout << c << endl;
```

```
cout << c / 10 << endl;
```

```
12345678901234567890
```

```
K
```

```
273
```

```
-273
```

```
1.2
```

```
1.31111
```

```
167
```

```
167.111
```

```
1.67111e+06
```

```
0.00023
```

```
2.3e-05
```

Manipulatory służą też do **formatowania** standardowego wyjścia.

Manipulatory są używane w dwóch kategoriach: zmiana **sposobu prezentowania** wartości liczbowych i kontrolowanie **szerokości i położenia wypełnienia**.

Większość manipulatorów, które zmieniają format występuje w parach: jeden manipulator ustawia stan formatu na nową wartość, a drugi go usuwa przywracając domyślne formatowanie.

Manipulatory zmieniające format strumienia zazwyczaj pozostawiają zmieniony stan dla wszystkich kolejnych operacji wyjścia.

manipulatory zdefiniowane w `<iostream>` i `<iomanip>`

<code>boolalpha</code>	wyświetla <code>true</code> i <code>false</code> jako napisy
<code>noboolalpha</code>	wyświetla <code>true</code> i <code>false</code> jako wartości numeryczne <code>1</code> i <code>0</code>
<code>showbase</code>	dodaje prefiks wskazujący podstawę systemu liczenia
<code>noshowbase</code>	nie dodaje prefiksu systemu liczenia
<code>showpoint</code>	zawsze wyświetla kropkę dziesiętną
<code>noshowpoint</code>	wyświetla kropkę dziesiętną, gdy wartość ma część ułamkową
<code>showpos</code>	wyświetla znak <code>+</code> w liczbach nieujemnych
<code>noshowpos</code>	nie wyświetla znaku <code>+</code> w liczbach nieujemnych
<code>uppercase</code>	wyświetla <code>0X</code> w systemie szesnastkowym i <code>E</code> w notacji naukowej
<code>nouppercase</code>	wyświetla <code>0x</code> w systemie szesnastkowym i <code>e</code> w notacji naukowej
<code>dec</code>	wyświetla wartości całkowite w systemie dziesiętnym
<code>hex</code>	wyświetla wartości całkowite w systemie szesnastkowym
<code>oct</code>	wyświetla wartości całkowite w systemie ósemkowym

manipulatory zdefiniowane w `<iostream>` i `<iomanip>`

<code>left</code>	dodaje znaki wypełnienia po prawej stronie wartości
<code>right</code>	dodaje znaki wypełnienia po lewej stronie wartości
<code>internal</code>	dodaje znaki wypełnienia między znakiem a wartością
<code>fixed</code>	wyświetla wartości zmiennoprzecinkowe w notacji dziesiętnej
<code>scientific</code>	wyświetla wartości zmiennoprzecinkowe w notacji naukowej
<code>hexfloat</code>	wyświetla wartości zmiennoprzecinkowe w systemie szesnastkowym (od C++ 11)
<code>defaultfloat</code>	resetuje format zmiennoprzecinkowy do domyślnego (od C++ 11)
<code>unitbuf</code>	opróżnia bufor po każdej operacji wyjściowej
<code>nounitbuf</code>	przywraca domyślne opróżnianie bufora
<code>skipws</code>	pomija białe znaki w operatorze wejściowym
<code>noskipws</code>	nie pomija białych znaków w operatorze wejściowym
<code>flush</code>	opróżnij bufor <code>ostream</code>
<code>ends</code>	wstawia <code>null</code> ,
<code>endl</code>	wstawia znak nowego wiersza, a następnie opróżnia bufor

Przykład formatowania wartości logicznych:

```
cout << "default bool values: " << true << " " << false << endl;  
cout << "alpha bool values: " << boolalpha << true << " " << false << endl;
```

```
default bool values: 1 0  
alpha bool values: true false
```


Wartości całkowite w różnych systemach liczbowych:

```
cout << "default: " << 20 << ", " << 1024 << endl;  
cout << "octal: " << oct << 20 << ", " << 1024 << endl;  
cout << "hex: " << hex << 20 << ", " << 1024 << endl;  
cout << "decimal: " << dec << 20 << ", " << 1024 << endl;
```

```
default: 20, 1024  
octal: 24, 2000  
hex: 14, 400  
decimal: 20, 1024
```

Wartości całkowite w różnych systemach liczbowych, dołączony prefiks systemu liczenia:

```
cout << showbase;
cout << "default: " << 20 << ", " << 1024 << endl;
cout << "in octal: " << oct << 20 << ", " << 1024 << endl;
cout << "in hex: " << hex << 20 << ", " << 1024 << endl;
cout << "in decimal: " << dec << 20 << ", " << 1024 << endl;
cout << noshowbase;
```

```
default: 20, 1024
in octal: 024, 02000
in hex: 0x14, 0x400
in decimal: 20, 1024
```

```
cout << uppercase << showbase << hex;  
cout << "printed in hexadecimal: " << 20 << ", " << 1024;  
cout << nouppercase << noshowbase << dec << endl;
```

```
printed in hexadecimal: 0X14, 0X400
```

Formatowanie i precyzja wartości zmiennoprzecinkowych:

```
cout << "precision: " << cout.precision() << ", value: " << sqrt(2.0) << endl;  
cout.precision(12);  
cout << "precision: " << cout.precision() << ", value: " << sqrt(2.0) << endl;  
cout << setprecision(3);  
cout << "precision: " << cout.precision() << ", value: " << sqrt(2.0) << endl;
```

```
precision: 6, value: 1.41421  
precision: 12, value: 1.41421356237  
precision: 3, value: 1.41
```

```
cout << "default format: " << 100 * sqrt(2.0) << endl;  
cout << "scientific: " << scientific << 100 * sqrt(2.0) << endl;  
cout << "fixed decimal: " << fixed << 100 * sqrt(2.0) << endl;  
cout << "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << endl;  
cout << "use defaults: " << defaultfloat << 100 * sqrt(2.0) << endl;
```

```
default format: 141  
scientific: 1.414e+02  
fixed decimal: 141.421  
hexadecimal: 0x1.1ad7bc01366b8p+7  
use defaults: 141
```

```
cout << 10.0 << endl;  
cout << showpoint << 10.0 << endl;  
cout << noshowpoint << endl;
```

```
10  
10.0
```

Wcięcia, justowanie, wypełnienie:

```
int i = -16;
double d = 3.14159;

cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << left;
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << right;
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << internal;
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << setfill('#');
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;
cout << setfill(' ');
```

```
i:      -16
d:      3.14
i: -16
d: 3.14
i:      -16
d:      3.14
i: -      16
d:      3.14
i: -#####16
d: #####3.14
```


Klasy `istream` i `ostream` zawierają również *niskopoziomowe metody* (funkcje) pozwalające na bezpośrednią obsługę wejścia i wyjścia z pominięciem formatowania.

funkcje jednoznakowe

<code>is.get(ch)</code>	pobiera kolejny znak ze strumienia wejściowego i przypisuje do <code>ch</code>
<code>os.put(ch)</code>	umieszcza znak w strumieniu wyjściowym
<code>is.get()</code>	zwraca kolejny znak ze strumienia wejściowego po konwersji do <code>int</code>
<code>is.putback(ch)</code>	cofa znak do strumienia wejściowego
<code>is.unget()</code>	umieszcza ostatni odczytany znak z powrotem w strumieniu wejściowym
<code>is.peek()</code>	odczytuje kolejny znak bez pobierania go

funkcje wieloznakowe

<code>is.get(sink, size, delim)</code>	odczytuje łańcuch znaków i zapisuje je w tablicy znakowej <code>sink</code>
<code>is.getline(sink, size, delim)</code>	odczytuje dane ze strumienia do napotkania znaku nowej linii
<code>is.read(sink, size)</code>	odczytuje znaki ze strumienia i zapisuje w tablicy znakowej <code>sink</code>
<code>is.gcount()</code>	zwraca liczbę znaków odczytanych przez ostatnią operację wejścia
<code>os.write(source, size)</code>	wstawia określoną liczbę znaków do strumienia
<code>is.ignore(size, delim)</code>	pomija znaki w strumieniu wejściowym

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych**
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Z czego jest zbudowany program?

W trakcie procesu kompilacji kod źródłowy jest dzielony na **jednostki leksykalne**.

Rozróżnia się następujące klasy jednostek leksykalnych:

- identyfikatory (*identifiers*),
- słowa kluczowe (*keywords*),
- stałe (*constants*),
- napisy (łańcuchy znaków, *string literals*),
- operatory (*operators*),
- separatory (*punctuators*).

Identyfikatory

Identyfikatory to ciągi liter, cyfr i znaków podkreślenia, muszą zaczynać się od litery lub znaku podkreślenia. Duże i małe litery są rozróżniane.

Standard zaleca rozróżnianie co najmniej 1024 znaków, współczesne kompilatory nie mają praktycznego limitu. Najnowsze gcc i MSVC przyjmują identyfikatory w UTF-8, zatem można stosować polskie znaki, ale **nie należy** tego robić.

Nazwy zaczynające się od podkreślenia i wielkiej litery oraz nazwy zawierające dwa podkreślenia są zarezerwowane dla kompilatora i biblioteki standardowej. Pojedyncze podkreślenie na początku nazwy jest dopuszczalne tylko lokalnie – bezpieczniej go unikać.

Identyfikatory są arbitralnie wybranymi nazwami dla **zmiennych, stałych, funkcji, typów danych** definiowanych przez programistę. Identyfikatory nie mogą być **słowami kluczowymi**.

maxvalue	mindelta	Pi_5
max_value	min_delta	Pi_Five
MaxValue	MinDelta	JW23
maxValue	minDelta	JW_23

Słowa kluczowe

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

and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	mutable	namespace	new	not
not_eq	operator	or	or_eq	private
protected	public	reinterpret_cast	static_cast	template
this	throw	true	try	typeid
typename	using	virtual	wchar_t	xor
xor_eq				

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

W kolejnych wersjach pojawiły się kolejne słowa kluczowe.

<code>alignas</code>	<code>alignof</code>	<code>char16_t</code>	<code>char32_t</code>	<code>constexpr</code>	<code>decltype</code>	← C++11
<code>noexcept</code>	<code>nullptr</code>	<code>static_assert</code>	<code>thread_local</code>			

<code>char8_t</code>	<code>concept</code>	<code>constexpr</code>	<code>constinit</code>	<code>co_await</code>	<code>co_return</code>	← C++20
<code>co_yield</code>	<code>requires</code>					

<code>contract_assert</code>	← C++26
------------------------------	---------

Separatory

Nawiasy kwadratowe (*brackets*) [] wykorzystywane są do deklarowania i odwoływania się do jedno- i wielowymiarowych tablic.

Nawiasy okrągłe (*parentheses*) () wykorzystywane są do grupowania wyrażeń, izolowania wyrażeń warunkowych, wskazując wywołanie funkcji i jej parametry.

Nawiasy klamrowe (*braces*) { } oznaczają początek i koniec instrukcji złożonej (blok).

Przecinek (*comma*) , rozdziela zwykle elementy na liście parametrów funkcji, występuje również w wyrażeniach przecinkowych.

Średnik (*semicolon*) ; jest znakiem kończącym instrukcję. Każde wyrażenie w języku C++ (również wyrażenie puste) zakończone znakiem średnika jest interpretowane jako instrukcja wyrażeniowa.

Komentarze

Komentarze to fragmenty tekstu spełniające funkcje dowolnych objaśnień robionych przez programistów dla programistów.

Komentarze nie mogą występować w napisach i stałych znakowych. Komentarze są usuwane z tekstu źródłowego programu.

```
/* To jest komentarz jednoliniowy */  
  
/*  
Ten komentarz obejmuje  
kilka linii  
kodu  
*/  
  
// Komentarz jednowierszowy, bez znacznika konca
```

Typy zmiennych w języku C++

typ, wielkość, charakterystyka

<code>bool</code>	1	wartość logiczna (rozmiar zależy od implementacji, praktycznie 1)
<code>char</code>	1	znak, liczba całkowita, ze znakiem lub bez (zależnie od implementacji)
<code>wchar_t</code>	2/4	znak poza ASCII (2 bajty w Windows, 4 w Linuksie)
<code>char8_t</code>	1	znak, typ dedykowany do kodowania UTF-8
<code>char16_t</code>	2	znak, typ dedykowany do kodowania UTF-16
<code>char32_t</code>	4	znak, typ dedykowany do kodowania UTF-32
<code>int</code>	4	liczba całkowita
<code>float</code>	4	liczba rzeczywista
<code>double</code>	8	liczba rzeczywista długa

Modyfikatory: `signed`, `unsigned`, `short`, `long` znane z języka C mogą być analogicznie stosowane w języku C++.

typ, wielkość, format, wartości

<code>char</code>	1	<code>signed</code>	od -128 do 127
<code>char</code>	1	<code>unsigned</code>	od 0 do 255
<code>char16_t</code>	2	UTF-16	od 0 do 65 535
<code>char32_t</code>	4	UTF-32	od 0 do 4 294 967 295
<code>short</code>	2	<code>signed</code>	od -32 768 do 32 767
<code>short</code>	2	<code>unsigned</code>	od 0 do 65 535
<code>int</code>	4	<code>signed</code>	od -2 147 483 648 do 2 147 483 647
<code>int</code>	4	<code>unsigned</code>	od 0 do 4 294 967 295
<code>long long</code>	8	<code>signed</code>	od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
<code>long long</code>	8	<code>unsigned</code>	od 0 do 18 446 744 073 709 551 615

Typ `long` ma 4 bajty w Windows i 8 bajtów w 64-bitowym Linuksie. Dawniej, w systemach 16-bitowych, typ `int` miał 2 bajty. W `char32_t` mieści się każdy punkt kodowy Unicode (największy to `0x10FFFF`, czyli `1 114 111`).

typ, wielkość, format, wartości

<code>float</code>	4	IEEE-754	od <code>-3.402 823 4e+38</code> do <code>-1.175 494 4e-38</code> do <code>1.175 494 4e-38</code> do <code>3.402 823 4e+38</code>
<code>double</code>	8	IEEE-754	od <code>-1.797 693 134 862 315 7e+308</code> do <code>-2.225 073 858 507 201 4e-308</code> do <code>2.225 073 858 507 201 4e-308</code> do <code>1.797 693 134 862 315 7e+308</code>
<code>long double</code>	10 (12/16)	x86	od <code>-1.189 731 495 357 231 765 021e+4932</code> do <code>-3.362 103 143 112 093 506 263e-4932</code> do <code>3.362 103 143 112 093 506 263e-4932</code> do <code>1.189 731 495 357 231 765 021e+4932</code>
<code>long double</code>	16	IEEE-754	od <code>-1.189 731 495 357 231 765 085 759 326 628 007 016 2e+4932</code> do <code>-3.362 103 143 112 093 506 262 677 817 321 752 6e-4932</code> do <code>3.362 103 143 112 093 506 262 677 817 321 752 6e-4932</code> do <code>1.189 731 495 357 231 765 085 759 326 628 007 016 2e+4932</code>

Format x86 ma 10 bajtów danych, w pamięci zajmuje 12 (32 bity) lub 16 bajtów (64 bity). W MSVC `long double` jest tym samym co `double`.

Operatory (arytmetyczne, logiczne etc.), **wyrażenia** (i instrukcje wyrażeniowe), **instrukcje** (pętle, sterowanie etc.), zasady definiowania zmiennych, zasady **definiowania funkcji**, sposób definiowania i obsługi **tablic** są podobne jak w języku C (nie ma VLA, a `const int N` może być rozmiarem tablicy).

Literał znakowy `'x'` jest typu `char` w C++, a w C typu `int`.

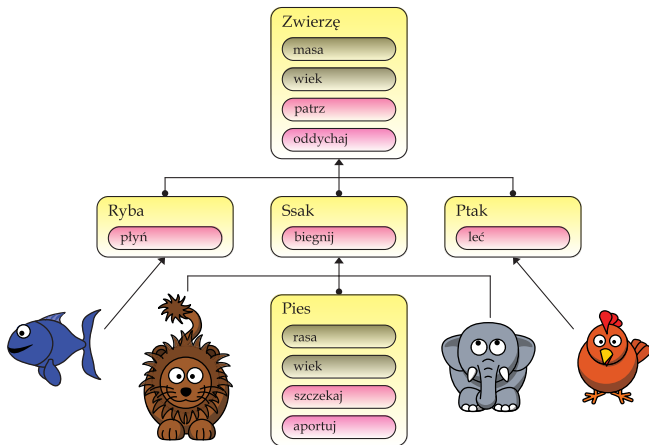
Zapominałskich odsyłam do wykładu z **Podstaw programowania**.

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo**
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Największym atutem programowania obiektowego jest zbliżenie programów komputerowych do naszego sposobu postrzegania rzeczywistości. Często nazywa się to zmniejszeniem **luki reprezentacji**.

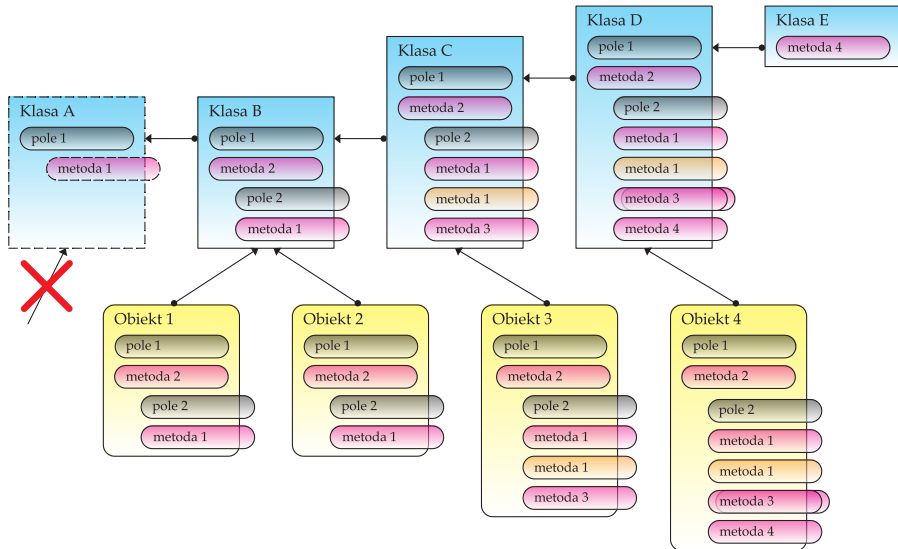
Wymyślając nowy lub analizując istniejący program obiektowy nasz mózg ma ułatwione zadanie. Dlatego ludzie są w stanie łatwiej zapanować nad kodem i tym samym tworzyć większe programy. Łatwiej jest również zrozumieć kod i pomysły innych programistów i tym samym współpracować w zespole oraz ponownie wykorzystywać istniejące rozwiązania.

Co więcej tego samego, naturalnego dla ludzi sposobu myślenia i tych samych pojęć można użyć przy analizie problemu, który ma być rozwiązany i projektowaniu jego programowego rozwiązania.

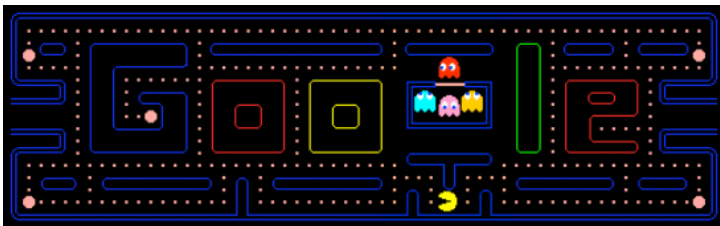


Arystoteles analizując rzeczywistość wprowadził pojęcia **formy** i **materii**. Formie w programowaniu obiektowym odpowiada **klasa** (*class*), materii jej **egzemplarz** (*instance*) wymiennie nazywany **obiekt**em (*object*).

Klasyfikacja, czyli łączenie występujących w rzeczywistości obiektów w grupy – klasy, jest najbardziej naturalnym sposobem rozumienia rzeczywistości.



Problem – Napisać program pozwalający na sterowanie Pacmanem (klasyczna gra), w najprostszej implementacji konsolowej. Powinien pozwalać na wyświetlanie wybranego znaku i sterowanie przy użyciu klawiszy kursorów.



Pozycja dowolnego, ustalonego wcześniej znaku o kodzie `code` będzie opisana za pomocą współrzędnych `x` i `y`. Przesuwanie znaku polega na wymazywaniu znaku z poprzedniej pozycji i ponownym wyświetlaniu na nowej pozycji. Pozycja powinna być sprawdzana.

Realizacja proceduralna

Programista ma do dyspozycji:

- funkcję `clearScreen( )`, czyszczącą ekran,
- funkcję `writeChar( )`, wyświetlającą na ekranie znak na określonej pozycji,
- funkcję `getKey( )`, której rezultatem jest kod klawisza sterującego (np. zmienna wyliczeniowa z wartościami typu: `KEY_ESC`, `KEY_UP`, `KEY_DOWN`, itd.).

```

int key, x = 1, y = 1; char code = '*';
clearScreen();
do
{
    writeChar(x, y, code); ← wyświetl znak na zadanej pozycji

    key = getKey(); ← obsługa naciśnięcia klawisza
    writeChar(x, y, ' '); ← nadpisanie znaku
    switch(key)
    {
        case KEY_UP:
            if(y > 1) --y; ← idź wyżej
            break;
        case KEY_DOWN:
            if(y < 24) ++y; ← idź niżej
            break;
        case KEY_LEFT:
            if(x > 1) --x; ← idź w lewo
            break;
        case KEY_RIGHT:
            if(x < 80) ++x; ← idź w prawo
            break;
    }
}
while(key != KEY_ESC);

```

Realizacja proceduralna charakteryzuje się wadami:

- brak powiązania pomiędzy zmiennymi opisującymi ten sam obiekt,
- brak powiązania pomiędzy funkcjami wyświetlającą i ukrywającą znak (prawidłowość parametrów),
- trudność z odczytaniem przeznaczenia kodu,
- brak powiązań między danymi a akcjami na rzecz obiektu.

Realizacja obiektowa

```
TPacman pac;  
pac.x = 1; pac.y = 1; pac.code = '*';  
int key;  
clearScreen();  
pac.show();  
do  
{  
    switch(key = getKey())  
    {  
        case KEY_UP:  
            pac.moveUp();  
            break;  
        case KEY_DOWN:  
            pac.moveDown();  
            break;  
        case KEY_LEFT:  
            pac.moveLeft();  
            break;  
        case KEY_RIGHT:  
            pac.moveRight();  
            break;  
    }  
}  
while(key != KEY_ESC);
```

TPacman to nazwa **klasy** obiektów. Wszystkie obiekty tej klasy będą posiadały te same cechy i umiejętności.

Wszystkie informacje o obiekcie pamiętane są w jego **polach**. Polami klasy **TPacman** są **x**, **y** i **code**.

```
class TPacman
{
    public:
        int x;      ← pola
        int y;      ←
        char code;  ←

        void show();
        void hide();

        void moveUp();
        void moveDown();
        void moveLeft();
        void moveRight();
};
```


Oprócz pól, obiekt posiada także zaimplementowane funkcje będące jego składnikami. Funkcje te są nazywane **metodami** i są prototypowane w definicji klasy.

```
class TPacman
{
    public:
        int x;
        int y;
        char code;

        void show(); ← metody
        void hide(); ←

        void moveUp(); ← metody
        void moveDown(); ←
        void moveLeft(); ←
        void moveRight(); ←
};
```

Definicje metod umieszczane są zwykle na zewnątrz definicji klasy. Wtedy nazwa każdej metody jest kwalifikowana nazwą klasy.

```
void TPacman::show()  
{  
    writeChar(x, y, code);  
}
```

```
void TPacman::hide()  
{  
    writeChar(x, y, ' ');  
}
```

```
void TPacman::moveUp()  
{  
    hide();  
  
    if(y > 1)  
        --y;  
  
    show();  
}
```

```
void TPacman::moveDown()  
{  
    hide();  
  
    if(y < 24)  
        ++y;  
  
    show();  
}
```

```
void TPacman::moveLeft()  
{  
    hide();  
  
    if(x > 1)  
        --x;  
  
    show();  
}
```

```
void TPacman::moveRight()  
{  
    hide();  
  
    if(x < 80)  
        ++x;  
  
    show();  
}
```

Klasa określa właściwości i możliwości każdego obiektu.

W szczególności klasa zawiera specyfikację danych i metod, informację o typie i reprezentacji pól, implementację metod, informacje o powiązaniach z innymi klasami.

Obiekt jest abstrakcją pewnego konkretnego bytu ze świata rzeczywistego, reprezentującego **rzecz**, **pojęcie**, **zjawisko** lub pewien **byt programistyczny** niemający swojego odpowiednika w rzeczywistości.

Obiekt jest **instancją** klasy.

Podstawowymi cechami obiektu są:

- Tożsamość (*identity*).
- Stan (*state*).
- Zachowanie, działanie (*behavior*).

Tożsamość – pozwala na odróżnienie danego obiektu od innego. Programista przypisuje obiektowi **unikatową nazwę**, lub odwołuje się do obiektu za pomocą jego adresu w pamięci (wskaźnika).

Stan – opisuje obiekt. Stan obiektu jest opisany zestawem atrybutów i ich wartości. Atrybuty są wewnętrznymi danymi obiektu, czyli **polami** (*fields*). Wartości atrybutów mogą się zmieniać w czasie wykonania programu.

Zachowanie – zdolność do działania opisują operacje, które obiekt może wykonać. Operacje są realizowane przez wykonanie **metod** (*methods*), czasem zwanych funkcjami składowymi (*member functions*). Metody mogą otrzymywać parametry i zwracać wartości, zwykle zmieniają stan obiektu. Metody są częścią obiektu.

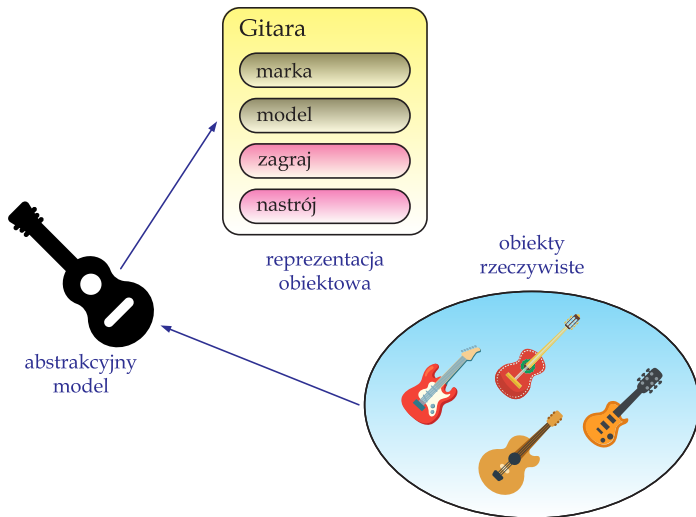
Powszechnie uważa się, że najważniejszymi cechami programowania obiektowego są: **abstrakcja**, **dziedziczenie**, **enkapsulacja** i **polimorfizm**.

Abstrakcja (*abstraction*)

Każdy obiekt w systemie służy jako model abstrakcyjnego wykonawcy (bytu ze świata rzeczywistego), który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie.

Abstrakcja pozwala postrzegać modelowaną rzeczywistość bez wnikania w jej wewnętrzną strukturę i bez ujawniania, w jaki sposób zaimplementowano dane cechy.

Obiekt jest dostawcą pewnych **informacji** i **usług**. Nie musimy wiedzieć, jak obiekt coś robi, tylko co robi. Nie musimy też wiedzieć, jak obiekt coś przechowuje, tylko co przechowuje.



```
class TPacman
{
    public:
        int x;
        int y;
        char code;

        void show();
        void hide();

        void moveUp();
        void moveDown();
        void moveLeft();
        void moveRight();
};
```

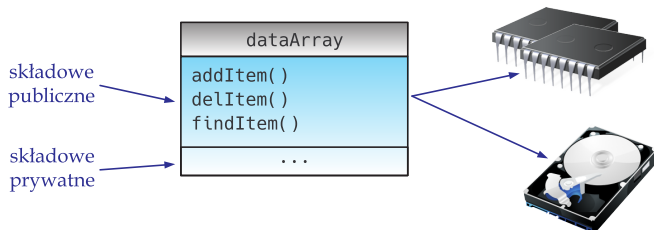
Enkapsulacja

Czyli ukrywanie implementacji, hermetyzacja. Zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Tylko własne metody obiektu są uprawnione do zmiany jego stanu.

Każdy typ obiektu prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.

Pewne języki osłabiają to założenie, dopuszczając pewien poziom bezpośredniego (kontrolowanego) dostępu do wnętrza obiektu. Ograniczają w ten sposób poziom abstrakcji.

Zmiana implementacji powinna być dla użytkownika klasy przezroczysta.



```
TPacman pac;
```

```
pac.x = 40;
pac.y = 12;
pac.code = '*';
```

```
cout << pac.x << endl;
cout << pac.y << endl;
cout << pac.code << endl;
```

```
TPacman pac;
```

```
pac.setX(40);
pac.setY(12);
pac.setCode('*');
```

```
cout << pac.getX() << endl;
cout << pac.getY() << endl;
cout << pac.getCode() << endl;
```

```

class TPacman
{
    public:
        int x, y;
        char code;

        void show();
        void hide();

        void moveUp();
        void moveDown();
        void moveLeft();
        void moveRight();
};

```

```

class TPacman
{
    public:
        void setX(int newX);
        void setY(int newY);
        void setCode(int newCode);

        int getX();
        int getY();
        int getCode();

        void show();
        void hide();

        void moveUp();
        void moveDown();
        void moveLeft();
        void moveRight();

    private:
        int x, y;
        char code;
};

```

Enkapsulacja pozwala na kontrolowanie poprawności danych przekazywanych obiektowi.

```
TPacman pac;  
  
pac.setCode( '*');  
pac.setX(-400);  
pac.setY(1200);  
  
pac.show();
```

```
void TPacman::setX(int newX)  
{  
    if(isXOnScreen(newX))  
        x = newX;  
    else  
        x = 1; ← wartość domyślna  
}
```

```
void TPacman::setY(int newY)  
{  
    if(isYOnScreen(newY))  
        y = newY;  
    else  
        y = 1; ← wartość domyślna  
}
```

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory**
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Opracujemy obiektową wersję programu do obliczania pól powierzchni płaskich figur geometrycznych.

Zacznijmy od klasy `TSquare`. Jej najistotniejszą cechą (*zasada abstrakcji*) będzie długość boku `side`.

Stosując **zasadę hermetyzacji** ukrywamy dane w części prywatnej i tworzymy metody zapewniające dostęp do pól danych. Przykład użycia klasy:

```
TSquare s;  
  
s.setSide(10);  
cout << s.getSide();  
cout << s.area();
```

Metody klasy będziemy nieformalnie dzielić na:

- **akcesory** – funkcje umożliwiające pobieranie wartości pól,
- **modyfikatory** – funkcje umożliwiające zmianę wartości pól,
- **realizatory** – funkcje umożliwiające realizację usług na rzecz obiektu.

Deklaracja klasy:

```
class TSquare
{
    public:
        void setSide(double newSide);
        double getSide();
        double area();

    private:
        double side;
};
```

Definicja metod (poza deklaracją klasy z użyciem operatora zakresu `::`):

```
void TSquare::setSide(double newSide)
{
    side = newSide;
}

double TSquare::getSide()
{
    return side;
}

double TSquare::area()
{
    return side * side;
}
```

Konstruktory

Próba użycia obiektu `TSquare` bez ustalonej wielkości boku będzie przynosiła nieprzewidywalne efekty:

```
TSquare s;  
  
cout << s.area();
```

Aby zapobiec takim przypadkom, należy zdefiniować **konstruktor** – czyli specjalną metodę, która przygotowuje obiekt do jego użycia.

Konstruktor jest uruchamiany automatycznie przez kompilator. Nie ma zdefiniowanego typu rezultatu i nosi taką samą nazwę jak klasa.

Rodzaje konstruktorów:

- **konstruktor domyślny** (*default constructor*) – aktywowany, gdy tworzony jest obiekt bez jawnie określonych danych inicjalizujących,

```
TSquare s;
```

- **konstruktor ogólny** lub parametrowy (*general constructor*) – aktywowany, gdy tworzony obiekt ma jawnie określone dane inicjalizujące,

```
TSquare s(10);
```

- **konstruktor kopiujący** (*copy constructor*) – aktywowany, gdy tworzony jest obiekt inicjalizowany danymi z innego obiektu **tej samej klasy**,

```
TSquare a = s, b(s);
```

- **konstruktor konwertujący** lub rzutujący (*converting constructor*) – aktywowany, gdy obiekt jest tworzony na podstawie wartości **innego typu** (innej klasy lub typu wbudowanego).

```
TSquare s(10);  
TRectangle a = s, b(s);
```

Deklaracja domyślnego konstruktora:

```
class TSquare
{
    public:

        TSquare();

        void setSide(double newSide);
        double getSide();
        double area();

    private:
        double side;
};
```

Kontekst aktywowania konstruktora domyślnego:

```
TSquare s;

TSquare squares[5];
```

Intuicyjna realizacja konstruktora:

```
TSquare::TSquare()  
{  
    side = 1;  
}
```

Realizacja konstruktora z listą inicjalizacyjną:

```
TSquare::TSquare(): side(1)  
{  
}
```

Listę inicjalizacyjną podaje się po znaku dwukropka `:`, najpierw podajemy **nazwę pola**, a w nawiasach wyrażenie inicjalizujące (wartość pola).

Deklaracja konstruktora ogólnego:

```
class TSquare
{
    public:

        TSquare();
        TSquare(double defaultSide);

        void setSide(double newSide);
        double getSide();
        double area();

    private:
        double side;
};
```

Kontekst aktywowania konstruktora ogólnego:

```
TSquare s(2);
```

Intuicyjna realizacja konstruktora:

```
TSquare::TSquare(double defaultSide)
{
    side = defaultSide;
}
```

Realizacja konstruktora z listą inicjalizacyjną:

```
TSquare::TSquare(double defaultSide): side(defaultSide)
{
}
```


Konstruktor ogólny pozwala na inicjalizowanie obiektu z modyfikatorem `const`:

```
const TSquare cs;
```

```
cs.setSide(3); ← błąd
```

```
const TSquare cs(3);
```

Metody klasy mogą być uzupełnione modyfikatorem `const`, wtedy nie będą mogły modyfikować obiektu, ale będą mogły być wywoływane na rzecz obiektów stałych:

```
class TSquare
{
    public:

    TSquare();
    TSquare(double defaultSide);

    void setSide(double newSide);
    double getSide() const;
    double area() const;

    private:
    double side;
};
```

```
double TSquare::getSide() const
{
    return side;
}

double TSquare::area() const
{
    return side * side;
}
```

Wykorzystanie operatora zakresu ::

Czy parametr funkcji `setSide( )` może się nazywać `side`? Zamiast:

```
void TSquare::setSide(double newSide)
{
    side = newSide;
}
```

chcielibyśmy:

```
void TSquare::setSide(double side)
{
    side = side;  ← ???
}
```

Oczywiście, taki zapis jest błędny, choć program się skompiluje. To jest błąd logiczny.

W takim przypadku można użyć operatora zakresu:

```
void TSquare::setSide(double side)
{
    TSquare::side = side;
}
```

Podobny problem występuje w konstruktorze:

```
TSquare::TSquare(double side)
{
    side = side;
}
```

W takim przypadku można użyć operatora zakresu:

```
TSquare::TSquare(double side)
{
    side = side; ← ???
}
```

Problem z przestanianiem pól nie występuje, kiedy zastosujemy listę inicjalizacyjną:

```
TSquare::TSquare(double side): side(side)
{
}
}
```

Przeciążanie funkcji – dygresja

W klasie `TSquare` istnieją dwa identyfikatory `TSquare( )`:

- konstruktor domyślny `TSquare( )`,
- konstruktor ogólny `TSquare(double defaultSide)`.

Identyfikator `TSquare( )` jest **przeciążony** (przetadowany). Formalnie, funkcja jest przeciążona, jeśli istnieje inna funkcja o tej samej nazwie, ale różnej **sygnaturze**.

Sygnatura funkcji jest to nazwa funkcji, liczba argumentów, uporządkowany zbiór typów argumentów funkcji, klasa lub obszar nazw do którego funkcja należy. Typ zwracany do niej nie należy.

Przeciążanie jest jedną z form **polimorfizmu** – **statyczną**, ponieważ wersję funkcji wybiera kompilator na etapie kompilacji. Polimorfizm **dynamiczny**, w którym o wersji funkcji decyduje typ obiektu w trakcie działania programu, zostanie omówiony później.

Deklaracja konstruktora kopiującego:

```
class TSquare
{
    public:

        TSquare();
        TSquare(double defaultSide);
        TSquare(TSquare &otherSquare);

        void setSide(double newSide);
        double getSide() const;
        double area() const;

    private:
        double side;
};
```

Kontekst aktywowania konstruktora kopiującego:

```
TSquare s(2);

TSquare a = s;
TSquare c(s);
```

Podobnie, dla typów prostych: `int i = 1, j = i;`

Intuicyjna realizacja konstruktora:

```
TSquare::TSquare(TSquare &otherSquare)
{
    side = otherSquare.side;
}
```

Realizacja konstruktora z listą inicjalizacyjną:

```
TSquare::TSquare(TSquare &otherSquare): side(otherSquare.side)
{
}
```

Problem z `const`:

```
const TSquare s(2);
TSquare a = s; ← błąd, niejawna referencja do obiektu const
```

Rozwiązanie problemu z `const`:

```
TSquare::TSquare(const TSquare &otherSquare): side(otherSquare.side)
{
}
```

Deklaracja klasy z `const`:

```
class TSquare
{
    public:

        TSquare();
        TSquare(double defaultSide);
        TSquare(const TSquare &otherSquare);

        void setSide(double newSide);
        double getSide() const;
        double area() const;

    private:
        double side;
};
```


Mamy klasę do opisu prostokąta:

```
class TRectangle
{
    public:
        TRectangle();
        TRectangle(double width, double height);
        TRectangle(const TRectangle &otherRectangle);
        TRectangle(const TSquare &otherSquare);

        void setWidth(double width);
        void setHeight(double height);

        double getWidth() const;
        double getHeight() const;

        double area() const;

    private:
        double width, height;
};
```

Definicja konstruktorów i realizatora:

```
TRectangle::TRectangle()  
: width(1), height(1)  
{  
}  
  
TRectangle::TRectangle(double width, double height)  
: width(width), height(height)  
{  
}  
  
TRectangle::TRectangle(const TRectangle &otherRectangle)  
: width(otherRectangle.width), height(otherRectangle.height)  
{  
}  
  
double TRectangle::area() const  
{  
    return width * height;  
}
```

Definicja modyfikatorów i akcesorów:

```
void TRectangle::setWidth(double width)
{
    TRectangle::width = width;
}

void TRectangle::setHeight(double height)
{
    TRectangle::height = height;
}

double TRectangle::getWidth() const
{
    return width;
}

double TRectangle::getHeight() const
{
    return height;
}
```

Definicja konstruktora konwertującego:

```
TRectangle::TRectangle(const TSquare &otherSquare)
: width(otherSquare.getSide()), height(otherSquare.getSide())
{
}
```

Konstruktor konwertujący odpowiada za utworzenie obiektu na podstawie wartości innego typu. Konstruktorem konwertującym jest **każdy** konstruktor, który można wywołać z jednym argumentem – także `TSquare(double)`.

Kontekst aktywowania konstruktora konwertującego:

```
TSquare s(10);

TRectangle r(s);
```

Niejawne konwersje i `explicit`

Kompilator może sam, bez wiedzy programisty, użyć konstruktora konwertującego do konwersji typów.

```
double area(const TSquare &s);
```

```
TSquare s = 10;    ← niejawnie: TSquare(10)
```

```
area(5.0);         ← niejawnie: area(TSquare(5.0))
```

Słowo kluczowe `explicit` zabrania niejawnego użycia konstruktora. Jawne tworzenie obiektu działa jak dotychczas.

```
class TSquare
{
    public:
        explicit TSquare(double side);
        ...
};
```

```
TSquare s = 10;      ← błąd
area(5.0);           ← błąd

TSquare t(10);       ← poprawnie
area(TSquare(5.0));  ← poprawnie
```

Konstruktory jednoargumentowe oznacza się `explicit`, chyba że niejawna konwersja jest zamierzona (np. liczba rzeczywista jest liczbą zespoloną).

Parametr domyślny to wartość określona na etapie **deklaracji funkcji**, która zostanie automatycznie wstawiona do parametru formalnego, jeżeli dana funkcja zostanie wywołana bez odpowiedniego parametru aktualnego.

Parametry domyślne można stosować z metodami klas jak i funkcji niezwiązanych z klasami.

Parametry domyślne muszą być zdefiniowane od końca listy parametrów.

```
void foo(int i, float f = 0, char c = 'A');
```

```
foo(10);
```

```
foo(20, 3.15);
```

```
foo(30, 22.1, 'Z');
```

W przypadku stosowania prototypów funkcji, wartości parametrów domyślnych określa się w prototypie.

```
void foo(int i, float f = 0, char c = 'A');
```

W definicji funkcji parametry domyślne nie występują.

```
void foo(int i, float f, char c)
{
}
```


Konstruktor domyślny jest **bezparametrowy** lub posiada wszystkie parametry będące **parametrami domyślnymi**.

Jednoczesne wystąpienie obu form nie spowoduje błędu kompilacji. Niejednoznaczność pojawia się dopiero przy próbie deklaracji obiektu (`TSquare s;`).

```
class TSquare
{
    public:

        TSquare(double defaultSide = 1);

        void setSide(double newSide);
        double getSide() const;
        double area() const;

    private:
        double side;
};
```

```
TSquare::TSquare(double defaultSide): side(defaultSide)
{
}
```

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie**
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Dziedziczenie porządkuje i wspomaga polimorfizm i enkapsulację dzięki umożliwieniu definiowania i tworzenia specjalizowanych obiektów na podstawie bardziej ogólnych.

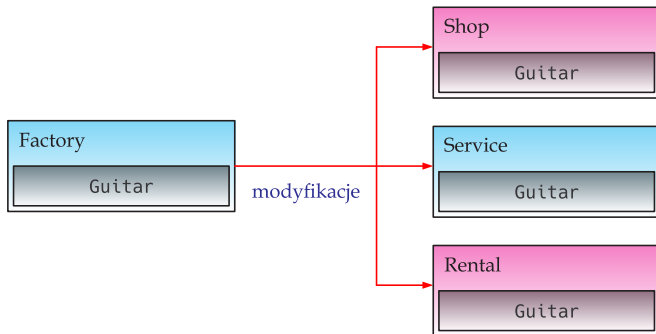
Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy. W typowym przypadku powstają grupy obiektów zwane klasami oraz grupy klas zwane drzewami. Odzwierciedlają one wspólne cechy obiektów.

Dziedziczenie to forma **ponownego wykorzystania** (*reuse*) i pozwala na:

- zwiększenie wydajności procesu programowania,
- poprawia jakość i niezawodność kodu,
- ułatwia rozwój i pielęgnację kodu.

Problemy spotykane przy ponownym wykorzystaniu kodu:

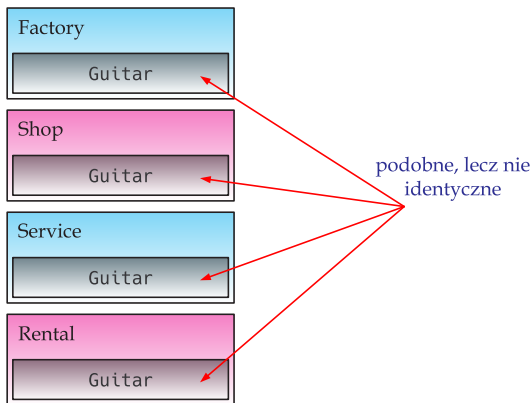
- powtarzalność nieidentycznych obiektów powoduje to, że musimy modyfikować istniejący kod, dostosowując go do specyfiki projektu,
- modyfikacje potrzebne z punktu widzenia jednego systemu mogą być nieużyteczne lub nieakceptowalne z punktu widzenia innego systemu,
- modyfikacje dostosowujące kod do specyfiki systemu zmuszają do utrzymywania różnych wersji tego samego kodu.



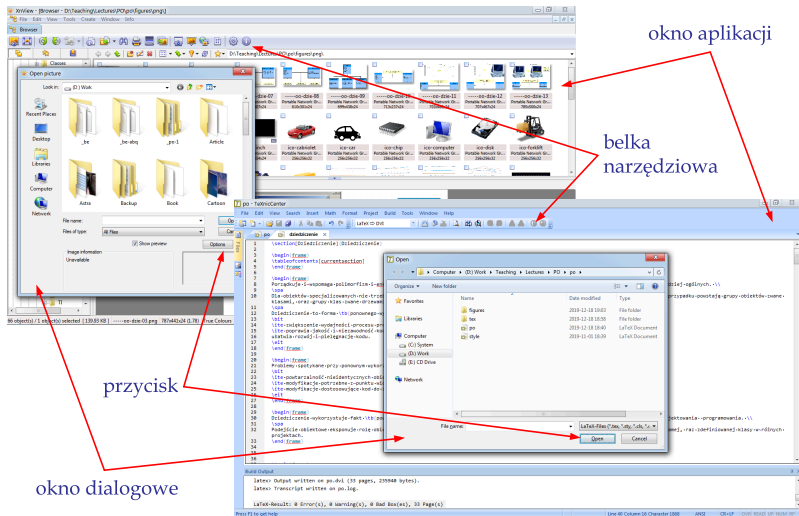
Dziedziczenie wykorzystuje fakt **powtarzalności elementów oprogramowania**. Powtarzalność taką można obserwować na etapie analizy, projektowania i programowania.

Podejście obiektowe eksponuje rolę obiektu. W programowaniu obiektowym ponowne wykorzystanie kodu polega na wykorzystaniu obiektów tej samej, raz zdefiniowanej klasy w różnych projektach.

W obrębie obiektów dziedziny problemu powtarzają się obiekty podobne, lecz bardzo rzadko identyczne.

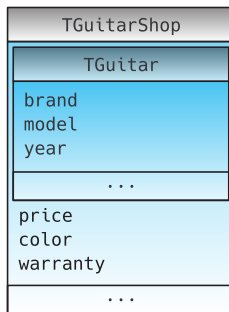


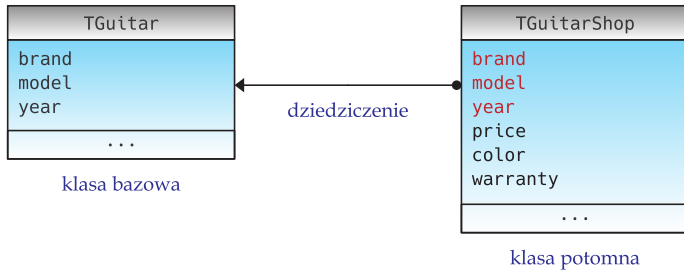
W obrębie obiektów implementacyjnych powtarzalność obiektów identycznych jest znacznie częstsza, co jednak nie jest regułą.

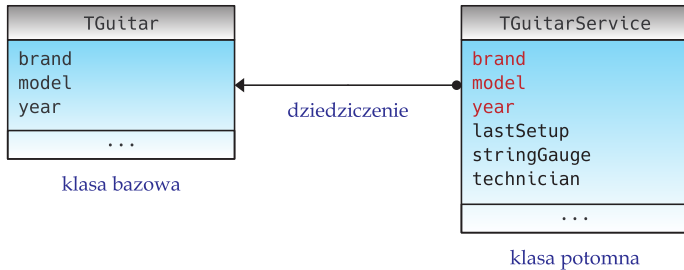


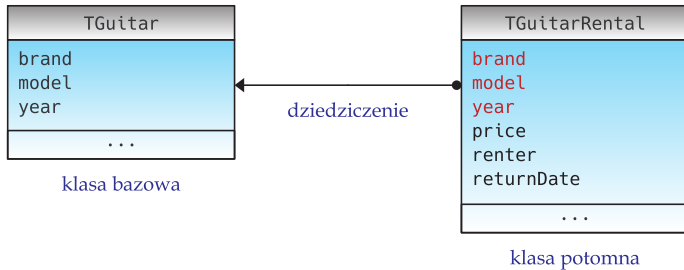
Dziedziczenie polega na **budowaniu nowych klas**, zwanych **klasami potomnymi**, na podstawie **klas już istniejących**, zwanych **klasami bazowymi**.

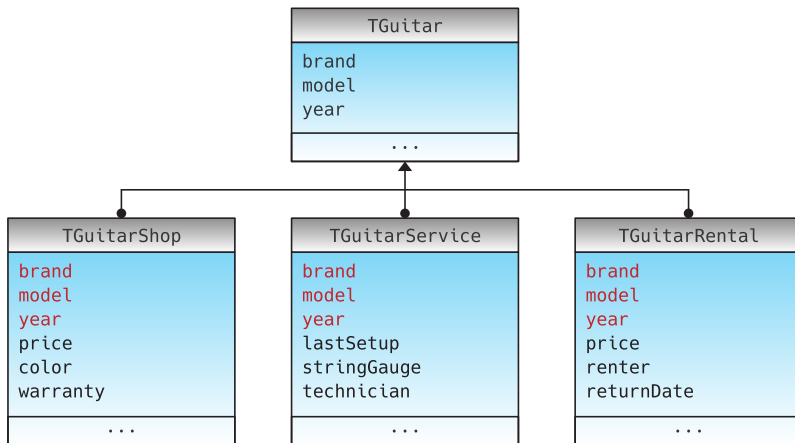
Każda klasa pochodna **dziedziczy wszystkie właściwości** klasy bazowej, **rozszerzając je o nowe pola i metody**.









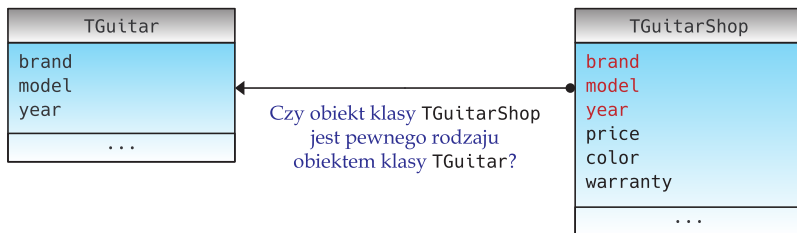


Jak zweryfikować poprawność wykorzystania dziedziczenia?

Reguła **is-a** (*is a kind of*) pozwala na sprawdzenie, czy zachodzi związek dziedziczenia (specjalizacji–generalizacji) pomiędzy klasami.

Realizowane jest to poprzez sprawdzenie poprawności zdania:

- Czy obiekt klasy pochodnej jest pewnego rodzaju (is-a) obiektem klasy bazowej?



Przykład weryfikacji poprawności dziedziczenia.

Zadajemy dwa pytania:

- czy obiekt klasy **A** jest obiektem klasy **B**?
- czy obiekt klasy **B** jest obiektem klasy **A**?

Możliwe odpowiedzi:

- zawsze,
- nigdy,
- czasami.

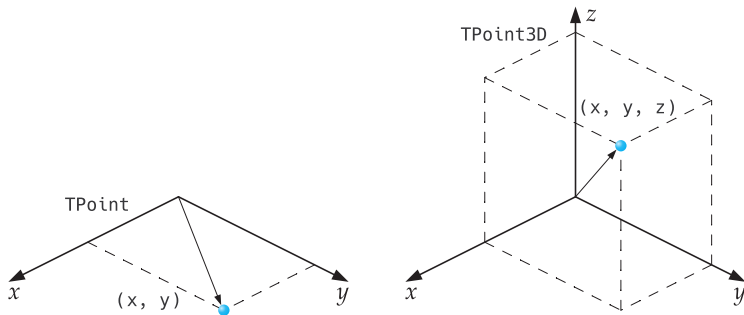
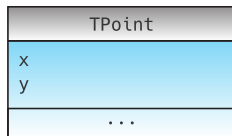
Interpretacja:

- dwie odpowiedzi nigdy: brak związku specjalizacja-generalizacja,
- dwie odpowiedzi zawsze: obiekty **A** i **B** są **synonimiczne** (np. dwie różne nazwy dla tej samej klasy),
- jeżeli: **A is-a B** = zawsze oraz **B is-a A** = czasami, wtedy **A jest specjalizacją B**.

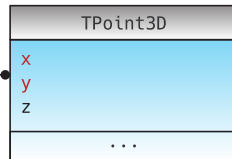
Czy klasa `TGuitarShop` jest klasą pochodną klasy `TGuitar`:

- czy obiekt klasy `TGuitarShop` jest obiektem klasy `TGuitar`? $\Leftarrow$ **zawsze**
- czy obiekt klasy `TGuitar` jest obiektem klasy `TGuitarShop`? $\Leftarrow$ **czasami**

Dziedziczenie w praktyce



Czy obiekt klasy `TPoint3D`
jest pewnego rodzaju
obiektem klasy `TPoint`?



Przykład wykorzystania klasy `TPoint` i `TPoint3D`:

```
TPoint ptOne(2, 3);  
TPoint3D ptTwo(2, 3, 4);  
  
cout << "x = " << ptOne.getX() << endl;  
cout << "y = " << ptOne.getY() << endl;  
  
cout << "x = " << ptTwo.getX() << endl;  
cout << "y = " << ptTwo.getY() << endl;  
cout << "z = " << ptTwo.getZ() << endl;
```

Przykładowa definicja klasy bazowej `TPoint`:

```
class TPoint
{
    public:
        TPoint(): x(0), y(0) {}
        TPoint(int x, int y): x(x), y(y) {}

        void setX(int x) {TPoint::x = x;}
        void setY(int y) {TPoint::y = y;}

        int getX() const {return x;}
        int getY() const {return y;}

    private:
        int x, y;
};
```

Przykładowa klasa pochodna **TPoint3D**:

```
class TPoint3D : public TPoint
{
    public:
        TPoint3D(): TPoint(), z(0) {}
        TPoint3D(int x, int y, int z): TPoint(x, y), z(z) {}

        void setZ(int z) {TPoint3D::z = z;}

        int getZ() const {return z;}

    private:
        int z;
};
```

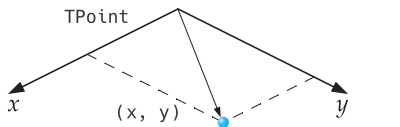
Dziedziczenie w **trybie publicznym** (`public TPoint`) zachowuje widoczność pól określoną w klasie bazowej.

Konstruktor **klasy pochodnej** (`TPoint3D( )`) aktywuje konstruktor **klasy bazowej** (`TPoint( )`) umieszczony na liście inicjalizacyjnej, odbywa się to przed wywołaniem ciała konstruktora klasy pochodnej.

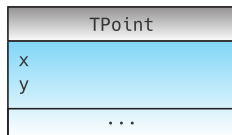
W klasie pochodnej definiujemy metody do obsługi **nowych pól** (`setZ( )` i `getZ( )`), obsługę **pól odziedziczonych** realizujemy z wykorzystaniem metod odziedziczonych.

Używamy konstruktorów (lub listy inicjalizacyjnej) klasy bazowej do zainicjowania pól odziedziczonych (`x` i `y`), nowe pola inicjują własne konstruktory (`z`).

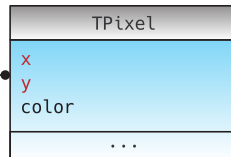
Kolejna klasa pochodna



TPixel



Czy obiekt klasy TPixel
jest pewnego rodzaju
obiektem klasy TPoint?



Przykład klasy pochodnej `TPixel`:

```
class TPixel : public TPoint
{
public:
    TPixel(): TPoint(), color(0) {}
    TPixel(int x, int y, int color): TPoint(x, y), color(color) {}

    void setColor(int color) {TPixel::color = color;}

    int getColor() const {return color;}

    void put() const
    {
        putPixel(getX(), getY(), color); ← przykładowa funkcja z biblioteki graficznej
    }

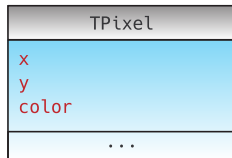
private:
    int color;
};
```

I jeszcze jedna klasa pochodna

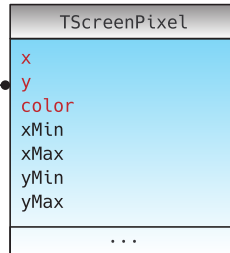
TPixel



TScreenPixel



Czy obiekt klasy TScreenPixel
jest pewnego rodzaju
obiektem klasy TPixel?



Przykład klasy pochodnej TScreenPixel:

```

class TScreenPixel : public TPixel
{
public:
    TScreenPixel(): TPixel() {}
    TScreenPixel(int x, int y, int color): TPixel(x, y, color) {}

    void setX(int x)
    {
        TPoint::setX((x >= xMin && x <= xMax) ? x : 0);
    }

    void setY(int y)
    {
        TPoint::setY((y >= yMin && y <= yMax) ? y : 0);
    }

    static int xMin, xMax, yMin, yMax;
};

int TScreenPixel::xMin = 0;
int TScreenPixel::xMax = 799;
int TScreenPixel::yMin = 0;
int TScreenPixel::yMax = 599;

```


Pola statyczne dla danej klasy występują tylko raz, są współdzielone przez wszystkie obiekty tej klasy. Istnieją nawet wtedy, gdy nie został utworzony żaden obiekt klasy.

Pola statyczne deklaruje się w klasie, a definiuje (i nadaje wartość początkową) poza klasą, w jednym pliku źródłowym. Od C++17 można je zdefiniować bezpośrednio w klasie, poprzedzając słowem `inline`.

```
class TScreenPixel : public TPixel
{
public:
    ...
    inline static int xMin = 0, xMax = 799,      // od C++17: definicja w klasie
                    yMin = 0, yMax = 599;
};
```

Do pól statycznych klasy można odwoływać się bez obiektu:

```
TScreenPixel::xMax = screen.getMaxX();
TScreenPixel::yMax = screen.getMaxY();
```

Wykorzystywane są do zapamiętywania informacji, które są wspólne dla wszystkich obiektów i nie muszą być pamiętane osobno w każdym z obiektów.

Redefinicja metody w klasie pochodnej

```
class TPoint
{
    void setX(int x) {TPoint::x = x;}
};
```

```
class TScreenPixel : public TPixel
{
    void setX(int x)
    {
        TPoint::setX((x >= xMin && x <= xMax) ? x : 0);
    }
};
```

W klasie pochodnej można zmienić sposób działania metody odziedziczonej poprzez jej redefinicję.

Do metody odziedziczonej można dalej odwoływać się stosując nazwę klasy i operator zakresu `::`.

Składowe chronione

W klasie pochodnej nie ma bezpośredniego dostępu do pól prywatnych klasy bazowej.

```
class TPixel : public TPoint
{
    public:
        void put() const
        {
            putPixel(getX(), getY(), color); ← getX(), nie x
        }
};
```

Przewidując, że pewna klasa będzie wykorzystywana jako klasa bazowa, można zadeklarować jej składowe (pola i metody) jako **chronione** `protected`.

Składowe zadeklarowane jako `protected` są dostępne w metodach wszystkich klas pochodnych (tak jak składowe `public`) i niedostępne poza nimi, w szczególności z zewnątrz klasy (tak jak składowe `private`).

Specyfikator `protected` działa więc jak `private`, z tym wyjątkiem, że metody klas pochodnych mają dostęp do składowych `protected` odziedziczonych po klasie bazowej.

Deklaracja pól `x` i `y` jako chronionych:

```
class TPoint
{
    protected:
        int x, y;
};
```

```
class TPixel : public TPoint
{
    public:
        void put() const
        {
            putPixel(x, y, color); ← dozwolone odwołanie
        }
};
```

Metoda klasy pochodnej ma dostęp do składowych `protected` tylko za pośrednictwem obiektów **własnej klasy** (lub klas od niej pochodnych), nie dowolnego obiektu klasy bazowej.

```
class TPixel : public TPoint
{
public:
    void copyXY(const TPixel &p)
    {
        x = p.x;  ← poprawnie, p jest obiektem klasy TPixel
        y = p.y;
    }
    void copyXY(const TPoint &p)
    {
        x = p.x;  ← błąd, p może być np. obiektem TPoint3D
        y = p.y;
    }
};
```

Gdyby było inaczej, dowolna klasa mogłaby uzyskać dostęp do chronionych składowych obcego obiektu, po prostu dziedzicząc po klasie bazowej.

Z zewnątrz klasy składowe `protected` są niedostępne, także dla obiektów klas pochodnych: `pixel.x = 5;` jest błędem.

Destruktory

Destruktor to funkcja wywoływana automatycznie przez kompilator bezpośrednio przed momentem, w którym obiekt przestaje istnieć.

Moment *śmierci* obiektu zależy od tego, w jaki sposób został on utworzony.

Obiekty **statyczne** giną po zakończeniu wykonania funkcji `main()`, obiekty **automatyczne** po wyjściu sterowania z bloku, w którym zostały zdefiniowane, obiekty **dynamiczne** w momencie usunięcia ich z pamięci operatorem `delete`.

Destruktor to bezparametrowa funkcja, bez określonego rezultatu, o nazwie takiej, jak nazwa klasy poprzedzona znakiem tyldy `~`.

```
class TPoint
{
public:
    ~TPoint();
};
```

```
class TScreenPixel
{
public:
    ~TScreenPixel();
};
```

Klasa `TPoint` ze zmodyfikowanymi konstruktorem i destruktor:

```
class TPoint
{
public:
    TPoint(int x, int y): x(x), y(y)
    {
        cout << "TPoint(" << x << ", " << y << ")" << endl;
    }

    ~TPoint()
    {
        cout << "~TPoint()" << endl;
    }
};
```

Aktywowanie konstruktora i destruktor dla obiektu klasy `TPoint`:

```
cout << "Przed utworzeniem obiektu pt klasy TPoint." << endl;
{
    TPoint pt(2, 3);
    cout << endl << "Punkt ma wspolrzedne:" << endl;
    cout << "x = " << pt.getX() << endl;
    cout << "y = " << pt.getY() << endl;
}
cout << endl << "Po usunieciu obiektu pt klasy TPoint." << endl;
```


Przed utworzeniem obiektu pt klasy TPoint.

TPoint(2, 3)

Punkt ma współrzędne:

x = 2

y = 3

~TPoint()

Po usunięciu obiektu pt klasy TPoint.

Klasa `TPoint3D` ze zmodyfikowanymi konstruktorem i destruktor:

```
class TPoint3D : public TPoint
{
    public :
        TPoint3D(int x, int y, int z): TPoint(x, y), z(z)
        {
            cout << "TPoint3D(" << x << ", " << y << ", " << z << ")" << endl;
        }

        ~TPoint3D()
        {
            cout << "~TPoint3D()" << endl;
        }
};
```

Aktywowanie konstruktora i destruktor dla obiektu klasy `TPoint3D`:

```
cout << "Przed utworzeniem obiektu pt klasy TPoint3D." << endl;
{
    TPoint3D pt(2, 3, 4);
    cout << endl << "Punkt ma wspolrzedne:" << endl;
    cout << "x = " << pt.getX() << endl;
    cout << "y = " << pt.getY() << endl;
    cout << "z = " << pt.getZ() << endl;
}
cout << endl << "Po usunieciu obiektu pt klasy TPoint3D." << endl;
```

Przed utworzeniem obiektu pt klasy TPoint3D.

```
TPoint(2, 3)
```

```
TPoint3D(2, 3, 4)
```

Punkt ma współrzędne:

```
x = 2
```

```
y = 3
```

```
z = 4
```

```
~TPoint3D()
```

```
~TPoint()
```

Po usunięciu obiektu pt klasy TPoint3D.

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm**
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Polimorfizm

Słowo polimorfizm pochodzi od dwóch greckich słów: *poly* czyli **wiele** i *morphos* czyli **postać**.

Polimorfizm oznacza zatem wielopostaciowość. Polimorfizm w programowaniu obiektowym oznacza zdolność obiektów do różnych zachowań, w zależności od bieżącego kontekstu wykonania programu.

Referencje i kolekcje obiektów mogą dotyczyć obiektów różnego typu, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego.

Inaczej mówiąc, są to mechanizmy pozwalające na używanie wartości, zmiennych i procedur na kilka różnych sposobów (wyabstrahowanie wyrażeń od konkretnych typów).

W języku C++ wyróżnia się polimorfizm **statyczny** (przeciążanie funkcji i operatorów, szablony) oraz **dynamiczny** (funkcje wirtualne). Ten rozdział dotyczy polimorfizmu dynamicznego.

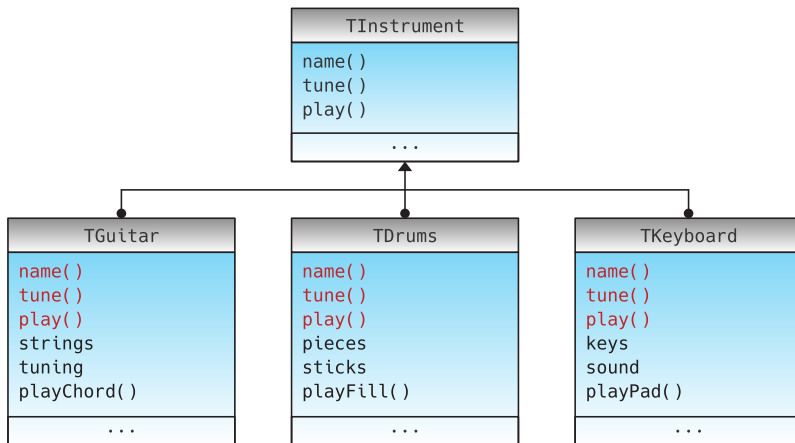
To jaka ma być **wykonana akcja**, ustalane jest na **etapie wykonania** programu, a wykonywane akcje mogą być różne.

W języku C++ wykonanie akcji polega na wywołaniu odpowiedniej funkcji. Polimorfizm w tym języku polega na tym, że wywoływane mogą być różne wersje tej samej funkcji.

Różne wersje tej samej funkcji **powstają w trakcie dziedziczenia**, kiedy to klasy pochodne w specjalny sposób redefiniują pewną funkcję składową.

Zdyscyplinowane stosowanie zasad abstrakcji, hermetyzacji i dziedziczenia prowadzi do powstania, często rozbudowanych, hierarchii klas.

Obiekty takich klas są często do siebie podobne, ale nie jednakowe. Problem stanowi spójne i wygodne zarządzanie grupami obiektów różnych klas należących do tej samej hierarchii, zdefiniowanej dziedziczeniem.



Obiekty spowinowacone mogą być traktowane w specjalny sposób.

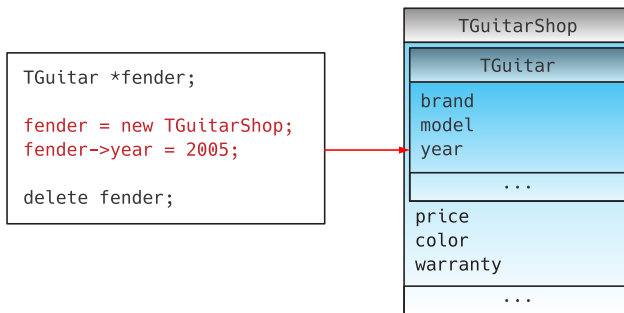
```
TInstrument *ins;  
  
ins = new TGuitar;  
ins->tune();  
  
delete ins;  
  
ins = new TDrums;  
ins->tune();  
  
delete ins;
```

```
void pointInfo(TPoint &pt)  
{  
    cout << "x = " << pt.getX() << endl;  
    cout << "y = " << pt.getY() << endl;  
}  
  
TPixel ptOne(2, 3, RED);  
TPoint3D ptTwo(2, 3, 4);  
  
pointInfo(ptOne);  
pointInfo(ptTwo);
```

W języku C++ wolno do **wskaźnika klasy bazowej** przypisać **wskazanie obiektu klasy pochodnej**. Analogicznie jest w przypadku referencji.

Obiekt klasy pochodnej jest przecież pewnego rodzaju obiektem klasy bazowej (jest specjalizacją klasy bazowej), posiada zatem wszystkie pola i funkcje składowe.

Wolno zatem pośrednio odwoływać się do odziedziczonych pól i funkcji składowych.

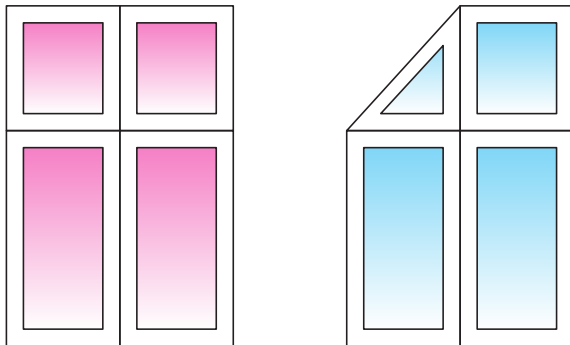


Polimorfizm w praktyce

Program wspomagający pracę technologa w firmie produkującej okna.

Zadaniem programu jest obliczanie:

- łącznego pola powierzchni wszystkich skrzydeł okna,
- przybliżonej, łącznej długości profili, użytych do produkcji każdego ze skrzydeł.



Stosując zasadę abstrakcji wyodrębniamy najistotniejsze cechy obiektów dla rozpatrywanego zagadnienia – szyby to figury geometryczne a okno to ich złożenie:

- łączna powierzchnia okna to, w przybliżeniu, suma pól figur opisujących szyby,
- łączna długość profili to, w przybliżeniu, suma obwodów figur opisujących szyby.

Realizacja programu sprowadza się do obliczeń pól powierzchni i obwodów obiektów, będących złożeniem elementarnych figur płaskich.

Nie powinniśmy mieć problemów z utworzeniem klas reprezentujących figury płaskie. Nie wiemy jak reprezentować i pracować z ich złożeniem.

Rozpoczniemy od utworzenia nieco *dziwnej* klasy – reprezentującej abstrakcyjną figurę geometryczną. Wyposażamy ją w funkcje obliczania pola i długości obwodu.

```
class TFigure
{
    public:
        TFigure() {}

        double area() const {return 0;}
        double perimeter() const {return 0;}
};
```

Klasa `TFigure` służyć będzie jak klasa bazowa dla specjalizowanych klas pochodnych, reprezentujących konkretne figury geometryczne.

Jej istotą jest stwierdzenie, że każda figura geometryczna powinna umieć wyznaczać swoje pole i obwód, w charakterystyczny dla siebie sposób.

Zatem każda z klas pochodnych, powinna redefiniować funkcje `area()` i `perimeter()`, w odpowiedni dla tych klas sposób.

Klasa reprezentująca **kwadrat** będzie teraz klasą pochodną w stosunku do klasy **TFigure**:

```
class TSquare : public TFigure
{
public:
    TSquare();
    TSquare(double side);

    void setSide(double side);
    double getSide() const;

    double area() const;
    double perimeter() const;

private:
    double side;
};
```

Redefinicja funkcji `area()` i `perimeter()` – obliczenia dla kwadratu:

```
double TSquare::area() const {return side * side;}
double TSquare::perimeter() const {return 4 * side;}
```

Klasa reprezentująca **prostokąt** będzie klasą pochodną w stosunku do klasy **TFigure**:

```
class TRectangle : public TFigure
{
public:
    TRectangle();
    TRectangle(double width, double height);

    void setWidth(double width);
    void setHeight(double height);
    double getWidth() const;
    double getHeight() const;

    double area() const;
    double perimeter() const;

private:
    double width, height;
};
```

Redefinicja funkcji `area()` i `perimeter()` – obliczenia dla prostokąta:

```
double TRectangle::area() const {return width * height;}
double TRectangle::perimeter() const {return 2 * width + 2 * height;}
```

Klasa reprezentująca **trójkąt** będzie klasą pochodną w stosunku do klasy **TFigure**:

```
class TTriangle : public TFigure
{
public:
    TTriangle();
    TTriangle(double base, double height);

    void setBase(double base);
    void setHeight(double height);
    double getBase() const;
    double getHeight() const;

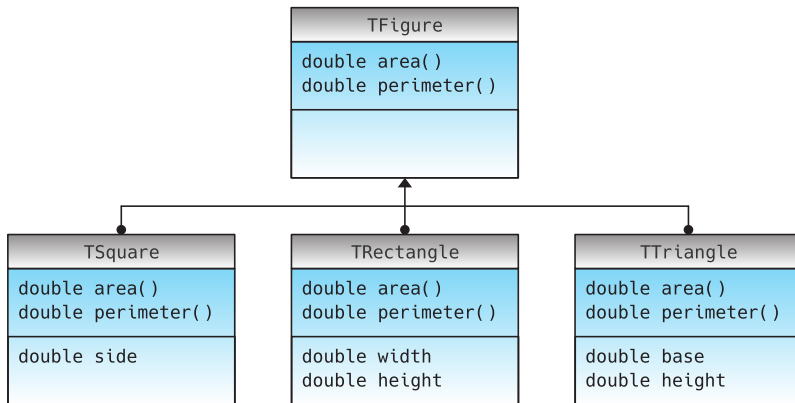
    double area() const;
    double perimeter() const;

private:
    double base, height;
};
```

Redefinicja funkcji **area()** i **perimeter()** – obliczenia dla trójkąta:

```
double TTriangle::area() const {return 0.5 * base * height;}
double TTriangle::perimeter() const {return sqrt(base * base +
    height * height) + base + height;}
```


Hierarchia klas figur płaskich



Założmy, że zdefiniowaliśmy wskaźnik do klasy bazowej `TFigure` i obiekty klas pochodnych:

```
TFigure *fig;  
  
TSquare s(10);  
TRectangle r(10, 20);  
TTriangle t(10,10);
```

Wiemy już, że można przypisać do wskaźnika `fig` wskazanie na obiekt klasy pochodnej:

```
fig = &s;  
fig = &r;  
fig = &t;
```

Wolno zatem pośrednio odwoływać się do odziedziczonych pól i funkcji składowych (czy na pewno tak jak się spodziewamy?).

```
fig = &s;  
  
cout << "Pole kwadratu wynosi: " << fig->area() << endl;  
cout << "Obwód kwadratu wynosi: " << fig->perimeter() << endl;
```

Ale nie można (dlaczego?):

```
fig->setSide(100); ← niedozwolone
```

Choć można dokonać *wymuszenia* stosując brutalne rzutowanie typów (niezalecane):

```
((TSquare *)fig)->setSide(100);
```

Wykorzystując wskaźnik do **klasy bazowej** można odwoływać się w legalny sposób do wszystkich odziedziczonych składowych **obiektu pochodnego**.

Stosując zwykłe rzutowanie (lub specjalne metody konwersji) można siłowo wymusić dostęp do składowych zdefiniowanych w klasie pochodnej.

W C++ istnieją specjalne operatory rzutowania, bardziej czytelne i ściślej kontrolowane przez kompilator niż notacja nawiasowa z języka C (więcej na końcu rozdziału).

Rozbudujmy klasy o dodatkową funkcję określającą nazwę figury:

```
class TFigure
{
    public:
        string getName() const {return "Figura";}
};
```

```
class TSquare : public TFigure
{
    public:
        string getName() const {return "Kwadrat";}
};
```

```
class TRectangle : public TFigure
{
    public:
        string getName() const {return "Prostokat";}
};
```

```
class TTriangle : public TFigure
{
    public:
        string getName() const {return "Trojkat";}
};
```

Napiszmy uniwersalną funkcję wyświetlającą informacje o figurze:

```
void figureInfo(TFigure *fig)
{
    cout << fig->getName() << endl;
    cout << "Pole: " << fig->area() << endl;
    cout << "Obwod: " << fig->perimeter() << endl;
}
```

Stwórzmy obiekty klas pochodnych od klasy `TFigure`:

```
TSquare s(10);
TRectangle r(10, 20);
TTriangle t(10,10);
```

I w końcu wywołujemy `figureInfo( )`:

```
figureInfo(&s);  
figureInfo(&r);  
figureInfo(&t);
```

I otrzymujemy...

```
Figura  
Pole: 0  
Obwod: 0
```

```
Figura  
Pole: 0  
Obwod: 0
```

```
Figura  
Pole: 0  
Obwod: 0
```

Nie działa!

Ale dlaczego?

Przypisanie konkretnego ciała funkcji do wywołania nazywa się wiązaniem (*binding*).

W języku C++ występują dwa rodzaje wiązania:

- **Wczesne wiązanie** (*early binding*) polega na przypisaniu konkretnej funkcji każdemu wywołaniu już na etapie kompilacji programu. Inna nazwa – **wiązanie statyczne** (*static binding*).
- **Późne wiązanie** (*late binding*) polega na przypisaniu konkretnej funkcji każdemu wywołaniu dopiero na etapie wykonania programu, w zależności od typu obiektu, którego wywołanie dotyczy. Inna nazwa to **dynamiczne wiązanie** (*dynamic binding*).

Jakie wiązanie stosuje kompilator?

```
TFigure *fig;
TSquare s(10);

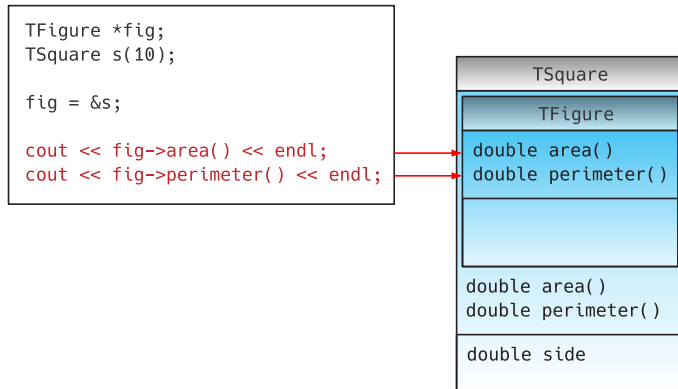
fig = &s;
cout << "Pole kwadratu wynosi: " << fig->area() << endl;
cout << "Obwod kwadratu wynosi: " << fig->perimeter() << endl;
```

Wywołanie odbywa się za pośrednictwem wskaźnika do typu `TFigure`. Kompilator **przygląda** się definicji klasy i deklaracji wywoływanej funkcji:

```
class TFigure
{
public:
    string getName() const {return "Figura";}
    double area() const {return 0;}
    double perimeter() const {return 0;}
};
```

Wywoływane funkcje są **zwykłymi** funkcjami. Takie funkcje są łączone statycznie. Kompilator wstawia wywołanie funkcji zdefiniowanych w klasie występującej w deklaracji wskaźnika `fig`. Wywoływane są zatem funkcje z klasy `TFigure`, niezależnie od klasy wskazywanego obiektu.

O tym która funkcja jest wywoływana decyduje kompilator na etapie kompilacji. Wersję funkcji determinuje **typ występujący w deklaracji zmiennej wskaźnikowej**.



Przyczyną problemów jest **wiązanie statyczne**. Mimo że funkcja wywoływana jest dla obiektów klas potomnych względem **TFigure**, kompilator wstawi wywołanie funkcji zgodnie z typem występującym w deklaracji parametru **fig**.

Dokonajmy drobnej modyfikacji deklaracji funkcji składowych klasy `TFigure`:

```
class TFigure
{
    public:
        TFigure() {}

        virtual string getName() const {return "Figura";}

        virtual double area() const {return 0;}
        virtual double perimeter() const {return 0;}
};
```

Uruchamiamy ponownie...

```
Kwadrat  
Pole: 100  
Obwod: 40  
  
Prostokat  
Pole: 200  
Obwod: 60  
  
Trojkat  
Pole: 50  
Obwod: 34.1421
```

Działa poprawnie!

Funkcje wirtualne

Mimo iż typ występujący w deklaracji wskaźnika to `TFigure`, wywołane zostaną funkcje należące do klasy **obektu wskazywanego** przez wskaźnik `fig`. Przy wiązaniu **dynamicznym** istotny jest typ obiektu **wskazywanego**!

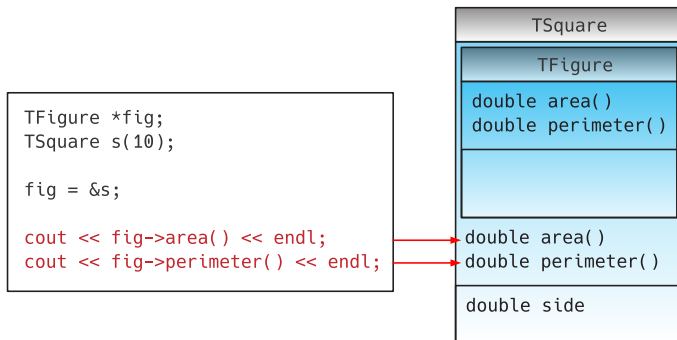
Wiązanie dynamiczne jest realizowane w języku C++ za pomocą funkcji **wirtualnych**. Funkcja wirtualna posiada w swojej deklaracji słowo kluczowe `virtual`.

Jeżeli w klasie pochodnej funkcja wirtualna nie zostanie przeddefiniowana, wszystko działa jak w przypadku innych funkcji. Jeżeli w klasie pochodnej funkcja wirtualna zostanie przeddefiniowana, to zostanie ona wywołana, gdy wskaźnik do klasy bazowej wskazuje na obiekt klasy pochodnej.

Wiązanie dynamiczne

O tym która funkcja jest wywoływana decyduje **typ obiektu wskazywanego**.

Funkcja wiązana dynamicznie musi być zadeklarowana jako wirtualna. Wystarczy, że słowo kluczowe **virtual** wystąpi w deklaracji klasy bazowej.



Specyfikator override

Funkcja w klasie pochodnej nadpisuje funkcję wirtualną tylko wtedy, gdy ma **identyczną** sygnaturę – łącznie z `const`. Drobna pomyłka tworzy nową, niezależną funkcję:

```
class TSquare : public TFigure
{
public:
    double area() {return side * side;} ← brak const - nowa funkcja
    ...
};
```

```
TFigure *fig = new TSquare(10);
cout << fig->area() << endl; ← 0 - wywołana TFigure::area()
```

Program się kompiluje i znów **nie działa**, tym razem bez widocznej przyczyny.

Specyfikator `override` (od C++11) informuje kompilator, że funkcja **ma** nadpisywać funkcję wirtualną klasy bazowej. Jeżeli niczego nie nadpisuje, zgłaszany jest błąd.

```
double area() override {return side * side;} ← błąd kompilacji
```

```
double area() const override {return side * side;} ← poprawnie
```


Klasa abstrakcyjna

Czysta wirtualność określa to, że metoda wirtualna z klasy bazowej nigdy nie powinna się wykonać.

W efekcie klasa taka staje się **klasą abstrakcyjną**. Oznacza to, że nie jest możliwe stworzenie obiektu tej klasy.

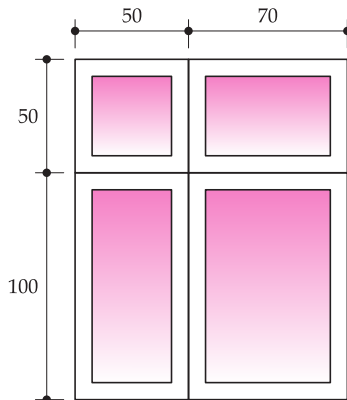
Klasa służy jedynie temu, aby zdefiniować pewnego rodzaju interfejs i jest przeznaczona jedynie po to, by od niej dziedziczyć.

```
class TFigure
{
    public:
        virtual string getName() const = 0;

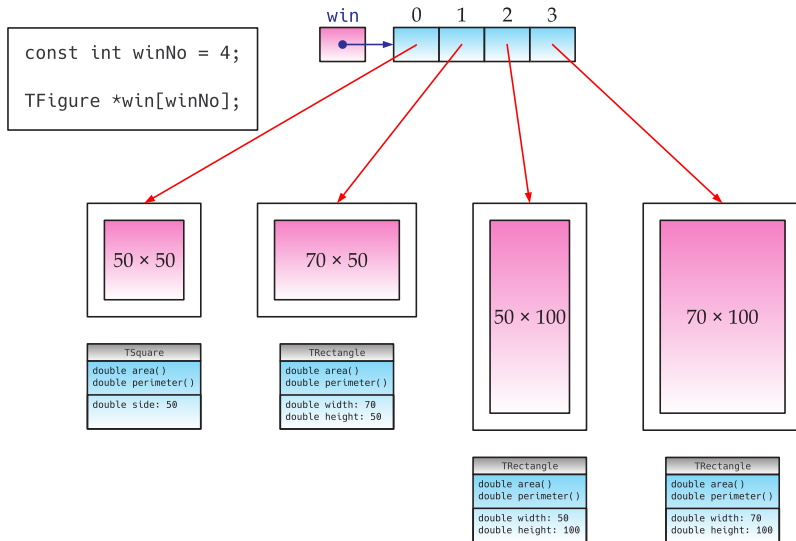
        virtual double area() const = 0;
        virtual double perimeter() const = 0;
};
```

Wykorzystanie późnego wiązania

Znany jest układ okna i wymiary skrzydeł. Jedno skrzydło jest kwadratowe, pozostałe są prostokątne. Należy wyznaczyć łączną powierzchnię szyb i długość wykorzystanych profili.



Reprezentowanie informacji o oknach



Definiowanie elementów, obliczenia, zwalnianie pamięci

```
const int winNo = 4; ← określenie liczby elementów

TFigure *win[winNo]; ← tablica wskaźników na elementy okna

win[0] = new TSquare(50); ← przydział pamięci
win[1] = new TRectangle(70, 50);
win[2] = new TRectangle(50, 100);
win[3] = new TRectangle(70, 100);

calcAndInfo(win, winNo); ← obliczenia i drukowanie wyników

for(int i = 0; i < winNo; delete win[i++]); ← zwalnianie pamięci
```

Funkcja calcAndInfo()

```
void calcAndInfo(TFigure *win[], int winNo)
{
    double totalArea = 0;
    double totalPerimeter = 0;

    for(int i = 0; i < winNo; i++)
    {
        totalArea += win[i]->area();
        totalPerimeter += win[i]->perimeter();
    }

    cout << "Powierzchnia szyb: " << totalArea << endl;
    cout << "Długość profili: " << totalPerimeter << endl;
}
```

Wirtualny destruktor

Program działa poprawnie, jednak zwalnianie pamięci przez wskaźnik typu `TFigure *` wywołuje wyłącznie destruktor **klasy bazowej**. Dstruktor jest wiązany tak samo jak inne funkcje – statycznie, zgodnie z typem wskaźnika.

W przypadku figur nie ma to znaczenia, ponieważ przechowują one wyłącznie liczby i ich destruktory nie mają nic do zrobienia.

Co jednak, jeśli obiekt klasy pochodnej zarządza jakimś zasobem, który trzeba zwolnić – pamięcią, plikiem czy połączeniem?

Gitara przy tworzeniu włącza wzmacniacz, a przy usuwaniu powinna go wyłączyć.

```
class TInstrument
{
public:
    ~TInstrument() {cout << "~TInstrument()" << endl;}
    virtual void play() const = 0;
};

class TGuitar : public TInstrument
{
public:
    TGuitar() {cout << "Wzmacniacz: ON" << endl;}
    ~TGuitar() {cout << "Wzmacniacz: OFF" << endl;}
    void play() const {cout << "E5" << endl;}
};
```

```
TInstrument *ins = new TGuitar;
ins->play();
delete ins;
```

```
Wzmacniacz: ON
E5
~TInstrument()
```

Wzmacniacz nie został wyłączony!

Wystarczy, że destruktor klasy bazowej będzie wirtualny:

```
class TInstrument
{
public:
    virtual ~TInstrument() {cout << "~TInstrument()" << endl;}
    virtual void play() const = 0;
};
```

```
Wzmacniacz: ON
E5
Wzmacniacz: OFF
~TInstrument()
```

Teraz o wyborze destruktora decyduje typ obiektu wskazywanego. Wywoływany jest destruktor klasy pochodnej, a po nim – jak zawsze – destruktor klasy bazowej.

Klasa posiadająca funkcje wirtualne powinna mieć **wirtualny destruktor**.

Poprawiona klasa bazowa figur:

```
class TFigure
{
public:
    virtual ~TFigure() = default;

    virtual string getName() const = 0;

    virtual double area() const = 0;
    virtual double perimeter() const = 0;
};
```

Od teraz zwalnianie elementów okna `delete win[i]` jest poprawne niezależnie od tego, co w przyszłości zostanie dodane do klas figur.

Destruktor `TFigure` nie ma nic do zrobienia, dlatego zamiast pisać puste ciało, można zażądać od kompilatora wersji domyślnej `= default`.

Zapis `= default` oznacza **wygeneruj standardową wersję**. Można go stosować także do **konstruktorów** i **operatorów przypisania**.

Pozornie to samo daje pusty destruktork:

```
| ~TFigure() = default; ← destruktork generowany przez kompilator
```

```
| ~TFigure() {} ← destruktork napisany przez programistę
```

Różnica polega na tym, że destruktork `{}` jest zawsze traktowany jak **napisany przez programistę**, nawet gdy jest pusty. Klasa przestaje być wtedy **trywialnie niszczalna** – kompilator nie może pominąć wywołania destruktorka, a obiektów nie wolno kopiować bajt po bajcie (np. `memcpy()`).

Różnica dotyczy jednak tylko destruktorków niewirtualnych. Destruktork wirtualny nigdy nie jest trywialny, niezależnie od zapisu, więc w przypadku `TFigure` oba warianty działają identycznie. `default` lepiej jednak oddaje intencję – **nic ponad standard**.

Ręczne zarządzanie pamięcią przez tablicę wskaźników dobrze pokazuje mechanizm polimorfizmu, ale współcześnie tak się nie pisze – łatwo zapomnieć o `delete` albo usunąć obiekt dwukrotnie.

W praktyce stosuje się kontenery biblioteki standardowej i **inteligentne wskaźniki**, które same zwalniają pamięć (nagłówki `vector` i `memory`):

```
vector<unique_ptr<TFigure>> win;  
  
win.push_back(make_unique<TSquare>(50));  
win.push_back(make_unique<TRectangle>(70, 50));  
win.push_back(make_unique<TRectangle>(50, 100));  
win.push_back(make_unique<TRectangle>(70, 100));  
  
for(const auto &w: win)  
    cout << w->area() << endl;  
  
← bez delete - pamięć zwolni się sama
```

Wirtualny destruktor jest tu równie potrzebny – `unique_ptr` usuwa obiekt przez wskaźnik do klasy bazowej. Kontenery zostaną omówione przy bibliotece STL.

Rzutowanie w C++

Rzutowanie w notacji nawiasowej (`typ`)**wyrażenie** wykona **każdą** konwersję, jaką da się przeprowadzić – także taką, której programista nie zamierzał. Kompilator nie ostrzeże, że zamiast zmiany typu liczbowego nastąpiła reinterpretacja bitów czy usunięcie `const`.

W C++ wprowadzono cztery operatory, każdy dla innego rodzaju konwersji:

operator, zastosowanie, kontrola		
<code>static_cast<>()</code>	konwersje typów liczbowych, konwersje w hierarchii klas (w górę i w dół)	kompilacja
<code>dynamic_cast<>()</code>	konwersje w dół hierarchii klas polimorficznych	wykonanie
<code>const_cast<>()</code>	dodanie lub usunięcie <code>const</code>	kompilacja
<code>reinterpret_cast<>()</code>	reinterpretacja bitów: wskaźnik na inny typ, liczba na wskaźnik	brak

Operatory te nie są bezpieczniejsze same w sobie, ale **jawnie wyrażają intencję**, kompilator odrzuca konwersje nienależące do danego rodzaju, a każde rzutowanie łatwo odnaleźć w kodzie.

`static_cast` – konwersje **rozsądne**, sprawdzane na etapie kompilacji:

```
double d = 3.7;
int n = static_cast<int>(d); ← zamiast (int)d
```

```
TFigure *fig = new TRectangle(50, 70);
TSquare *sq = static_cast<TSquare *>(fig); ← kompiluje się, ale fig nie jest kwadratem
```

Przy konwersji w dół hierarchii `static_cast` wierzy programiście na słowo – nie sprawdza, czy obiekt wskazywany jest naprawdę danego typu.

`reinterpret_cast` – dostęp do surowej reprezentacji danych, np. bajtów liczby:

```
float f = 123.321;
unsigned char *b = reinterpret_cast<unsigned char *>(&f);

for(int i = 0; i < sizeof(f); i++)
    cout << hex << int(b[i]) << ' '; ← 5a a4 f6 42
```

Wynik zależy od platformy (kolejność bajtów), dlatego `reinterpret_cast` jest najmniej bezpieczny ze wszystkich.

`dynamic_cast` – konwersja w dół hierarchii **sprawdzana w trakcie działania programu**. Jeżeli obiekt wskazywany nie jest wskazanego typu, rezultatem jest `nullptr`.

Działa tylko dla klas **polimorficznych**, czyli posiadających choć jedną funkcję wirtualną – typ obiektu musi być znany w czasie wykonania.

```
for(int i = 0; i < winNo; i++)
{
    TSquare *sq = dynamic_cast<TSquare *>(win[i]);

    if(sq != nullptr)
        cout << "Kwadrat o boku " << sq->getSide() << endl;
    else
        cout << "To nie kwadrat" << endl;
}
```

```
Kwadrat o boku 50
To nie kwadrat
To nie kwadrat
To nie kwadrat
```

Konieczność częstego używania `dynamic_cast` zwykle oznacza, że brakuje funkcji wirtualnej w klasie bazowej.

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa**
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Błędy i ich obsługa

W trakcie działania programu zawsze może dojść do powstania nietypowej sytuacji. Nie da się ich całkowicie wyeliminować. Można jednak przewidywać, iż taka sytuacja wystąpi i się na nią odpowiednio przygotować.

Kontrolowanie poprawności wykonywanych operacji jest możliwe, jednak uciążliwe bez specjalnie przygotowanych do tego celu technik (przykład w C).

Uciążliwość pisania sekwencji kontroli i obsługi błędów budzi ochotę do użycia instrukcji skoku `goto`, ale jest to co najmniej **bardzo nieeleganckie**.


```
if((file = fopen(name, "rt")) == NULL)
    return IN_FILE_ERROR;
else

    if((num_of_lines = count_lines(file)) == 0)
        return EMPTY_IN_FILE;
    else

        if((lines = malloc(num_of_lines * sizeof(char *))) == NULL)
            return NO_MEM_ERROR;
        else

            if((buffer = malloc(MAX_LINE_LEN)) == NULL)
                return NO_MEM_ERROR;
            else

                while(fgets(buffer, MAX_LINE_LEN, file) != NULL)
                {
                    ...
                }

    ...
```

W języku C przewidziano zestaw funkcji i makr do kontroli błędów i reagowania na ich powstanie:

- `assert( )`,
- `signal( )`,
- `raise( )`,
- `setjmp( )`,
- `longjmp( )`.

Są one jednak rzadko stosowane – mało znane i wymagające uwagi.

Użycie sygnałów i nielokalnych skoków jest niebezpieczne w C++, nie zapewnia bowiem właściwego aktywowania np. destruktorów.

Błąd czyli sytuacja wyjątkowa

W języku C++ wprowadzono mechanizm pozwalający programiście na reagowanie na sytuacje błędne czy nietypowe – programista może wygenerować **wyjątek**.

Wyjątek (*exception*) to obiekt pewnej klasy. Wygenerowanie wyjątku polega na przekazaniu obiektu opisującego wyjątek z fragmentu kodu, w którym wystąpił problem, do fragmentu, w którym przewidziano jego obsługę.

```
class itemsError  
{  
  
};
```

Wygenerowanie wyjątku powoduje przerwanie wykonywania sprawiającego problemy kodu i przejście do obsługi sytuacji problematycznej. Obsługa ta może znajdować się w innym miejscu kodu.

```
int getItems()  
{  
    int numOfItems;  
    cin >> numOfItems;  
  
    if(numOfItems <= 0)  
        throw itemsError();  
  
    ...  
}
```

Wyjątek jest obiektem, jego klasa określa typ sytuacji wyjątkowej. Obiekt może w sobie posiadać pola oraz funkcje składowe, pozwalające na sprecyzowanie informacji o zaistniałej sytuacji wyjątkowej.

```
try
{
    getItem();
}

catch( itemsError )
{
    cout << "Bład!";
}
```

Jeśli wyjątek nie zostanie obsłużony, to program zostanie awaryjnie zakończony, dalsza część programu się nie wykona.

Zgłaszanie wyjątków

Wyjątki są generowane instrukcją `throw`. Po słowie kluczowym `throw` występuje dowolne wyrażenie pewnego typu. Wartość tego wyrażenia jest zgłaszana jako wyjątek. Wyjątki mogą być również typu wbudowanego.

```
if(numOfItems <= 0)
    throw 1;
```

```
if(numOfItems <= 0)
    throw "Liczba ujemna";
```

```
try
{
    getItems();
}

catch(int)
{
    cout << "Bład typu int";
}
```

```
try
{
    getItems();
}

catch(char const *)
{
    cout << "Bład typu char const *";
}
```

Tak zdefiniowana obsługa jest wrażliwa na różne typy wyjątków. W obrębie danego typu wyjątku, różne jego wartości nie są rozróżniane.

Co się dzieje po zgłoszeniu wyjątku?

Po wygenerowaniu wyjątku instrukcją `throw`, biblioteka czasu wykonania (*RTL*, *run time library*) wykonuje następujące kroki:

- pobiera wyjątek, określa jego typ,
- przeszukuje stos wywołań funkcji w poszukiwaniu takiej, która zawiera obsługę wyjątku tego typu (czyli odpowiednią instrukcję `catch`),
- jeżeli odpowiednia obsługa wyjątku zostanie znaleziona, stos jest w tym miejscu rozwijany (*unwind*) i wyjątek jest obsługiwany,
- jeżeli w trakcie tych poszukiwań nie zostanie znaleziony pasujący blok obsługi wyjątku, dochodzimy do punktu wejściowego programu, czyli funkcji `main()`,
- jeżeli tutaj też nie ma obsługi zgłoszonego wyjątku, program zostanie przerwany w trybie awaryjnym,
- w trakcie przechodzenia do kolejnych bloków na stosie wywołań, usuwane są wszystkie obiekty automatyczne, czemu towarzyszy aktywowanie ich destruktorów (jeżeli wyjątek nie zostanie obsłużony nigdzie, rozwinięcie stosu nie jest gwarantowane).

Propagacja wyjątku kończy się:

- jego obsługą w odpowiednim bloku **catch**, po czym wykonane zostaną kolejne instrukcje następujące po tym bloku,
- przerwaniem wykonania programu, jeśli wyjątek nie zostanie obsłużony.

Proces obsługi wyjątku jest jednokierunkowy. W jego trakcie niszczone są obiekty automatyczne, a sterowanie **nigdy nie wróci** automatycznie do miejsca zgłoszenia wyjątku.

Destruktor **nie powinien zgłaszać wyjątków**. Od C++11 wyjątek, który wydostanie się z destruktora, powoduje **awaryjne przerwanie** programu – niezależnie od tego, czy w programie przewidziano jego obsługę.

Awaryjne przerywanie programu

Funkcja standardowa `exit(int)` kończy program w łagodny sposób: wywołuje destruktory obiektów globalnych i statycznych, opróżnia bufor i zamyka pliki. Nie wywołuje jednak destruktorów **obektów automatycznych** – stos nie jest rozwijany, tak jak przy obsłudze wyjątku.

Funkcja `abort()` kończy program w drastyczny sposób, nie wywołuje żadnych destruktorów i nie opróżnia buforów.

W przypadku wystąpienia nieobsłużonego wyjątku, wywoływana jest biblioteczna funkcja `terminate()`. Domyślnie funkcja ta wywołuje funkcję `abort()` pochodzącą z nagłówka `cstdlib`.

Można zmienić działanie funkcji `terminate()` poprzez zainstalowanie własnej funkcji kończącej, służy do tego funkcja `set_terminate()`.

Własna funkcja kończąca musi być bezargumentową funkcją o rezultacie typu `void`.

Rezultatem wywołania funkcji `set_terminate( )` jest wskaźnik na poprzednio zainstalowaną funkcję.

Można zatem zapamiętać adres oryginalnej funkcji kończącej i przywrócić ją w razie konieczności.

Własna funkcja kończąca

```
#include <exception>
#include <iostream>
#include <cstdlib>

using namespace std;

void myTerminator()
{
    cout << "I'll be back!" << endl;
    exit(1);
}

void(*oldTerminator)() = set_terminate(myTerminator);

void throwFoo()
{
    throw 1;
}

int main()
{
    throwFoo();
    return 0;
}
```

Większa liczba rodzajów wyjątków

Wyjątki mogą być różnych typów. Można zatem rozróżniać rodzaje zgłaszanych sytuacji wyjątkowych.

```
class itemsNormalError
{
};

class itemsCriticalError
{
};
```

```
int getItems()
{
    int numOfItems;
    cin >> numOfItems;

    if(numOfItems <= 0)
        throw itemsNormalError();
    else
    {
        ...
        throw itemsCriticalError();
    }
}
```

Wyłapywanie wszystkich klas wyjątków:

```
class itemsNormalError
{
};

class itemsCriticalError
{
};
```

```
try
{
    getItems();
}

catch(itemsCriticalError)
{
    cout << "Błąd krytyczny";
}

catch(itemsNormalError)
{
    cout << "Błąd zwykły";
}
```

Wychwytywanie różnych klas wyjątków za jednym *zamachem*:

```
try
{
   .getItems();
}

catch(...)
{
    cout << "Biore wszystkie!";
}
```

Wychwytывanie wybranych wyjątków i *hurtowa* obsługa reszty:

```
try
{
   .getItems();
}

catch(itemsCriticalError)
{
    cout << "Błąd krytyczny";
}

catch(...)
{
    cout << "I reszta";
}
```

Dane w wyjątkach

Dotychczas wyjątki były rozróżniane za pomocą ich typów. Inny typ wyjątku – inna sytuacja wyjątkowa.

Wyjątki to obiekty. Mogą posiadać pola, można je inicjować i odczytywać. W polach obiektu stanowiącego wyjątek można przechowywać informację o różnych przyczynach powstania wyjątku.

```
class ItemError
{
    public:
        enum {CRITICAL, NORMAL};

        ItemError(int code): errCode(code) {}
        int getCode() const {return errCode;}

    private:
        int errCode;
};
```


Generowanie wyjątku jednego typu ze zróżnicowaniem kodu pamiętanego w polu klasy:

```
int getItems()  
{  
    int numOfItems;  
    cin >> numOfItems;  
  
    if(numOfItems <= 0)  
        throw itemsError(itemsError::NORMAL);  
    else  
    {  
        ...  
        throw itemsError(itemsError::CRITICAL);  
    }  
}
```

Wyłapanie wyjątku i identyfikacja kodu błędu:

```
try
{
    getItem();
}

catch(itemsError &error)
{
    switch(error.getCode())
    {
        case itemsError::CRITICAL:
            cout << "Wyjatek krytyczny";
            break;

        case itemsError::NORMAL:
            cout << "Wyjatek zwykly";
            break;
    }
}
```

Eleganckie podejście obiektowe z hermetyzacją kodu błędu:

```
class itemsError
{
public:
    enum {CRITICAL, NORMAL};

    itemsError(int code): errCode(code) {}
    int getCode() const {return errCode;}
    const char *errorName() const;

private:
    int errCode;
};

const char *itemsError::errorName() const
{
    switch(errCode)
    {
        case CRITICAL: return "Wyjatek krytyczny";
        case NORMAL   : return "Wyjatek zwykly";
        default       : return "Wyjatek nieznan";
    }
}
```

Wyjątek sam się przedstawia.

```
try
{
    getItem();
}

catch(const itemsError &error)
{
    cout << error.errorName();
}
```

Wyjątki i sprzątanie – RAI

Funkcja wypisująca ponumerowane linie pliku tekstowego w języku C. Każde wyjście z funkcji wymaga **ręcznego zwolnienia** wszystkich zdobytych wcześniej zasobów.

```
int printLines(const char *name)
{
    FILE *file = NULL;
    char *buffer = NULL;
    int n = 0;

    if((file = fopen(name, "r")) == NULL)
        return IN_FILE_ERROR;

    if((buffer = malloc(MAX_LINE_LEN)) == NULL)
    {
        fclose(file);           ← zasób zdobyty wcześniej
        return NO_MEM_ERROR;
    }

    ...
}
```

```
while(fgets(buffer, MAX_LINE_LEN, file) != NULL)
    printf("%d: %s", ++n, buffer);

free(buffer);           ← sprzątanie w kolejności odwrotnej do zdobywania
fclose(file);

return n == 0 ? EMPTY_IN_FILE : OK;
}
```

Z każdym kolejnym zasobem sprzątania przybywa, a pominięcie jednego wywołania oznacza wyciek.

W C++ stosuje się zasadę **RAII** (*Resource Acquisition Is Initialization*): zasób jest zdobywany w **konstruktorze** obiektu i zwalniany w jego **destruktorze**.

```
class TFile
{
public:
    TFile(const char *name, const char *mode)
    {
        file = fopen(name, mode);    ← zdobycie zasobu

        if(file == nullptr)
            throw fileError();        ← obiekt nie powstaje
    }

    ~TFile()
    {
        fclose(file);                ← zwolnienie zasobu
        cout << "~TFile()" << endl;
    }

    ...
}
```

```

TFile(const TFile &) = delete;           ← kopiowanie zabronione, dwa
TFile & operator =(const TFile &) = delete;  ← obiekty zamknęłyby ten sam plik
FILE *get() const {return file;}

private:
    FILE *file;
};

```

Domyślny **konstruktor kopiujący** i domyślny **operator przypisania** skopiowałyby jedynie wskaźnik **file** – dwa obiekty zamknęłyby ten sam plik, a przy przypisaniu plik otwarty wcześniej przez obiekt docelowy nie zostałby zamknięty wcale.

Zapis **= delete** zabrania kompilatorowi **generowania danej funkcji**, a próba jej użycia kończy się błędem kompilacji. Jest uzupełnieniem **= default** – zamiast **wygeneruj standardową wersję** oznacza **nie generuj wcale**.

Ta sama funkcja z obiektami RAII nie zawiera żadnego kodu sprzątającego.

```
void printLines(const char *name)
{
    TFile file(name, "r");
    vector<char> buffer(MAX_LINE_LEN);
    int n = 0;

    while(fgets(buffer.data(), MAX_LINE_LEN,
        file.get()) != nullptr)
        cout << ++n << ": " << buffer.data();

    if(n == 0)
        throw emptyFileError();
} ← ~vector(), ~TFile()
```

```
try
{
    printLines("empty.txt");
}

catch(const fileError &)
{
    cout << "Błąd otwarcia pliku";
}

catch(const emptyFileError &)
{
    cout << "Plik jest pusty";
}
```

```
~TFile()
Plik jest pusty
```

Plik zostaje zamknięty **przed** obsługą wyjątku – w trakcie rozwijania stosu. Tak samo przy normalnym zakończeniu funkcji i przy błędzie przydziłu pamięci w `vector` (wyjątek `bad_alloc`).

Biblioteka standardowa C++ jest zbudowana zgodnie z zasadą RAII. Klasa `ifstream` sama zamyka plik w destruktorze, `string` i `vector` same zwalniają pamięć.

```
void printLines(const string &name)
{
    ifstream file(name); ← nagłówek fstream

    if(!file)
        throw fileError();

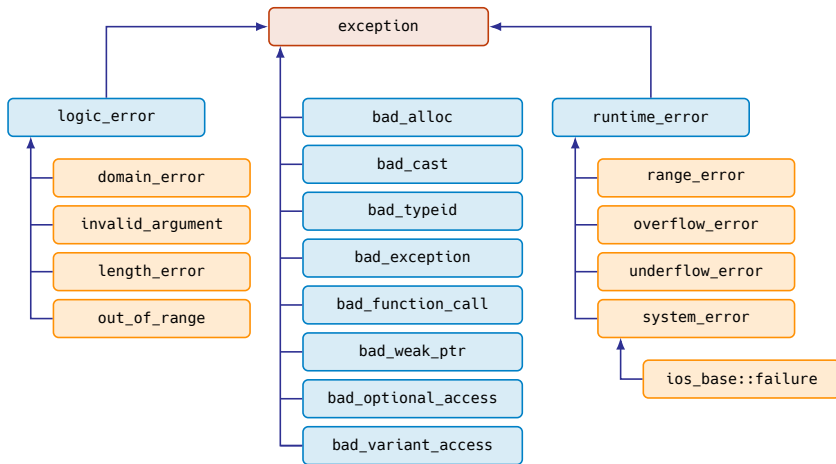
    string line;
    int n = 0;

    while(getline(file, line))
        cout << ++n << ": " << line << endl;

    if(n == 0)
        throw emptyFileError();
} ← ~string(), ~ifstream()
```

Wyjątki są bezpieczne dopiero wtedy, gdy zasoby są *opakowane* w obiekty. Obsługa wyjątków i RAII stanowią całość – jedno bez drugiego prowadzi do wycieków.

Wyjątki biblioteki standardowej



Klasy wyjątków zdefiniowane są w nagłówkach `exception`, `stdexcept`, `new`, `typeinfo` i innych. Wszystkie dziedziczą po klasie `exception`, posiadającej wirtualną metodę `what( )` zwracającą opis błędu.

Własne wyjątki a biblioteka standardowa

Własne klasy wyjątków najwygodniej wyprowadzić z klas biblioteki standardowej. Komunikat przekazuje się konstruktorowi klasy bazowej, a metodę `what( )` dostaje się w spadku.

```
class itemsError : public runtime_error ← nagłówek stdexcept
{
    public:
        enum {CRITICAL, NORMAL};

        itemsError(int code, const char *msg): runtime_error(msg), errCode(code) {}
        int getCode() const {return errCode;}

    private:
        int errCode;
};
```

```
int getItem(const vector<int> &v, int k)
{
    if(k < 0)
        throw itemsError(itemsError::NORMAL, "Indeks nie moze byc ujemny");

    return v.at(k); ← at() zgłasza out_of_range, gdy k >= v.size()
}
```

W odróżnieniu od operatora `[]` metoda `at()` sprawdza poprawność indeksu.

Ponieważ wszystkie wyjątki mają wspólną klasę bazową, jeden blok `catch` może obsłużyć zarówno własne wyjątki, jak i wyjątki biblioteki standardowej.

```
vector<int> v = {10, 20, 30};

try
{
    cout << getItem(v, k) << endl;
}

catch(const itemsError &e) ← najpierw wyjątki szczegółowe
{
    cout << "itemsError: " << e.what() << endl;
}

catch(const exception &e) ← potem ogólne - także out_of_range, bad_alloc...
{
    cout << "exception: " << e.what() << endl;
}
```

Dla $k = 1$:

```
20
```

Dla $k = -1$:

```
itemsError: Indeks nie może być ujemny
```

Dla $k = 5$:

```
exception: vector::_M_range_check: __n (which is 5) >= this->size()  
(which is 3)
```

Bez własnego sprawdzenia `at(-1)` również zgłosiłoby `out_of_range`, ale z mało czytelnym komunikatem – indeks jest typu `size_t`, więc `-1` zamienia się na `18446744073709551615`.

Treść komunikatu wyjątków bibliotecznych zależy od implementacji (przykład komunikatu pochodzi z **g++**).

Kolejność bloków `catch` ma znaczenie – wybierany jest pierwszy pasujący. Gdyby `catch(const exception &)` stał pierwszy, przechwyciłby również `itemsError`.

Łapanie przez referencję zapobiega przy tym **obcięciu** obiektu pochodnego do klasy bazowej.

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty**
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Sterna

Sterna (*heap*) to wydzielony obszar pamięci:

- przeznaczony do przechowywania danych dynamicznych,
- kontrolowany **ręcznie** przez programistę,
- **ograniczony** pod względem rozmiaru.

Rozmiar sterty zmienia się wraz ze zmianą środowiska systemowego i kompilatora. Zwykle jest on wystarczający dla typowych programów. Jednak przetwarzanie grafiki czy danych multimedialnych może wymagać ustawień specyficznych.

Typowy scenariusz wykorzystania sterty:

- przydział niezbędnej w danym momencie ilości pamięci – najpóźniej, jak to możliwe,
- wykorzystanie przydzielonego obszaru,
- zwolnienie przydzielonej pamięci natychmiast, gdy nie jest już używana.

W języku C dynamiczny przydział i zwalnianie pamięci realizowały funkcje:

- `void * malloc(size_t size),`
- `void * calloc(size_t nitems, size_t size),`
- `void * realloc(void *ptr, size_t size),`
- `void free(void *ptr).`

W języku C++ dynamiczny przydział i zwalnianie pamięci realizują:

- operator `new`,
- operator `delete`.

Przydzielanie i zwalnianie pamięci dla obiektu

Użycie operatora **new** powoduje przydzielenie pamięci dla obiektu w obszarze zwanym stertą oraz inicjalizację tego obiektu z wykorzystaniem odpowiedniego konstruktora.

```
TSquare *s;  
s = new TSquare;  
  
cout << s->getSide( );  
  
delete s;
```

Można sterować rodzajem inicjalizacji.

```
TSquare *a = new TSquare;      ← konstruktor domyślny  
TSquare *b = new TSquare(10); ← konstruktor ogólny  
TSquare *c = new TSquare(*b); ← konstruktor kopiujący
```

Usuwanie obiektu. Operator **delete** wywołuje destruktory i zwalnia przydzieloną pamięć.

```
delete a;  
delete b;  
delete c;
```

- Operator `delete` może być stosowany wyłącznie dla obiektów utworzonych operatorem `new`.
- Mieszanie operatorów `new` i `delete` z funkcjami typu `malloc()` i `free()` daje niezdefiniowane rezultaty.
- Jeżeli operator `delete` zostanie użyty dla wskaźnika zerowego, nic się nie stanie, a operacja ta nie jest błędna.
- Dwukrotne (lub wielokrotne) usunięcie tego samego obiektu jest błędem i prowadzi do niezdefiniowanych rezultatów.
- Odwoływanie się do obszaru pamięci, który został wcześniej zwolniony, jest błędem zarówno w C jak i w C++.

Dobłą praktyką jest zerowanie wskaźnika tuż po zwolnieniu pamięci, pozwala to na kontrolowanie, czy można odwołać się do obiektu wskazywanego w innym miejscu programu.

```
TSquare *s = new TSquare;
...
delete s;
s = nullptr; // s = 0 lub rzadziej s = NULL
```

```
if(s) // if(s != 0) lub if(s != NULL)
{
    s->setSide(100);
    ...
}
```

W języku C++ **można zakładać**, że wskaźnik pusty (pokazujący na nic) ma wartość **zero**. Korzystanie ze stałej **NULL** wymaga włączenia przynajmniej pliku nagłówkowego **cstddef**.

Krótkie przypomnienie o tablicach i wskaźnikach

konstrukcja, typ zmiennej, alokacja pamięci, zawartość, zwolnienie

<code>TSquare tab[N]</code>	tablica	automatyczna (stos)	N obiektów <code>TSquare</code>	pamięć jest zwalniana automatycznie po wyjściu z zakresu
<code>TSquare *tab[N]</code>	tablica wskaźników	automatyczna (stos)	N wskaźników na <code>TSquare</code>	<code>delete tab[i]</code> dla każdego stworzonego obiektu
<code>TSquare *tab = new TSquare[n]</code>	wskaźnik	dynamiczna (sterta)	n obiektów <code>TSquare</code>	<code>delete []tab</code>
<code>TSquare **tab = new TSquare *[n]</code>	wskaźnik do wskaźnika	dynamiczna (sterta)	n wskaźników na <code>TSquare</code>	<code>delete tab[i]</code> dla każdego wskaźnika i <code>delete []tab</code>

N oznacza stałą znaną na etapie **kompilacji**, **n** – dowolną wartość ustaloną w trakcie **działania programu**.

Przydzielanie pamięci dla tablic obiektów

Operator `new` może być wykorzystany do utworzenia tablicy obiektów.

Stosowanie tego operatora zapewnia, że dla każdego elementu tablicy zostanie aktywowany konstruktor domyślny.

```
TSquare *squares = new TSquare[10];
```

Tak utworzona tablica może być wykorzystana w normalny dla tablic sposób.

```
for(int i = 0; i < 10; i++)  
    cout << squares[i].getSide();
```

Taką tablicę należy jednak usunąć w specyficzny sposób:

```
delete []squares;
```

Zapewni to uaktywnienie destruktor dla każdego elementu tablicy.

Tablice dynamiczne mogą mieć rozmiar określony zmienną.

```
int numOfSquares;

cout << "Podaj liczbę kwadratów:";
cin >> numOfSquares;

TSquare *squares = new TSquare[numOfSquares];

for(int i = 0; i < numOfSquares; i++)
    cout << squares[i].getSide() << endl;

delete []squares;
```

Operatory `new` i `delete` mogą być stosowane dla typów wbudowanych.

```
int n;  
  
cout << "Podaj liczbę elementów:";  
cin >> n;  
  
int *integers = new int[n]{};  
  
for(int i = 0; i < n; i++)  
    cout << integers[i] << endl;  
  
delete []integers;
```

Należy uważać, aby nie utracić dynamicznie przydzielanych obszarów pamięci.

```
int n;

cout << "Podaj liczbę elementow:";
cin >> n;

int *integers = new int[n]{};

...
integers = new int[n]{}; → wcześniej przydzielony obszar został utracony
...

for(int i = 0; i < n; i++)
    cout << integers[i] << endl;

delete []integers;
```

Można temu częściowo zaradzić stosując wskaźnik, który nie może być później modyfikowany.

```
int *const integers = new int[n]{};
```

Przydzielanie pamięci dla tablic wielowymiarowych

Przy alokacji tablic wielowymiarowych wszystkie wymiary poza pierwszym muszą być dodatnimi wyrażeniami o wartości znanej na etapie kompilacji. Pierwszy wymiar może być zadany zmienną.

```
const int MAX_LINE_LEN = 256;
int numOfLines;

char (*lines)[MAX_LINE_LEN];

cout << "Podaj liczbę linii:";
cin >> numOfLines;

lines = new char[numOfLines][MAX_LINE_LEN];

for(int i = 0; i < numOfLines; i++)
{
    sprintf(lines[i], "Linia nr: %-2d", i + 1);
    cout << endl << lines[i];
}

delete []lines;
```

Klasy zarządzające pamięcią

Klasa przechowująca tablicę liczb całkowitych o rozmiarze zadawanym w trakcie działania programu. Pamięć jest przydzielana w konstruktorze i zwalniana w destruktorze (RAII).

```
class TIntArray
{
public:
    TIntArray(int size): size(size), arr(new int[size]{} ) {}
    ~TIntArray() {delete [] arr;}

    void setItem(int k, int value) {arr[k] = value;}
    int getItem(int k) const {return arr[k];}
    int getSize() const {return size;}

private:
    int size;
    int *arr;
};
```

```

{
    TIntArray a(5);
    TIntArray b = a;    ← konstruktor kopiujący generowany przez kompilator

    b.setItem(0, 10);
    cout << "a[0] = " << a.getItem(0) << endl;
}    ← ~TIntArray() dla b, potem dla a

```

```

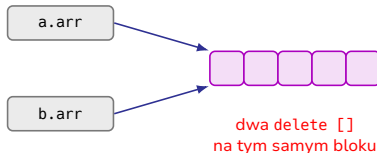
a[0] = 10
free(): double free detected in tcache 2

```

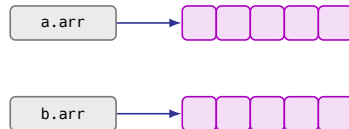
Komunikat pochodzi z biblioteki C (Linux, glibc), która wykryła dwukrotne zwolnienie tego samego obszaru pamięci. Wykrycie nie jest gwarantowane – równie dobrze program mógłby zakończyć się błędem w zupełnie innym miejscu.

Konstruktor kopiujący i operator przypisania generowane przez kompilator kopiują obiekt **pole po polu**. Skopiowany zostaje wskaźnik **arr**, a nie tablica, na którą wskazuje – powstaje **kopia płytka**.

kopia płytka (domyślna)



kopia głęboka (własna)



Obiekty **a** i **b** **współdzielą tablicę**, zmiana w jednym jest widoczna w drugim, a na końcu bloku oba destruktory zwalniają ten sam obszar pamięci. Dwukrotne **delete** ma niezdefiniowane skutki – w g++ program zostaje przerwany, w MSVC (tryb Debug) pojawia się okno z asercją.

Potrzebna jest **kopia głęboka** – nowy obiekt powinien otrzymać własną tablicę z kopią zawartości.

Reguła trzech

Jeżeli klasa wymaga własnego **destruktor**a, to niemal na pewno wymaga też własnego **konstruktora kopiującego** i **operatora przypisania**.

```
TIntArray(const TIntArray &other): size(other.size), arr(new int[other.size])
{
    for(int i = 0; i < size; i++)
        arr[i] = other.arr[i];           ← kopia zawartości, nie wskaźnika
}
```



```

TIntArray & operator =(const TIntArray &other)
{
    if(this != &other)                ← przypisanie do samego siebie
    {
        int *tmp = new int[other.size]; ← najpierw nowa tablica
        for(int i = 0; i < other.size; i++)
            tmp[i] = other.arr[i];

        delete [] arr;                ← dopiero potem zwolnienie starej
        arr = tmp;
        size = other.size;
    }
    return *this;
}

```

Operator przypisania musi dodatkowo zwolnić tablicę, którą obiekt posiadał wcześniej. Przydział nowej pamięci przed zwolnieniem starej sprawia, że w razie wyjątku `bad_alloc` obiekt pozostaje nienaruszony. Szczegóły przeciążania operatorów – w kolejnym rozdziale.

Klasa zarządzająca zasobem ma trzy możliwości:

- napisać kopiowanie samodzielnie (kopia głęboka, jak wyżej),
- zabronić kopiowania (jak w klasie `TFile`):

```
TIntArray(const TIntArray &) = delete;
TIntArray & operator =(const TIntArray &) = delete;
```

- nie zarządzać pamięcią samodzielnie, lecz powierzyć ją obiektowi, który już to potrafi – **reguła zera**:

```
class TIntArray
{
public:
    TIntArray(int size): arr(size) {} ← wektor size zer
    ...                               ← bez destruktor, bez kopiowania,
                                     wszystko generuje kompilator
private:
    vector<int> arr;                  ← kopiowany głęboko, zwalniany sam
};
```

Od C++11 dochodzą jeszcze **operacje przenoszenia** (konstruktor przenoszący i przenoszący operator przypisania) – mówi się wtedy o **regule pięciu**. Klasa `vector` zostanie omówiona przy bibliotece STL.

Za mało pamięci

Jeżeli operator `new` nie potrafi znaleźć ciągłego, wolnego obszaru pamięci dla obiektu:

- sprawdza, czy programista zainstalował funkcję obsługi takiej sytuacji,
- jeżeli tak, wywołuje ją i **ponawia próbę** przydziatu – funkcja obsługi musi więc zwolnić pamięć, zgłosić wyjątek albo zakończyć program, inaczej pętla nigdy się nie skończy.
- jeżeli funkcji obsługi nie ma, zgłaszany jest wyjątek `bad_alloc`,
- dawniej rezultatem operatora była wartość zero – obecnie takie zachowanie daje `new (nothrow)`.

Własną funkcję obsługi braku pamięci można zdefiniować wykorzystując funkcję `set_new_handler()` zdefiniowaną w pliku nagłówkowym `new`. Musi to być bezparametrowa funkcja o rezultacie `void`.

```
void out_of_memory()  
{  
    cerr << "Brak pamieci";  
    exit(1);  
}
```

Sprawdź, kiedy skończy się pamięć

```
#include <iostream>
#include <cstdlib>
#include <new>

using namespace std;

void out_of_memory()
{
    cerr << "Brak pamieci";
    exit(1);
}

int * volatile last; ← volatile, kompilator nie może pominąć zapisu

int main()
{
    set_new_handler(out_of_memory);

    for(;;)
        last = new int[100000];

    return 0;
}
```

Bez zapamiętania wskaźnika kompilator z włączoną optymalizacją (tryb *Release*) usunie całą alokację – jej wynik nie jest nigdzie używany – i program zapętlą się, nie przydzielając żadnej pamięci.

W Linuksie system może przydzielać pamięć *na wyrost* i zamiast wywołania funkcji obsługi braku pamięci program zostanie zakończony przez system.

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone**
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL

Co to takiego przeciążanie?

Przeciążanie funkcji (*function overloading*) to tworzenie większej liczby funkcji o takiej samej nazwie.

Nazwa funkcji może być zatem użyta wielokrotnie do realizacji różnych czynności. Jest więc *przeciążona* dodatkowymi *obowiązkami*.

Kompilator zadba o dobranie właściwej wersji funkcji przeciążonej w zależności od kontekstu jej wywołania.

```
int add(int a, int b)
{
    return a + b;
}

double add(double a, double b)
{
    return a + b;
}
```

```
cout << endl << "Dodawanie int:" << add(1, 1);
cout << endl << "Dodawanie double:" << add(1.0, 1.0);
```

Przeciążanie funkcji jest możliwe, jeżeli **sygnatury** funkcji przeciążonych różnią się od siebie.

Aby sygnatury były inne, funkcje muszą się różnić typami parametrów, liczbą parametrów lub jednocześnie typami i liczbą parametrów.

Funkcje o tych samych nazwach i listach parametrów, różniące się tylko typem rezultatu, nie są przez kompilator rozróżniane i powodują wystąpienie błędu kompilacji.

Dlaczego sygnatura funkcji nie uwzględnia rezultatu funkcji?

```
int addAndPrint(int a, int b)
{
    cout << a + b;
    return a + b;
}

void addAndPrint(int a, int b)    ← błąd - redefinicja funkcji
{
    cout << a + b;
}
```

Gdyby takie przeciążenie było dozwolone, kompilator nie potrafiłby wybrać wersji funkcji.

```
int result = addAndPrint(10, 20);    ← prawdopodobnie int - ale wynik można też
                                     niejawnie konwertować, więc nie jest to pewne
```

```
addAndPrint(10, 20);                ← wynik ignorowany - obie wersje pasują
```

Delikatne problemy i zaskoczenia

Dane są następujące funkcje:

```
void printData(long num)
{
    cout << "Liczba long: " << num << endl;
}
void printData(float num)
{
    cout << "Liczba float: " << num << endl;
}
void printData(double num)
{
    cout << "Liczba double: " << num << endl;
}
```

Poprawne wywołania:

```
long ln = 200;
float fn = 200;
double dn = 200;

printData(ln);
printData(fn);
printData(dn);
```

Wywołanie niepoprawne (błąd kompilatora):

```
| printData(200);
```

Wywołanie poprawne ale zaskakujące:

```
| printData(3.14); ← liczba double
```

W przypadku, gdyby istniały tylko dwie wersje funkcji:

```
void printData(long num)
{
    cout << "Liczba long: " << num << endl;
}

void printData(float num)
{
    cout << "Liczba float: " << num << endl;
}
```

wywołanie `printData(3.14);` spowodowałoby błąd. Konwersje `double` do `long` i `double` do `float` są równorzędne.

Zasady definiowania funkcji

Sygnatury i typ rezultatu są jednakowe, czyli deklaracja tej samej funkcji. Różnice w nazwach parametrów nie są znaczące.

```
void printData(long num);  
  
void printData(long number);
```

Sygnatury są jednakowe, typy rezultatów są różne, kompilator potraktuje drugą deklarację jako niepoprawną redefinicję tej samej funkcji i zgłosi błąd. Przy doborze wersji funkcji przeciążonej typ rezultatu nie jest brany pod uwagę.

```
void printData(long num);  
  
long printData(long num);
```

Nazwa zdefiniowana przez `typedef` nie oznacza nowego typu.

```
typedef int integer;  
  
int printData(int num);  
  
integer printData(integer num);
```

Sygnatury obu funkcji różnią się typami parametrów lub ich liczbą, mamy dwie wersje funkcji przeciążonej.

```
int printData(int num);  
  
long printData(long num);
```

Rozróżnianie wersji funkcji

Poszczególne wersje funkcji przeciążonej różni sygnatura funkcji – nazwa funkcji i lista parametrów.

Określenie, która wersja funkcji przeciążonej ma być wywołana, nazywa się **rozróżnianiem wersji** lub **rozpoznawaniem wywołania** (*function call resolution*).

Najważniejszym elementem rozpoznawania wywołania jest dopasowywanie parametrów (*argument matching*). Polega on na porównaniu parametrów aktualnych wywołania z parametrami formalnymi funkcji.

Proces dopasowania parametrów może skończyć się jednym z wariantów:

- udane dopasowanie,
- dopasowanie nie jest możliwe,
- dopasowanie jest niejednoznaczne.

Rodzaje dopasowania

Przeciążona funkcja:

```
void printData(long num);  
void printData(float num);  
void printData(double num);
```

Udane dopasowanie:

```
printData(10L);  
printData(10.5f);  
printData(10.5);
```

Dopasowanie nie jest możliwe:

```
printData("10");
```

Dopasowanie jest niejednoznaczne:

```
printData(10);
```

Istnieją cztery sposoby uzyskania dopasowania, stosowane w następującej kolejności:

- ścisła zgodność,
- zgodność dzięki **awansowaniu**,
- zgodność dzięki **standardowym przekształceniom** typów,
- zgodność dzięki przekształceniom typów **definiowanym** przez programistę.

Ścisła zgodność

Typy parametru aktualnego i formalnego są takie same.

```
void printData(int);  
void printData(const char *);  
  
printData(0);           ← zgodna z printData(int)  
  
printData("Napis");     ← zgodna z printData(const char *)  
  
printData((const char *)0); ← zgodna z printData(const char *)
```

Literał napisowy jest w C++ typu `const char[]`. Dawniej, ze względu na zgodność z językiem C, dopuszczano jego niejawną konwersję do `char *` – od C++11 jest to błąd, choć część kompilatorów nadal ją przepuszcza z ostrzeżeniem.

Zgodność dzięki awansowaniu

Jeżeli nie występuje ścisła zgodność, to przed następną próbą dopasowania, typ parametru aktualnego jest **rozszerzany** wg następujących zasad:

- parametr typu `char`, `unsigned char` lub `short` awansują do typu `int`; jeżeli rozmiary typu `int` i `short` są równe, parametr `unsigned short` awansuje do `unsigned int`, w przeciwnym wypadku do `int`,
- parametr typu `float` awansuje do typu `double`,
- parametr typu wyliczeniowego awansuje do typu `int`.

```
void printData(int);
void printData(short);
void printData(long);
```

```
printData('a');    ← zgodna z printData(int) - a awansuje do int
printData(3.14);   ← brak możliwości awansu - niejednoznaczne
```

```
void printData(char);
void printData(int);
void printData(unsigned int);
```

```
unsigned char uc;
printData(uc);    ← zgodna z printData(int), uc awansuje do int
```

Zgodność dzięki standardowym przekształceniom typów

Jeżeli nie występuje ścisła zgodność, nawet po zastosowaniu awansowania parametru aktualnego, przed następną próbą dopasowania zastosowane zostaną standardowe **przekształcenia typów** wg następujących zasad:

- każdy argument aktualny **typu liczbowego** będzie zgodny z dowolnym **innym typem** liczbowym parametru formalnego,
- każdy argument aktualny typu **wyliczeniowego** będzie zgodny z dowolnym **typem liczbowym** parametru formalnego,
- literał `0` będzie zgodny z parametrem formalnym **typu wskaźnikowego** jak i **typu liczbowego**,
- wskaźnik do dowolnego typu będzie zgodny z parametrem formalnym typu `void *`.

```
void printData(int);  
void printData(float);
```

```
printData(3.14); ← dwa możliwe przekształcenia - niejednoznaczne
```

```
void printData(int);
```

```
printData(3.14); ← zgodna z printData(int) - przekształcenie standardowe
```

Zgodność dzięki przekształceniom typów definiowanym przez programistę – z operatorem konwersji typu

Jeżeli żaden z poprzednich sposobów nie doprowadził do dopasowania parametrów to dokonuje się zdefiniowanych przez programistę przekształceń typów (jeśli istnieją).

Definiowanie przekształceń typów dotyczy klas, dla nich można definiować **operatory przekształcenia typu**. Takie operatory pozwalają na zdefiniowanie sposobu konwersji obiektu pewnej klasy na obiekt zadanego typu.

```
class myInt
{
    public:
        operator int(); ← operator przekształcenia obiektu myInt do typu int
};

myInt i;

void printData(int);
void printData(float);

printData(i); ← zgodna z printData(int) - wykorzystanie przekształcenia do int
```

Zgodność dzięki przekształceniom typów definiowanym przez programistę – z wykorzystaniem konstruktora

Kompilator w trakcie dopasowywania parametrów będzie używał **konwersji konstruktorowych**.

Polega ona na niejawnym utworzeniu obiektu tymczasowego i zainicjowaniu go odpowiednim konstruktorem.

```
void printName(const string &name);  
  
printName("Ala"); ← zgodna z printName(const string &) - utworzenie  
                   obiektu tymczasowego konstruktorem  
                   string(const char *): printName(string("Ala"))
```

Obiekt tymczasowy można przekazać tylko przez **referencję do stałej** (lub przez wartość). Przy parametrze `string &name` wywołanie `printName("Ala")` jest błędem – zmiany wprowadzone przez funkcję trafiłyby do obiektu, który zaraz przestanie istnieć.

Dzięki możliwości przeciążania funkcji:

- można stosować jednakowe, spójne nazwy funkcji mimo różnic w typach i liczbie parametrów,
- poszczególne wersje funkcji mogą być prostsze i łatwiejsze do napisania, być może niektóre wersje funkcji będą wyraźnie szybsze.

Dobór odpowiedniej wersji funkcji przeciążonej:

- odbywa się na etapie kompilacji i jest pamiętany w kodzie wynikowym,
- nie wpływa na efektywność wykonania programu,
- jest prosty i oczywisty w przypadkach jednoznacznych,
- może powodować błędy kompilacji lub nieprzewidziane zachowanie kodu w przypadku gdy parametry wywołania **nie pasują** do parametrów formalnych funkcji.

Przeciążanie operatorów na przykładzie

Należy opracować program realizujący obliczenia arytmetyczne dla liczb zespolonych. Do realizacji tych obliczeń wykorzystać klasę reprezentującą liczbę zespoloną `TComplex`.

Za liczbę zespoloną z przyjmuje się liczbę w następującej postaci:

$$z = a + i \cdot b,$$

gdzie i jest jednostką urojoną o właściwości: $i^2 = -1$.

Liczbę $a = \Re(z)$ nazywamy częścią rzeczywistą, zaś liczbę $b = \Im(z)$ częścią urojoną liczby zespolonej z .

Podstawowe operacje na liczbach zespolonych:

- dodawanie

$$(a + i \cdot b) + (c + i \cdot d) = (a + c) + i \cdot (b + d),$$

- odejmowanie

$$(a + i \cdot b) - (c + i \cdot d) = (a - c) + i \cdot (b - d),$$

- mnożenie

$$(a + i \cdot b)(c + i \cdot d) = (ac - bd) + i \cdot (bc + ad),$$

- dzielenie

$$\frac{(a + i \cdot b)}{(c + i \cdot d)} = \frac{(ac + bd)}{(c^2 + d^2)} + i \cdot \frac{(bc - ad)}{(c^2 + d^2)}.$$

Przechowywanie wartości w klasie

Klasa `TComplex` posiada konstruktor ogólny, który dzięki parametrom domyślnym staje się domyślnym, kopiujący nie jest konieczny.

```
class TComplex
{
public:
    TComplex(double re = 0, double im = 0);

    double getReal() const;
    double getImag() const;

    void setReal(double re);
    void setImag(double im);

private:
    double real;
    double imag;
};
```

```
TComplex::TComplex(double re, double im): real(re), imag(im)
{
}

double TComplex::getReal() const
{
    return real;
}

double TComplex::getImag() const
{
    return imag;
}

void TComplex::setReal(double re)
{
    real = re;
}

void TComplex::setImag(double im)
{
    imag = im;
}
```

Użycie klasy

Kreowanie obiektów:

```
TComplex a;           ← konstruktor TComplex() i domyślne parametry  
TComplex b(2);        ← konstruktor TComplex(2) i parametr domyślny  
TComplex c(1, 1);     ← konstruktor TComplex(1, 1)
```

Dodawanie liczb zespolonych – wersja prymitywna:

```
a.setReal(b.getReal() + c.getReal());  
a.setImag(b.getImag() + c.getImag());
```

Czy **można tak?**

```
a = b + c;  
a = b * c;
```

W zasadzie tak, ale aktualnie nie. Kompilator nie wie jak dodawać, mnożyć, dzielić czy odejmować liczby zespolone (manipulowanie obiektami). Należy go tego nauczyć – czyli związać z klasą `TComplex` zestaw operatorów, realizujących określone działania.

Operatory są w naturalny sposób przeciążone. Operatory potrafią wykonywać określone działania dla danych różnych typów.

```
int ia, ib, ic;  
ic = ia + ib;  
  
float fa, fb, fc;  
fc = fa + fb;
```

Koncepcja przeciążania operatorów w języku C++ nie jest nowa. Nowością jest potraktowanie użycia operatora jako wywołania specjalnej funkcji, zwanej funkcją operatorową.

Programista może napisać dla większości operatorów specjalne funkcje, określające sposób działania danego operatora dla argumentów, z których przynajmniej jeden jest obiektem.

Zasady wykorzystania funkcji operatorowych:

- nie wolno zmieniać znaczenia operatora określonego dla wbudowanych typów danych,
- nie wolno budować nowych operatorów,
- przynajmniej jeden argument funkcji operatorowej musi być obiektem jakiejś klasy,
- nie wolno zmieniać zdefiniowanych pierwotnie reguł pierwszeństwa operatorów,
- musi być zachowana liczba argumentów operatora.

Np. przeciążenie operatora przypisania dla klasy `TComplex`:

```
TComplex a;  
TComplex b(1, 2);  
  
a = b;
```

Spełnia powyższe zasady.

Definicja i implementacja funkcji operatorowej dla operatora =

```
class TComplex
{
public:
    void operator = (TComplex &z);
    ...
};
```

```
void TComplex::operator = (TComplex &z)
{
    real = z.real; // setReal(z.getReal());
    imag = z.imag; // setImag(z.getImag());
}
```

Jak używać, jak to działa?

```
TComplex a;  
TComplex b(1, 2);  
  
a = b;
```

Przypisanie `a = b` spowoduje wywołanie `a.operator = (b)`.

Na rzecz obiektu `a` zostanie wywołana funkcja `TComplex::operator = ( )` z argumentem `b`.

Czy taki operator zawsze działa jak należy?

```
TComplex a;  
TComplex b;  
TComplex c(1, 2);  
  
a = b = c;
```

Można się spodziewać, że przypisanie `a = b = c` powinno zadziałać w taki sposób `a.operator = (b.operator = (c))`.

Czy implementacja operatora pozwala na takie działanie (co zwraca operator)?

Po poprawkach, operator będzie działał poprawnie w przypadku wielokrotnego przypisania.

```
TComplex & TComplex::operator = (TComplex &z)
{
    real = z.real;
    imag = z.imag;

    return *this;
}
```

```
TComplex a;
TComplex b;
TComplex c(1, 2);

a = b = c;
```

Przypisanie `a = b = c` spowoduje wywołanie `a.operator = (b.operator = (c))`.

Operator przypisania warto zabezpieczyć przed przypisaniem do samego siebie.

```
TComplex a;
```

```
a = a;
```

```
TComplex & TComplex::operator = (TComplex &z)
```

```
{
```

```
    if(&z != this)
```

```
    {
```

```
        real = z.real;
```

```
        imag = z.imag;
```

```
    }
```

```
    return *this;
```

```
}
```

W klasie `TComplex` przypisanie do samego siebie jest nieszkodliwe, warunek pokazuje jedynie mechanizm.

Ma on znaczenie w klasach zarządzających zasobami. Operator przypisania w takiej klasie zwykle najpierw zwalnia pamięć posiadaną przez obiekt, a następnie kopiuje dane z obiektu źródłowego (np. wcześniej omawiane `TIntArray`).

Przy przypisaniu `a = a` obiekt źródłowy i docelowy są tym samym obiektem, więc kopiowanie odbyłoby się ze zwolnionej już pamięci.

```
delete [] arr;                                ← zwolnienie własnej tablicy...

arr = new int[other.size];

for(int i = 0; i < other.size; i++)
    arr[i] = other.arr[i];                    ← ...a other.arr to ta sama, zwolniona tablica
```

Dodanie modyfikatora `const` do parametru funkcji operatorowej pozwoli na współpracę z obiektami stałymi.

```
TComplex & TComplex::operator = (const TComplex &z)
{
    if(&z != this)
    {
        real = z.real;
        imag = z.imag;
    }

    return *this;
}
```

```
const TComplex a(0, 0);
TComplex b;

b = a;
```

Należy pamiętać, że operator przypisania to co innego niż konstruktor kopiujący. Czasem łatwo się pomylić.

```
TComplex a(1, 2);  
TComplex b = a; ← konstruktor kopiujący  
  
b = a;          ← operator przypisania
```

Czy trzeba przeciążać operator przypisania i definiować konstruktor kopiujący?

Jeżeli operator przypisania nie jest jawnie zdefiniowany dla danej klasy, a jest potrzebny, kompilator generuje domyślny operator przypisania, realizujący kopiowanie **pole po polu**.

Jeżeli konstruktor kopiujący nie jest jawnie zdefiniowany, kompilator generuje domyślny, również kopiujący **pole po polu**.

Dla klasy `TComplex` wersje domyślne robią dokładnie to samo co napisane wyżej – własny operator przypisania posłużył tu jedynie do pokazania budowy funkcji operatorowej.

Zgodnie z **regułą trzech** własne kopiowanie pisze się wtedy, gdy klasa sama zarządza zasobem (np. pamięcią dynamiczną, jak `TIntArray`). W pozostałych przypadkach lepiej zdać się na kompilator (**reguła zera**) – ręcznie napisane kopiowanie nic nie daje, a łatwo w nim zapomnieć o nowym polu.

Klasa TComplex po poprawkach

```
class TComplex
{
public:
    TComplex(double re = 0, double im = 0);
    TComplex(const TComplex &z);

    double getReal() const;
    double getImag() const;

    void setReal(double re);
    void setImag(double im);

    TComplex & operator = (const TComplex &z);

private:
    double real;
    double imag;
};
```

```
TComplex::TComplex(double re, double im): real(re), imag(im) {}  
TComplex::TComplex(const TComplex &z): real(z.real), imag(z.imag) {}  
  
double TComplex::getReal() const {return real;}  
double TComplex::getImag() const {return imag;}  
  
void TComplex::setReal(double re) {real = re;}  
void TComplex::setImag(double im) {imag = im;}  
  
TComplex & TComplex::operator = (const TComplex &z)  
{  
    if(&z != this)  
    {  
        real = z.real;  
        imag = z.imag;  
    }  
    return *this;  
}
```

Konstruktor kopiujący i operator przypisania pozostawiono jako przykłady – dla tej klasy nie są potrzebne, wersje generowane przez kompilator działają identycznie.

Dodawanie liczb zespolonych

```
TComplex a;  
TComplex b(2);  
TComplex c(1, 1);  
  
a = b + c;
```

Wyrażenie `a = b + c` spowoduje wywołanie

`a.operator = (b.operator + (c)).`

W C i C++ rezultatem funkcji nie powinien być **wskaźnik ani referencja do zmiennych automatycznych** funkcji. Te lokowane są zwykle na stosie i po zakończeniu działania funkcji **przestają istnieć**.

Możliwa implementacja operatora dodawania:

```
class TComplex
{
    public:
        TComplex operator + (const TComplex &z) const;
        ...
};
```

```
TComplex TComplex::operator + (const TComplex &z) const
{
    TComplex temp;
    temp.real = real + z.real;
    temp.imag = imag + z.imag;

    return temp;
}
```

Przykłady użycia operatorów:

```
TComplex a;  
TComplex b(2);  
  
a + b;
```

Wyrażenie spowoduje wywołanie `a.operator + (b)`.

```
TComplex a;  
TComplex b(2);  
TComplex c(1, 1);  
  
a = b + c;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator + (c))`.

Wersja elegancka – bez dodatkowych zmiennych pomocniczych z wykorzystaniem konstruktora:

```
class TComplex
{
    public:
        TComplex operator + (const TComplex &z) const;
        ...
};
```

```
TComplex TComplex::operator + (const TComplex &z) const
{
    return TComplex(real + z.real, imag + z.imag);
}
```

Odejmowanie liczb zespolonych

W sposób analogiczny jak dla dodawania:

```
TComplex a;  
TComplex b(2);  
TComplex c(1, 1);  
  
a = b - c;
```

```
class TComplex  
{  
    public:  
        TComplex operator - (const TComplex &z) const;  
        ...  
};
```

```
TComplex TComplex::operator - (const TComplex &z) const  
{  
    return TComplex(real - z.real, imag - z.imag);  
}
```

Zmiana znaku liczb zespolonych

Należy rozszerzyć klasę o kolejną funkcję operatorową (przeciążaną) dla operatora w wersji unarnej. Operator **nie posiada drugiego argumentu**.

```
TComplex a;
TComplex b(2, 2);

a = -b;
```

```
class TComplex
{
public:
    TComplex operator - (const TComplex &z) const;
    TComplex operator - () const; ← unarny minus
    ...
};
```

```
TComplex TComplex::operator - () const
{
    return TComplex(-real, -imag);
}
```

Przykłady użycia operatorów:

```
TComplex a;  
TComplex b(2);  
  
a = -b;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator - ())`.

```
TComplex a;  
TComplex b(2);  
TComplex c(1, 1);  
  
a = b - c;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator - (c))`.

Dla kompletności implementacji należy zdefiniować unarny plus:

```
class TComplex
{
    public:
        TComplex operator + (const TComplex &z) const;
        TComplex operator + () const; ← unarny plus
        ...
};
```

```
TComplex TComplex::operator + () const
{
    return TComplex(real, imag);
}
```

Przykłady użycia:

```
TComplex a;
TComplex b(2);

a = +b;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator + ())`.

Operatory a różne typy

W przypadku różnych typów argumentów operator może nie działać.

```
TComplex z;  
double d;  
  
z + d;  
d + z;
```

Wyrażenia spowodują wywołania:

```
z.operator + (TComplex(d))  
d.operator + (z) ???
```

Funkcja operatorowa **nie musi** być (z pewnymi wyjątkami, np. przypisanie) zdefiniowana jako **funkcja składowa** a jako zwykła funkcja, posiadająca przynajmniej jeden argument będący obiektem.

Można to wykorzystać do definiowania funkcji operatorowych współpracujących z różnymi typami parametrów. Na przykład liczba zespolona (obiekt) i rzeczywista (typ wbudowany).

Dla jednoargumentowego operatora `#` i obiektu `0`:

```
| #0;
```

wyrażenie spowoduje wywołanie:

- funkcji składowej `0.operator # ( )`,
- funkcji zwykłej (nieskładowej) `operator #(0)`.

Dla dwuargumentowego operatora `#` i obiektów `A` i `B`:

```
| A # B;
```

wyrażenie spowoduje wywołanie:

- funkcji składowej `A.operator # (B)`,
- funkcji zwykłej (nieskładowej) `operator # (A, B)`.

Implementacja operatora dodawania w postaci zwykłej funkcji:

```
TComplex operator + (double n, const TComplex &z)
{
    return TComplex(n + z.getReal(), z.getImag());
}
```

```
TComplex a;
TComplex b(2);
double d;

d = a + b;
a = d + b;
```

Wyrażenia spowodują wywołania:

`operator + (d, b)`

`a.operator = (operator + (d, b))`

Nieskładowa funkcja operatorowa ma utrudniony dostęp do pól klasy.

```
TComplex operator + (double n, const TComplex &z)
{
    return TComplex(n + z.getReal(), z.getImag());
}
```

Aby funkcja nieskładowa miała wygodniejszy dostęp do pól klasy, może się z nią **zaprzyjaźnić** – modyfikator `friend`. Taka funkcja będzie miała bezpośredni dostęp do prywatnych pól klasy.

```
class TComplex
{
    friend TComplex operator + (double n, const TComplex &z);
    ...
};

TComplex operator + (double n, const TComplex &z)
{
    return TComplex(n + z.real, z.imag);
}
```

Kilka przykładów na różne okazje:

```
class TComplex
{
    friend TComplex operator + (double n, const TComplex &z);
    friend TComplex operator + (const TComplex &a, double n);
    friend TComplex operator + (const TComplex &a, const TComplex &b);
    ...
};
```

```
TComplex operator + (double n, const TComplex &z)
{
    return TComplex(n + z.real, z.imag);
}
```

```
TComplex operator + (const TComplex &z, double n)
{
    return TComplex(n + z.real, z.imag);
}
```

```
TComplex operator + (const TComplex &a, const TComplex &b)
{
    return TComplex(a.real + b.real, a.imag + b.imag );
}
```

Wykorzystanie funkcji:

```
TComplex a, b, c;  
TComplex d(2);  
TComplex e(2, 2);  
double n = 3;  
  
a = n + c;  
b = d + 2;  
c = d + e;
```

Wyrażenia spowodują wywołania:

```
a.operator = (operator + (n, c))  
b.operator = (operator + (d, double(2)))  
c.operator = (operator + (d, e))
```

W przypadku *braku* odpowiednich funkcji operatorowych dla typów, np. przy deklaracji i użyciu:

```
class TComplex
{
    friend TComplex operator + (const TComplex &a, const TComplex &b);
};
```

```
TComplex a, b, c;
TComplex d(2);
TComplex e(2, 2);
```

```
a = 6 + c;
b = d + 9;
c = 3 + 5;
```

Wyrażenia spowodują wywołania:

```
a.operator = (operator + (TComplex(double(6)), c))
b.operator = (operator + (d, TComplex(double(9))))
c.operator = (TComplex(double(3 + 5)))
```

Funkcje operatorowe są używane, gdy przynajmniej jeden argument jest obiektem. W wyrażeniu `3 + 5` oba argumenty są typu `int`, więc wykonywane jest dodawanie liczb, a dopiero wynik `8` jest zamieniany na obiekt `TComplex` przez konstruktor.

Kompilator będzie próbował konwertować typy argumentów, żeby dopasować odpowiednią funkcję.

Takie postępowanie może zakończyć się niepowodzeniem.

Wtedy można zdefiniować **operator konwersji** (np. `operator double()`), jak w klasie `myInt` z `operator int()`, aby mieć pewność, że typy są zmieniane w odpowiedni i kontrolowany sposób.

Wybór sposobu implementacji (funkcje operatorowe dla różnych typów albo operatory konwersji) zależy od programisty. Nie należy jednak stosować obu jednocześnie – klasa `TComplex` z konstruktorem `TComplex(double)` i operatorem `operator double()` sprawi, że wyrażenie `z + 2.0` stanie się niejednoznaczne – kompilator nie będzie wiedział, czy zamienić liczbę na obiekt, czy obiekt na liczbę.

Operatory przedrostkowe i przyrostkowe

Można przeciążać operatory **przedrostkowe** i **przyrostkowe**. Wersje przedrostkowe definiuje się w poznany sposób.

Wersje przyrostkowe definiuje się przez funkcję operatorową z parametrem `int`. Parametr nie ma żadnego znaczenia praktycznego, jest istotny tylko ze względu na syntaktykę języka.

```
class TComplex
{
    public:
        TComplex & operator ++ ();    ← postać przedrostkowa
        TComplex operator ++ (int);   ← postać przyrostkowa

        TComplex & operator -- ();    ← postać przedrostkowa
        TComplex operator -- (int);   ← postać przyrostkowa

        ...
};
```

Przykład implementacji operatorów przedrostkowych:

```
TComplex & TComplex::operator ++ ()  
{  
    ++real;  
    ++imag;  
  
    return *this; ← zwraca zmodyfikowany obiekt  
}
```

```
TComplex & TComplex::operator -- ()  
{  
    --real;  
    --imag;  
  
    return *this; ← zwraca zmodyfikowany obiekt  
}
```

Wywołanie operatorów przedrostkowych:

```
TComplex a;  
TComplex b(2);  
  
a = ++b;  
a = --b;
```

Wyrażenia spowodują wywołania:

a.operator = (b.operator ++ ())

a.operator = (b.operator -- ())

Przykład implementacji operatorów przyrostkowych:

```
TComplex TComplex::operator ++ (int)
{
    TComplex copy;

    copy = *this; ← kopia obiektu, przypisanie
    ++real;
    ++imag;

    return copy; ← zwraca kopię
}

TComplex TComplex::operator -- (int)
{
    TComplex copy(*this); ← kopia obiektu, konstruktor kopiujący

    --real;
    --imag;

    return copy; ← zwraca kopię
}
```

Wywołanie operatorów przyrostkowych:

```
TComplex a;  
TComplex b(2);  
  
a = b++;  
a = b--;
```

Wyrażenia spowodują wywołania:

a.operator = (b.operator ++ (0))

a.operator = (b.operator -- (0))

Przeciążenie operatorów << i >>

Często posługujemy się operatorem <<.

```
cout << "Ala ma psa!";
```

```
int n = 2;  
cout << n;
```

Operator << jest przeciążony i taki zapis spowoduje wywołanie:

`cout.operator <<(n).`

`cout` jest obiektem klasy `ostream`, dla tej klasy dokonano przeciążania operatora.

Przeciążanie jest możliwe, gdy jednym z argumentów jest obiekt typu zdefiniowanego przez użytkownika. Takim typem zdefiniowanym (choć bibliotecznym) jest klasa `ostream`.

Możemy po raz kolejny przetłumaczyć operator `<<`. Jest on dwuoperandowy, więc argumenty będą typu `ostream` i `TComplex`.

Nie mamy dostępu do klasy `ostream`, zatem możemy użyć tylko zwykłej funkcji operatorowej.

```
ostream & operator <<(ostream &s, const TComplex &z)
{
    s << "(" << z.getReal() << ", " << z.getImag() << "i)";

    return s;
}
```

W podobny sposób można przetładować operator `>>` dla klasy `istream`.

```
istream & operator >>(istream &s, TComplex &z)
{
    double re, im;

    if(s >> re >> im) ← oba odczyty udane?
    {
        z.setReal(re);
        z.setImag(im);
    } ← jeśli nie - z pozostaje bez zmian

    return s; ← strumień ze stanem błędu wraca do wywołującego
}
```


Takie funkcje wymagają zastosowania metod akcesorów i modyfikatorów klasy. Można to zmienić definiując funkcje operatorowe jako zaprzyjaźnione z klasą.

Dzięki temu zwykła funkcja ma prawo dostępu do prywatnych składników klasy. To tak, jakby składniki te stały się dla niej publiczne.

```
class TComplex
{
    public:
        friend ostream & operator <<(ostream &s, const TComplex &z);
        friend istream & operator >>(istream &s, TComplex &z);
        ...
};
```

Deklaracja przyjaźni tylko deklaruje przyjaźń, nazwa funkcji zaprzyjaźnionej nie staje się przez to nazwą z zakresu tej klasy.

Funkcja zaprzyjaźniona z klasą nie zawiera wskaźnika `this` do klasy, która uznaje ją za przyjaciela.

```
ostream & operator <<(ostream &s, const TComplex &z)
{
    s << "(" << z.real << ", " << z.imag << "i)";

    return s;
}
```

```
istream & operator >>(istream &s, TComplex &z)
{
    double re, im;

    if(s >> re >> im)
    {
        z.real = re;
        z.imag = im;
    }

    return s;
}
```

operatory, które można przeciążyć

+	-	*	/	%	~	&	
^	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=	=	*=
<<=	>>=	[]	()	->	->*	new	delete
		<=>	new[]	delete[]	co_await		

operatory i instrukcje, których przeciążyć nie wolno

::	.*	.	?:
sizeof	typeid	alignof	noexcept

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw**
- 13 Szablony
- 14 Podstawy STL

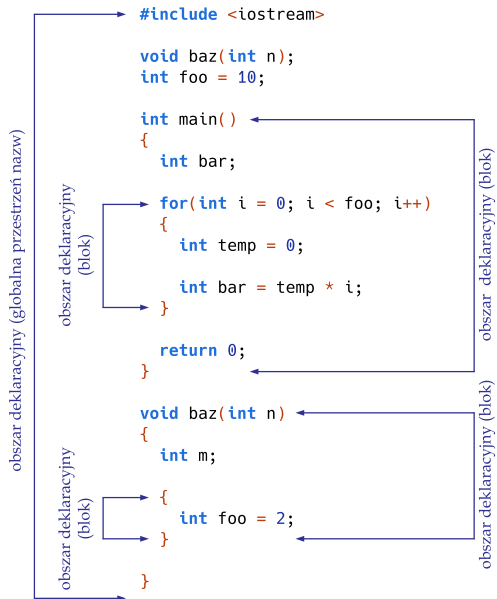
Tradycyjne przestrzenie nazw

Nazwy (identyfikatory) mogą odnosić się do zmiennych, funkcji, struktur, wyliczeń, klas i składowych struktur i klas.

Kiedy projekt programistyczny rozrasta się, rośnie też ryzyko kolizji tych nazw. Ryzyko to zwiększa się jeszcze bardziej, jeśli w projekcie wykorzystywane są biblioteki pochodzące z różnych źródeł.

Obszar deklaracyjny (*declarative region*) to taki obszar, w którym można umieszczać deklaracje: np. zmienne globalne można deklarować poza wszelkimi funkcjami. Obszarem deklaracyjnym dla tego rodzaju zmiennych jest obszar pliku, w których są deklarowane.

Jeśli zmienna zostanie zadeklarowana w funkcji, jej obszar deklaracyjny będzie ograniczony do bloku, w którym została zadeklarowana.

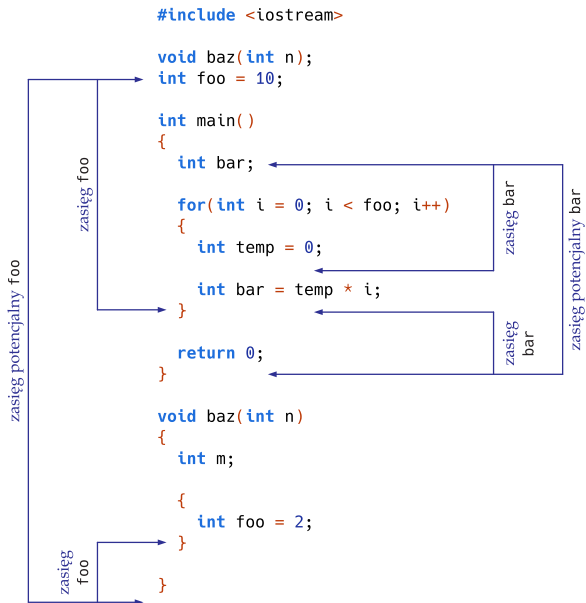


Zasięg potencjalny (*potential scope*) zmiennej rozciąga się od miejsca deklaracji do końca obszaru deklaracyjnego. Zasięg potencjalny jest więc węższy od obszaru deklaracyjnego, bo nie można korzystać ze zmiennej przed miejscem jej deklaracji.

Zmienna nie musi jednak być widoczna w całym jej zasięgu potencjalnym. Może na przykład zostać przestonięta przez inną zmienną o takiej samej nazwie zadeklarowanej w zagnieżdżonym obszarze deklaracyjnym.

Na przykład zmienna lokalna funkcji (dla niej obszarem deklaracyjnym jest funkcja) zakrywa zmienną globalną zadeklarowaną w tym samym pliku (jej obszar deklaracyjny to plik).

Ta część programu, dla której zmienna jest widoczna, to **zasięg zmiennej** i w takim znaczeniu stosowaliśmy ten termin dotychczas.



Nowe mechanizmy przestrzeni nazw

C++ dodał możliwość tworzenia **nazwanych przestrzeni nazw**, wyznaczających nowego rodzaju obszary **deklaracyjne**, których głównym zadaniem jest udostępnienie miejsca do deklaracji nazw.

Nazwy z jednej przestrzeni nazw nie kolidują z identycznymi nazwami deklarowanymi w innych przestrzeniach nazw.

Dodatkowo istnieją mechanizmy pozwalające poszczególnym częściom programu na korzystanie z nazw deklarowanych w przestrzeni nazw.

```

namespace jack
{
    void fetch();      ← prototyp funkcji
    int bucket;        ← definicja zmiennej
    double capacity;   ← definicja zmiennej
    struct well        ← definicja struktury
    {
        ...
    };
}

```

```

namespace jill
{
    double fetch;      ← definicja zmiennej
    int bucket();      ← prototyp funkcji
    double capacity;   ← definicja zmiennej
    struct hill        ← definicja struktury
    {
        ...
    };
}

```

Przestrzenie nazw mogą być lokowane na poziomie globalnym albo wewnątrz innych przestrzeni nazw, **nie mogą** jednak być **zagnieżdżane w blokach**.

Stąd nazwa zadeklarowana w przestrzeni nazw cechuje się domyślnie łączeniem zewnętrznym (chyba że odnosi się do stałej).

Poza przestrzeniami nazw definiowanymi przez programistę wyróżniona została **globalna** przestrzeń nazw. Odpowiada ona obszarowi deklaracyjnemu pliku.

To, co dotychczas było określane mianem **zmiennych globalnych**, możemy nazywać elementami **globalnej przestrzeni nazw**.

Nazwy w dowolnych przestrzeniach nazw nie kolidują ze sobą. Stąd nazwa `fetch` z przestrzeni `jack` może współistnieć z taką nazwą z przestrzeni `jill`.

Podobnie `hill` umieszczona w przestrzeni nazw `jill` może współistnieć z inną nazwą `hill` zewnętrzną wobec tej przestrzeni nazw.

Reguły obowiązujące deklaracje i definicje w przestrzeniach nazw są identyczne jak stosowne reguły definicji i deklaracji globalnych.

Przestrzenie nazw są **otwarte**, co oznacza, że można je uzupełniać innymi nazwami.

```
namespace jill
{
    string goose(const string);
}
```

Podobnie z definicjami uzupełniającymi prototypy:

```
namespace jack
{
    void fetch()
    {
        ...
    }
}
```

Odwołanie do przestrzeni nazw

Najprostszym mechanizmem umożliwiającym odwołanie do nazw wewnątrz przestrzeni nazw jest operator zakresu `:`.

```
jack::bucket = 5;  
jill::hill mountain;  
jack::fetch();
```

Identyfikator pozbawiony operatora zakresu (np. `bucket`) to **nazwa niekwalifikowana**.

Identyfikatory poprzedzone operatorem zakresu (np. `jack::bucket`) to nazwy **kwalifikowane**.

Deklaracja i dyrektywa `using`

Konieczność kwalifikowania wszystkich nazw jest kłopotliwa.

Język C++ przewiduje dwa dodatkowe mechanizmy upraszczające stosowanie przestrzeni nazw: **deklaracje** `using` i **dyrektywy** `using`.

Deklaracje `using` pozwalają na lokalne udostępnianie wybranych identyfikatorów przestrzeni nazw.

Dyrektywy `using` udostępniają zaś lokalnie całą zawartość przestrzeni nazw.

Deklaracja zawiera kwalifikowaną nazwę poprzedzoną słowem `using`.

```
| using jill::fetch;
```

Deklaracja dodaje wskazaną nazwę do obszaru deklaracyjnego, w którym wystąpiła.

Po wystąpieniu takiej deklaracji można zamiast `jill::fetch` stosować nazwę `fetch`.

Deklaracja `using` wprowadza nazwę do obszaru deklaracyjnego, w którym wystąpiła. Użycie `using jill::fetch` w funkcji `main()` wprowadza nazwę `fetch` do obszaru deklaracyjnego wyznaczonego funkcją.

```
namespace jill
{
    double fetch;
    int bucket();
    double capacity;
    struct hill { ... };
}
```

```
char fetch;           ← deklaracja globalnej zmiennej fetch

int main()
{
    using jill::fetch; ← wprowadzenie fetch do lokalnego obszaru deklaracyjnego
    double fetch;      ← błąd, nazwa fetch jest już zadeklarowana

    cin >> fetch;       ← odczyt wartości do jill::fetch
    cin >> ::fetch;     ← odczyt wartości do globalnej zmiennej fetch
}
```

Umieszczenie deklaracji `using` na zewnątrz funkcji wprowadza nazwę do globalnej przestrzeni nazw pliku.

```
namespace jill
{
    double fetch;
    int bucket();
    double capacity;
    struct hill { ... };
}
```

```
void foo();

using jill::fetch; ← wprowadzenie fetch do globalnej przestrzeni nazw pliku

int main()
{
    cin >> fetch; ← odczyt wartości do jill::fetch
    foo();
}

void foo()
{
    cout << fetch; ← wyświetlenie jill::fetch
}
```

Dyrektywa `using` zawiera określenie przestrzeni nazw poprzedzone słowami `using namespace` i udostępnia wszystkie nazwy ze wskazanej przestrzeni nazw, które od tej pory można stosować z pominięciem operatora zakresu.

```
using namespace jack;
```

Umieszczenie dyrektywy `using` na poziomie globalnym (poziomie pliku) czyni zawartość przestrzeni nazw dostępną globalnie (w pliku).

```
#include <iostream>

using namespace std;

int main()
{
    ...
}
```

Osadzenie dyrektywy `using` we wnętrzu funkcji udostępnia nazwy przestrzeni nazw w obrębie tej jednej funkcji.

```
#include <iostream>

int main()
{
    using namespace jack;
    ...
}
```

Stosowanie dyrektywy `using` celem wciągania kompletu nazw z przestrzeni nazw nie jest równoznaczne wielokrotnemu zastosowaniu deklaracji `using` dla poszczególnych nazw.

Dyrektywa bardziej przypomina **masowe zastosowanie** operatora zakresu.

Po deklaracji `using` nazwa jest traktowana jak zadeklarowana w miejscu wystąpienia deklaracji `using`. Jeśli w danej funkcji zadeklarowana jest już nazwa, nie można takiej nazwy wciągnąć do funkcji deklaracją `using`.

Przy zastosowaniu dyrektywy `using` rozstrzyganie nazw odbywa się tak, jakby nazwy te zostały zadeklarowane w najmniejszym możliwym obszarze deklaracyjnym zawierającym zarówno przestrzeń nazw, jak i dyrektywę `using`.

Jeśli dyrektywa `using` zaimportuje nazwę, która jest już zdefiniowana w funkcji, nazwa lokalna będzie zakrywać nazwę importowaną z przestrzeni nazw, jak i ewentualną nazwę deklarowaną globalnie.

Wciąż można jednak odwołać się lokalnie do nazwy z przestrzeni nazw, stosując operator zakresu.

```
namespace jill
{
    double fetch;
    int bucket();
    double capacity;
    struct hill { ... };
}
```

```
char fetch;           ← deklaracja globalnej zmiennej fetch
int main()
{
    using namespace jill; ← import wszystkich nazw przestrzeni jill
    hill mountain;       ← utworzenie struktury typu jill::hill
    int water = bucket(); ← wywołanie funkcji jill::bucket
    double fetch;        ← poprawnie, zakrycie jill::fetch

    cin >> fetch;        ← odczyt wartości do lokalnej nazwy fetch
    cin >> ::fetch;       ← odczyt wartości do globalnej nazwy fetch
    cin >> jill::fetch;   ← odczyt wartości do nazwy jill::fetch
}

void foo()
{
    hill top;            ← błąd
    jill::hill crest;    ← poprawnie
}
```

W funkcji `main( )` nazwa `jill::fetch` jest wciągana do lokalnej przestrzeni nazw. Nie posiada jednak lokalnego zasięgu, więc nie zakrywa globalnej wersji `fetch`.

Deklarowana lokalnie `fetch` zakrywa zarówno `jill::fetch`, jak i globalną wersję `fetch`.

Obie te zmienne są jednak dostępne przy użyciu operatora zakresu.

Choć dyrektywa `using` w obrębie funkcji powoduje traktowanie importowanych nazw jako zadeklarowanych poza tą funkcją, wcale nie udostępnia tych nazw pozostałym funkcjom pliku.

Dlatego w funkcji `foo( )` nie można wykonać niekwalifikowanego odwołania do nazwy `hill`.

Deklaracje `using` są **bezpieczniejsze** od **dyrektyw** `using`, ponieważ jawnie wymieniają udostępniane lokalnie nazwy.

Jeśli wciągana deklaracją nazwa koliduje z nazwą lokalną, kompilator da o tym znać.

Dyrektywa udostępnia lokalnie wszystkie nazwy przestrzeni nazw, nawet te zupełnie nam niepotrzebne.

Jeśli dojdzie do kolizji nazw, wersja lokalna zakryje wersję z przestrzeni nazw i nie będzie ze strony kompilatora żadnego ostrzeżenia.

Zagnieżdżanie przestrzeni nazw

Deklaracje przestrzeni nazw można zagnieżdżać.

```
namespace elements
{
    namespace hot
    {
        int fire;
    }

    float water;
}
```

Do zmiennej `fire` można się odwołać z użyciem operatora zakresu.

```
elements::hot::fire = 998;
```

Można też użyć dyrektywy `using` i udostępnić nazwy wewnętrznej przestrzeni nazw.

```
using namespace elements::hot;

fire = 998;
```

W obrębie przestrzeni nazw można też stosować deklaracje i dyrektywy `using`.

```
namespace pieces
{
    using jill::fetch;
    using namespace elements;

    using std::cout;
    using std::cin;
}
```

Przykładowe odwołanie do `jill::fetch`, która jest składową przestrzeni nazw `pieces`.

```
| std::cin >> pieces::fetch;
```

Ponieważ ta sama nazwa jest też elementem przestrzeni nazw `jill`, można równie dobrze stosować `jill::fetch`.

```
| std::cout << jill::fetch;
```

Można też (o ile nie ma kolizji) zastosować dyrektywę.

```
| using namespace pieces;
| cin >> fetch;
```

Dyrektywa `using` ma cechę **przechodności**. Operator `#` jest przechodnim, kiedy `X # Y` i `Y # Z` implikuje `X # Z`.

Przykładem operatora przechodniego jest operator większości `>`. Jeśli `X` jest większe od `Y`, a `Y` od `Z`, to na pewno `X` jest większe od `Z`.

Wynika z tego, że instrukcja:

```
using namespace pieces;
```

daje efekt w postaci lokalnego udostępnienia przestrzeni nazw `elements` za pośrednictwem dyrektywy `using` z przestrzeni nazw `pieces`, a więc jest równoważne sekwencji:

```
using namespace pieces;  
using namespace elements;
```

Aliasy przestrzeni nazw

Dla przestrzeni nazw można zdefiniować aliasy.

```
namespace myFavoriteThing
{
    ...
}
```

```
namespace mft = myFavoriteThing;
```

Pozwala to na upraszczanie stosowania zagnieżdżonych nazw.

```
namespace eh = elements::hot;

using eh::fire;
```

Alias może też wskazywać przestrzeń udostępnioną dyrektywą `using`:

```
namespace ph = pieces::hot;           ← hot jest widoczne w pieces dzięki dyrektywie
namespace ph = pieces::elements::hot; ← błąd, elements nie jest zagnieżdżona w pieces
```

Nienazwane przestrzenie nazw

Jeżeli w deklaracji przestrzeni nazw zostanie pominięta nazwa, zostanie utworzona **nienazwana przestrzeń nazw**.

```
namespace  
{  
    int ice;  
    int water;  
}
```

Kod taki zachowuje się tak, jakby był uzupełniony **dyrektywą using**. Nazwy deklarowane w tej przestrzeni nazw znajdują się w potencjalnym zasięgu aż do końca obszaru deklaracyjnego zawierającego nienazwaną przestrzeń nazw.

Nazwy w nienazwanej przestrzeni nazw zachowują się więc jak nazwy globalne.

Skoro przestrzeń nie ma nazwy, nie można zastosować wobec niej jawnie **dyrektywy using** ani **deklaracji using**, udostępniających zawartość przestrzeni nazw poza nią.

W szczególności nie można stosować nazw z nienazwanej przestrzeni nazw w pliku innym niż plik zawierający deklarację tej przestrzeni nazw.

Takie cechy nienazwanych przestrzeni nazw czynią je alternatywą dla statycznych zmiennych podlegających łączeniu wewnętrznemu.

Standard C++98 oznaczał stosowanie słowa `static` w przestrzeniach nazw i zasięgu globalnym jako przestarzałe, zalecając w zamian nienazwane przestrzenie nazw. Od C++11 oznaczenie to wycofano – obie formy są równoprawne.

Nienazwana przestrzeń nazw ma jednak tę przewagę, że obejmuje również **typy** (klasy, struktury, wyliczenia), do których słowa `static` nie da się zastosować.

Dobre zwyczaje stosowania przestrzeni nazw:

- zamiast zewnętrznych zmiennych globalnych należy stosować zmienne w nazwanych przestrzeniach nazw,
- zamiast statycznych zmiennych globalnych należy stosować zmienne w nienazwanych przestrzeniach nazw,
- tworząc bibliotekę funkcji lub klas, należy umieszczać jej elementy w przestrzeni nazw,
- dyrektywę `using` powinno się stosować jedynie w roli tymczasowego środka importu,
- nie powinno się stosować dyrektywy `using` w plikach nagłówkowych (na zachowanie programu wpływ może mieć kolejność włączania plików nagłówkowych); jeśli jest to konieczne, należy stosować dyrektywę `using` w pliku nagłówkowym za wszystkimi dyrektywami `include` preprocesora,
- do nazw z przestrzeni nazw powinno się odwoływać za pośrednictwem deklaracji i operatora zakresu.

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony**
- 14 Podstawy STL

Szablon dla funkcji

Szablon funkcji (*function template*) jest ogólną postacią funkcji, podającą przepis na utworzenie konkretnej funkcji.

```
template <typename T> // template <typename T> T add(T a, T b)
T add(T a, T b)
{
    return a + b;
}
```

Deklarację szablonu funkcji poprzedza specyfikacja:

```
template <typename name>
```

lub równoważna, starsza forma:

```
template <class name>
```

Oba zapisy znaczą dokładnie to samo. Słowo `typename` wprowadzono później, ponieważ lepiej oddaje sens – parametrem szablonu może być dowolny typ, także wbudowany (np. `int`), a nie tylko klasa.

Identyfikator `name`, w przypadku funkcji `add` to `T`. Reprezentuje oznaczenie **nieokreślonego typu**.

Typ zostanie określony przez kompilator przy wywołaniu funkcji, co spowoduje wygenerowanie jej skonkretyzowanej wersji.

Wywołanie funkcji szablonej

Na etapie wywołania funkcji szablonej, programista określa typ na podstawie którego kompilator tworzy postać funkcji skonkretyzowanej.

```
cout << add<int>(1, 2) << endl;      ← 3  
cout << add<double>(1.2, 2.5) << endl; ← 3.7
```

Konkretyzację typu można pominąć, kompilator sam określi typy na etapie wywołania.

```
cout << add(1, 2) << endl;      ← 3  
cout << add(1.2, 2.5) << endl; ← 3.7
```

Pozwala to uniknąć niejednoznaczności (niektóre kompilatory mogą wyświetlić ostrzeżenia), np.:

```
cout << add<int>(1.2, 2.5); ← 3
```

Funkcja z szablonu może być konkretyzowana nie tylko dla typów wbudowanych.

```
TComplex a(1, 3);
TComplex b(4, 4);
TComplex c;

c = add(a, b);
cout << c.getReal() << " + " << c.getImag() << "i" << endl;  ← 5 + 7i
```

Przy wcześniej zdefiniowanym szablonie funkcji `add`, wywołanie:

```
add(1.2, 2);  ← error: no matching function for call to 'add(double, int)'
              note: candidate: 'template<class T> T add(T, T)'
```

Szablon funkcji może posiadać więcej niż jeden typ uogólniony.

```
template <typename T1, typename T2> T1 add(T1 a, T2 b)
{
    return a + b;
}
```

Przy tak zdefiniowanym szablonie funkcji `add`, wywołanie będzie poprawne:

```
add(1.2, 2);    ← 3.2
```

jednak takie wywołanie:

```
add(2, 1.2);    ← 3
```

spowoduje konwersję typu rezultatu, które może być sygnalizowane przez niektóre kompilatory ostrzeżeniem.

Problem z typem rezultatu można rozwiązać, pozwalając kompilatorowi samodzielnie go wyznaczyć (od C++14):

```
template <typename T1, typename T2> auto add(T1 a, T2 b)
{
    return a + b; ← typ rezultatu wynika z typu wyrażenia a + b
}
```

Teraz oba wywołania dają ten sam wynik:

```
add(1.2, 2); ← 3.2 - rezultat typu double
add(2, 1.2); ← 3.2 - rezultat typu double
```

Słowo `auto` w miejscu typu rezultatu oznacza, że kompilator wyznaczy go na podstawie wyrażenia w instrukcji `return`. Typ wyrażenia `a + b` wynika z reguł konwersji – dla `int` i `double` jest to `double`.

W C++11 ten sam efekt wymaga jawnego wskazania typu po strzałce:

```
template <typename T1, typename T2> auto add(T1 a, T2 b) -> decltype(a + b)
```

Szablon klasy na przykładzie

Potrzebujemy wygodnej, dynamicznej tablicy liczb całkowitych o zmiennym rozmiarze. Chcemy używać jej w sposób:

```
int n;

cout << "Długość tablicy: ";
cin >> n;

TArrayOfInt seq(n);

for(int i = 0; i < 4; i++)
    seq.addItem(i);

for(int i = 0; i < seq.getLength(); i++)
    cout << seq[i] << ' ';

cout << endl << "Dodaje elementy: " << endl;

for(int i = 2; i < 5; i++)
    seq.insertItem(i, 10 + i);

for(int i = 0; i < seq.getLength(); i++)
    cout << seq[i] << ' ';
```



```
Dlugosc tablicy: 10
0 1 2 3
Dodaje elementy:
0 1 12 13 14 2 3
```

Okazało się, że potrzebna jeszcze będzie tablica dla typu zmiennoprzecinkowego.

Zastosowanie **szablonów** pozwoli na **uogólnienie** zdefiniowanej klasy na inne **typy**.

Skonkretyzowanej tablicy dla typu `int` powstałej w oparciu o szablon można używać tak:

```
int n;

cout << "Dlugosc tablicy: ";
cin >> n;

TArray<int> seq(n);

for(int i = 0; i < 4; i++)
    seq.addItem(i);

for(int i = 0; i < seq.getLength(); i++)
    cout << seq[i] << ' ';

cout << endl << "Dodaje elementy: " << endl;

for(int i = 2; i < 5; i++)
    seq.insertItem(i, 10 + i);

for(int i = 0; i < seq.getLength(); i++)
    cout << seq[i] << ' ';
```

Tablica dla typu `float`:

```
int n;

cout << "Dlugosc tablicy: ";
cin >> n;

TArray<float> seq(n);

for(int i = 0; i < 4; i++)
    seq.addItem(i);

for(int i = 0; i < seq.getLength(); i++)
    cout << showpoint << seq[i] << ' ';

cout << endl << "Dodaje elementy: " << endl;

for(int i = 2; i < 5; i++)
    seq.insertItem(i, 10 + i);

for(int i = 0; i < seq.getLength(); i++)
    cout << showpoint << seq[i] << ' ';
```

```
Dlugosc tablicy: 10
```

```
0.00000 1.00000 2.00000 3.00000
```

```
Dodaje elementy:
```

```
0.00000 1.00000 12.0000 13.0000 14.0000 2.00000 3.00000
```

Taki szablon klasy będzie mógł być konkretyzowany dla własnych typów.

```
int n;

cout << "Dlugosc tablicy: ";
cin >> n;

TArray<TComplex> seq(n);

for(int i = 0; i < 4; i++)
    seq.addItem(TComplex(i, i));

for(int i = 0; i < seq.getLength(); i++)
    cout << seq[i] << ' ';

cout << endl << "Dodaje elementy: " << endl;

for(int i = 2; i < 5; i++)
    seq.insertItem(i, TComplex(10 + i, 10 + i));

for(int i = 0; i < seq.getLength(); i++)
    cout << seq[i] << ' ';
```

```
Dlugosc tablicy: 10
```

```
(0, 0i) (1, 1i) (2, 2i) (3, 3i)
```

```
Dodaje elementy:
```

```
(0, 0i) (1, 1i) (12, 12i) (13, 13i) (14, 14i) (2, 2i) (3, 3i)
```

Implementacja szablonu klasy

Tablica będzie przechowywać elementy uogólnionego typu `T`.

```
template <typename T> class TArray
{
    public:
        ...

    private:
        T *arr;           ← wskaźnik na tablicę obiektów typu T
        int length;       ← liczba elementów w tablicy
        int maxLength;    ← rozmiar tablicy
        ...
};
```

Akcesory i przydatne funkcje:

```
template <typename T> class TArray
{
public:
    int getLength() const;      ← pobranie aktualnej liczby elementów
    int getMaxLength() const;  ← pobranie maksymalnej liczby elementów
    ...

private:
    void arrInit(int arrSize);  ← tworzenie tablicy
    void arrDelete();           ← kasowanie tablicy
    void arrCopy(const TArray &z); ← kopiowanie tablicy
    ...
};
```

Funkcje `arrInit()` i `arrCopy()` wywoływane są wyłącznie dla obiektu, który nie posiada tablicy – w konstruktorach lub po `arrDelete()`.


```
template <typename T> int TArray<T>::getLength() const
{
    return length;
}
```

```
template <typename T> int TArray<T>::getMaxLength() const
{
    return maxLength;
}
```

```
template <typename T> void TArray<T>::arrInit(int arrSize)
{
    arr = nullptr;
    length = 0;
    maxLength = 0;

    if(arrSize > 0)
    {
        arr = new T[arrSize];
        maxLength = arrSize;
    }
}
```

```
template <typename T> void TArray<T>::arrDelete()  
{  
    delete [] arr;    ← dla nullptr nic się nie stanie  
    arr = nullptr;  
    length = 0;  
    maxLength = 0;  
}
```

```
template <typename T> void TArray<T>::arrCopy(const TArray &z)  
{  
    arrInit(z.maxLength);    ← nowa tablica (lub nullptr)  
  
    for(int i = 0; i < z.length; i++)    ← kopiowanie tylko zajętych elementów  
        arr[i] = z.arr[i];  
  
    length = z.length;  
}
```

Konstruktory i destruktory:

```
template <typename T> class TArray
{
public:
    TArray(int arrSize = 10); ← konstruktor ogólny i zarazem domyślny
    TArray(const TArray &z); ← konstruktor kopiujący
    ~TArray(); ← destruktory
    ...

private:
    ...
};
```

Klasa zarządza pamięcią dynamiczną, dlatego – zgodnie z **regułą trzech** – posiada własny destruktory, konstruktor kopiujący i operator przypisania.

```
template <typename T> TArray<T>::TArray(int arrSize)
{
    arrInit(arrSize);
}
```

```
template <typename T> TArray<T>::TArray(const TArray &z)
{
    arrCopy(z);
}
```

```
template <typename T> TArray<T>::~TArray()
{
    arrDelete();
}
```

Przeciążone operatory:

```
template <typename T> class TArray
{
public:
    TArray & operator =(const TArray &z);    ← operator przypisania tablic
    T & operator [](int k);                  ← operator indeksowania
    const T & operator [](int k) const;      ← indeksowanie obiektów stałych
    ...

private:
    ...
};
```

Wersja `const` operatora indeksowania pozwala odczytywać elementy tablicy przekazanej przez referencję do stałej, np. `const TArray<int> &`.

```
template <typename T> TArray<T> & TArray<T>::operator =(const TArray &z)
{
    if(this != &z)
    {
        arrDelete();
        arrCopy(z);
    }

    return *this;
}
```

```
template <typename T> T & TArray<T>::operator [](int k)
{
    return arr[k];
}
```

```
template <typename T> const T & TArray<T>::operator [](int k) const
{
    return arr[k];
}
```

Dodawanie i usuwanie elementów tablicy:

```
template <typename T> class TArray
{
public:
    void addItem(T item);           ← dopisanie elementu na koniec tablicy
    void deleteItem();              ← skasowanie ostatniego elementu tablicy
    void insertItem(int k, T item); ← wstawienie elementu na pozycji (z przesuwaniem)
    void removeItem(int k);         ← skasowanie elementu z pozycji (z przesuwaniem)
    ...

private:
    ...
};
```

```
template <typename T> void TArray<T>::addItem(T item)
{
    if(length < maxLength)
    {
        arr[length] = item;
        length++;
    }
}
```

```
template <typename T> void TArray<T>::deleteItem()
{
    if(length > 0)
        length--;
}
```



```

template <typename T> void TArray<T>::insertItem(int k, T item)
{
    if(k >= 0 && k < length && length < maxLength) ← musi być miejsce w tablicy
    {
        for(int i = length - 1; i >= k; i--) ← przesunięcie od ostatniego
            arr[i + 1] = arr[i];             elementu

        arr[k] = item;
        length++;
    }
}

```

```

template <typename T> void TArray<T>::removeItem(int k)
{
    if(k >= 0 && k < length)
    {
        for(int i = k; i < length - 1; i++)
            arr[i] = arr[i + 1];

        length--;
    }
}

```

Iterowanie po wszystkich elementach:

```
template <typename T> class TArray
{
public:
    void doForEach(void (* f)(T &z)); ← wykonuje funkcję f(T &z) dla wszystkich
                                     elementów tablicy

private:
    ...
};
```

```
template <typename T> void TArray<T>::doForEach(void (* f)(T &z))
{
    for(int i = 0; i < length; i++)
        f(arr[i]);
}
```

Szablony a pliki nagłówkowe

Szablon nie jest gotowym kodem, lecz przepisem na jego utworzenie. Kompilator generuje kod klasy (konkretyzuje szablon) dopiero w miejscu użycia, np.

`TArray<int>`, i musi wtedy widzieć **pełną definicję** szablonu – także definicje metod.

Dlatego definicje metod szablonu klasy umieszcza się w pliku nagłówkowym, razem z definicją klasy, poniżej treść pliku nagłówkowego `tarray.h`:

```
#ifndef TARRAY_H
#define TARRAY_H

template <typename T> class TArray
{
    ...
};

template <typename T> TArray<T>::TArray(int arrSize)
{
    arrInit(arrSize);
}
...

#endif
```

Jeżeli definicje metod trafią do osobnego pliku `tarray.cpp`, jak w przypadku zwykłych klas, kompilacja przebiegnie poprawnie, ale konsolidator zgłosi błąd (g++):

```
| undefined reference to 'TArray<int>::TArray(int)'
```

W przypadku MSVC będzie to błąd:

```
| LNK2019: unresolved external symbol
```

Przykład z szablonem funkcji:

```
template <typename T> void printItem(T &item)
{
    cout << "Item = " << item << endl;
}
```

```
TArray<int> seq(10);

seq.addItem(1);
seq.addItem(2);
seq.addItem(3);

seq.doForEach(printItem<int>);
```

Przy takiej funkcji typ **T**, dla którego konkretyzujemy klasę i funkcję, musi mieć zdefiniowany operator **<<**.

```
int main()
{
    int n;
    cout << "Dlugosc tablicy: ";
    cin >> n;

    TArray<TComplex> seq(n);

    seq.addItem(TComplex(1, 1));
    seq.addItem(TComplex(2, 2));

    cout << "Liczba elementow: " << seq.getLength() << endl;

    for(int i = 0; i < seq.getLength(); i++)
        cout << seq[i] << endl;

    for(int i = 3; i <= 6; i++)
        seq.addItem(TComplex(i, i));

    cout << "Dlugosc tablicy: " << seq.getMaxLength() << endl;
    cout << "Liczba elementow: " << seq.getLength() << endl;

    seq.doForEach(printItem<TComplex>);

    return 0;
}
```

```
Dlugosc tablicy: 10  
Liczba elementow: 2  
(1, 1i)  
(2, 2i)  
Dlugosc tablicy: 10  
Liczba elementow: 6  
Item = (1, 1i)  
Item = (2, 2i)  
Item = (3, 3i)  
Item = (4, 4i)  
Item = (5, 5i)  
Item = (6, 6i)
```

- 1 Informacje ogólne
- 2 Środowisko programowania
- 3 Łagodny start
- 4 Jednostki leksykalne i typy danych
- 5 Więcej o programowaniu orientowanym obiektowo
- 6 Klasy i konstruktory
- 7 Dziedziczenie
- 8 Polimorfizm
- 9 Wyjątki i ich obsługa
- 10 Zarządzanie pamięcią i obiekty
- 11 Funkcje i operatory przeciążone
- 12 Przestrzenie nazw
- 13 Szablony
- 14 Podstawy STL**

Biblioteka STL

Standard Template Library, STL – biblioteka C++ zawierająca algorytmy, kontenery, iteratory oraz inne konstrukcje w formie szablonów, gotowe do użycia w programach.

STL jest to tzw. biblioteka generyczna, oznacza to, że jej komponenty są parametryzowane, niemal każdy z nich jest szablonem.

Umożliwia to równie dobrą współpracę z typami wbudowanymi w język, z typami wbudowanymi w bibliotekę, co z typami zdefiniowanymi przez użytkownika, pod warunkiem, że spełniają pewne określone warunki.

Komponenty biblioteki STL:

- **kontenery** (*containers*) – stosowane do zarządzania zbiorami obiektów. Kontenery mogą być implementowane jako tablice, listy powiązane, kolejki, mapy.
- **iteratory** (*iterators*) – stosowane do przeszukiwania kontenerów (zbiorów obiektów), przechodzenia między obiektami.
- **algorytmy** – określają operacje wykonywane na zawartości kontenerów lub na obiektach.

Biblioteka STL nie jest przykładem programowania obiektowego. Reprezentuje odmienny paradygmat nazwany **programowaniem generycznym** lub **programowaniem uogólnionym**.

Programowanie uogólnione pozwala na pisanie kodu programu, w językach typowanych statycznie, bez wcześniejszej znajomości typów danych, na których kod ten będzie pracował.

Obecnie wiele języków programowania ma możliwość wykorzystywania uogólnień, np.: **C++**, **D**, **Java** oraz **Haskell**.

Kontenery

Jedną z najważniejszych rzeczy wprowadzonych przez STL, są **kontenery**. Kontenery służą do **przechowywania** i zarządzania **kolekcjami danych**.

Jest ich kilka rodzajów, różnią się konstrukcją i wydajnością poszczególnych operacji. Np. kontener typu **vector** trzyma obiekty w liniowym obszarze pamięci, co umożliwia swobodny dostęp (*random access*) do wszystkich elementów.

Można ten kontener indeksować liczbą całkowitą, podobnie jak robi to się ze zwykłymi tablicami.

Niestety, wstawienie nowego elementu gdziekolwiek indziej niż na końcu jest operacją liniowego czasu, gdyż trzeba *przesunąć* elementy, żeby zrobić miejsce na nowy.

Z kolei w kontenerze typu **list** wstawianie i usuwanie elementów w miejscu wskazanym iteratorem jest operacją o stałym czasie wykonania, ale nie jest możliwe indeksowanie – dotarcie do wybranej pozycji wymaga przejścia po kolejnych elementach.

Typy kontenerów:

- **kontenery sekwencyjne** – gdzie każdy element ma określoną pozycję, np.: `array`, `vector`, `deque`, `list`, `forward_list`; kontenery sekwencyjne są zbudowane na tablicach (o stałym lub zmiennym rozmiarze) albo na listach,
- **kontenery uporządkowane** (w tym asocjacyjne) – gdzie pozycja elementu zależy od jego wartości i kryterium sortowania, np.: `set`, `multiset`, `map`, `multimap`; kontenery uporządkowane są zbudowane na zrównoważonych drzewach BST,
- **kontenery nieuporządkowane** – gdzie pozycja elementu zależy od **skrót**u (*hash*) jego wartości, a nie od kolejności wartości, np.: `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`; kontenery nieuporządkowane są zbudowane na tablicach z haszowaniem.

Kontenery sekwencyjne są kolekcjami uporządkowanymi, w których każdy element posiada określoną pozycję. Pozycja ta zależy od momentu oraz miejsca wstawienia, jest jednak niezależna od wartości elementu.

Kontenery asocjacyjne są kolekcjami sortowanymi, w których aktualna pozycja elementu zależy od jego wartości (albo klucza w przypadku kontenerów operujących na parach klucz-wartość), zgodnie z określonym kryterium sortowania.

Kontenery asocjacyjne nieporządkujące to kolekcje nieporządkujące i niezachowujące pozycji elementów, bo ich zadaniem głównym jest ustalanie, czy (a nie gdzie) element znajduje się w kolekcji. Elementy nie zachowują więc uporządkowania ani względem kolejności wstawiania, ani względem wartości – jedno i drugie może w czasie życia kontenera ulegać zmianie.

Elementy kontenerów

Wymagania wobec elementów zależą od operacji wykonywanych na kontenerze. W większości przypadków element musi być:

- **zniszczalny** (publiczny destruktork),
- **kopiowalny** lub co najmniej **przenaszalny** (konstruktor kopiujący lub przenoszący, przypisanie kopiujące lub przenoszące).

Do C++11 elementy musiały być kopiowalne. Obecnie wystarczy możliwość przeniesienia, dlatego w kontenerze można przechowywać np. obiekty `unique_ptr`, których kopiować nie wolno: `vector<unique_ptr<TFigure>>`.

W pewnych sytuacjach elementy kontenerów muszą spełniać wymagania dodatkowe:

- **konstruktor domyślny** (utworzenie niepustego kontenera, np. `vector<T> v(10)`),
- **operator porównywania ==** (wyszukiwanie),
- **operator porównywania <** (kryterium sortowania, kontenery uporządkowane),
- **funkcja skrót** i operator `==` (kontenery nieuporządkowane).

Semantyka wartości

Kontenery STL realizują **semantykę wartości**: przechowują **własne kopie** wstawianych elementów. Obiekt wstawiony do kontenera jest kopiowany (lub przenoszony), a późniejsza zmiana oryginału nie wpływa na element w kontenerze.

Dostęp do elementów (**operator []**, **at()**, **front()**, **back()**, iteratory) odbywa się przez **referencje** – zmiana elementu odczytanego w ten sposób zmienia obiekt przechowywany w kontenerze.

Semantykę **referencji** uzyskuje się, przechowując w kontenerze **inteligentne wskaźniki** do obiektów zamiast samych obiektów.

Inteligentne wskaźniki

Inteligentny wskaźnik to obiekt zachowujący się jak wskaźnik (operatory `*` i `->`), który sam zwalnia wskazywany obiekt – zgodnie z zasadą RAII. Klasy zdefiniowane są w nagłówku `memory`.

`unique_ptr<>` – **wyłączny** właściciel obiektu. Obiekt jest usuwany razem ze wskaźnikiem.

```
{
    unique_ptr<TSquare> p = make_unique<TSquare>(10);    ← zamiast new TSquare(10)

    cout << p->area() << endl;

    unique_ptr<TSquare> q = p;                          ← błąd - kopiowanie zabronione
    unique_ptr<TSquare> q = move(p);                    ← przeniesienie własności, p jest teraz pusty
                                                        ← ~TSquare() - bez delete
}
```

Kopiowanie `unique_ptr` jest zabronione (= `delete`) – dwa wskaźniki usunęłyby ten sam obiekt. Własność można jedynie **przenieść** funkcją `move()` (nagłówek `utility`).

Funkcja `make_unique()` jest dostępna od C++14.

`shared_ptr` – **współdzielona** własność. Wskaźnik zlicza swoje kopie (*licznik referencji*), obiekt jest usuwany, gdy zniknie ostatnia z nich.

```
shared_ptr<TSquare> a = make_shared<TSquare>(5);
{
    shared_ptr<TSquare> b = a;           ← kopiowanie dozwolone
    cout << a.use_count() << endl;     ← 2
}
cout << a.use_count() << endl;       ← 1 - b przestał istnieć
```

Kontenery wskaźników `shared_ptr` realizują semantykę referencji – kilka kontenerów może odwoływać się do tego samego obiektu:

```
vector<shared_ptr<TSquare>> v1, v2;

v1.push_back(a);
v2.push_back(a);

v1[0]->setSide(7);
cout << v2[0]->area() << endl;       ← 49 - ten sam obiekt
```

W praktyce domyślnym wyborem jest `unique_ptr`, a `shared_ptr` stosuje się wtedy, gdy obiekt naprawdę ma kilku właścicieli.

Wspólne cechy i operacje

Wszystkie kontenery zapewniają **semantykę wartości**.

Wszystkie elementy posiadają określoną kolejność (kontenery udostępniają operacje zwracające iteratory służące do iteracji po kolejnych elementach).

Operacje na kontenerach nie są **bezpieczne**, czyli nie sprawdzają możliwości wystąpienia każdego rodzaju błędu (to funkcja wywołująca musi zapewnić spełnienie wymagań przez parametry operacji).

Wspólne operacje:

- konstruktor domyślny, kopiujący, przenoszący, zakresowy (kopiuje elementy z innej kolekcji z podanego za pomocą iteratorów zakresu), z listą wartości,
- destruktor (usuwa wszystkie wartości),
- przypisanie kopiujące i przenoszące,
- funkcje składowe `empty()`, `size()`, `clear()`, `swap()`,
- funkcja globalna `swap()`,
- operatory relacyjne `==`, `!=`, `<`, `<=`, `>`, `>=`,
- iteratory `begin()`, `end()`, `cbegin()`, `cend()`.

Od tej reguły są wyjątki: `array` nie ma `clear()` (ma stały rozmiar), `forward_list` nie ma `size()`, a kontenery nieuporządkowane mają tylko operatory `==` i `!=`.

Tablice

Jako tablicę można potraktować egzemplarz klasy `array<>`.

Tablica taka modeluje tablicę o **stałym rozmiarze** znaną z języka C, zapewniając jednak interfejs kontenera STL. Rozmiar jest parametrem szablonu i musi być znany na etapie kompilacji.

Elementy przechowywane są bezpośrednio w obiekcie, bez przydzielenia pamięci dynamicznej – tablica lokalna leży na stosie, jak zwykła tablica.



stała liczba elementów

Tablica `array<>`, podobnie jak zwykła tablica, **nie inicjuje** elementów typów wbudowanych – tablica lokalna bez inicjalizatora ma wartości przypadkowe. Pusty inicjalizator `{}` zeruje wszystkie elementy.

```
array<int, 5> arr;      ← wartości przypadkowe
array<int, 5> arr{};   ← wszystkie elementy równe 0
```

Wykorzystanie inicjalizacji zbiorczej – brakujące elementy są zerowane:

```
array<int, 5> arrA{1, 2, 3};    ← 1, 2, 3, 0, 0
array<int, 3> arrB = {4, 5, 6};
array<string, 3> arrS = {string("ala"), "ma", "psa"};
```

W C++11 inicjalizacja bez znaku `=` wymagała podwójnych klamer: `arrA{{1, 2, 3}}`. Od C++14 wystarczają pojedyncze.

Sortowanie kontenera:

```
sort(arrA.begin(), arrA.end());
```

Wyświetlenie zawartości kontenera wspak:

```
reverse_copy(arrB.begin(), arrB.end(), ostream_iterator<int>(cout, " "));
```

Zawartość z użyciem pętli:

```
for(const auto &e: arrA)
    cout << e << " ";

for(size_t i = 0; i < arrS.size(); i++)
    cout << arrS[i] << " ";
```

Funkcja `size()` zwraca wartość typu bez znaku (`size_t`), dlatego licznik pętli ma ten sam typ – porównanie `int` z `size_t` kompilator zgłasza ostrzeżeniem.

Wektory

Wektor `vector<>` przechowuje swoje elementy w tablicy dynamicznej.

Szybki dostęp do każdego elementu za pomocą indeksowania.

Dołączanie i usuwanie elementów na końcu wektora jest bardzo szybkie, ale wstawienie lub usunięcie elementu ze środka zabiera więcej czasu.



Deklaracja, wstawianie do wektora i zawartość wektora:

```
vector<int> vecA;  
  
int n = 5;  
  
for(int i = 0; i < n; i++)  
    vecA.push_back(i + 1);  
  
for(size_t i = 0; i < vecA.size(); i++)  
    cout << vecA[i] << " ";
```

Usunięcie pierwszego elementu wektora, usunięcie ostatnich elementów, usunięcie parzystych elementów:

```
vector<int> vecB {0, 1, 2, 3, 4, 5, 6, 7, 8};

vecB.erase(vecB.begin());           ← vecB = 1, 2, 3, 4, 5, 6, 7, 8

vecB.erase(vecB.end() - 3, vecB.end()); ← vecB = 1, 2, 3, 4, 5

for(auto it = vecB.begin(); it != vecB.end(); ) ← vecB = 1, 3, 5
    if(*it % 2 == 0)
        it = vecB.erase(it);
    else
        it++;
```

Kolejka o dwóch końcach

Kolejka o dwóch końcach `deque<>` przechowuje swoje elementy w kilku blokach pamięci (tablicach), dzięki czemu może rosnąć w obie strony bez przenoszenia istniejących elementów.

Uzyskujemy szybki dostęp do każdego elementu za pomocą indeksowania.

Dołączanie i usuwanie elementów na końcu i na początku kolejki jest bardzo szybkie, ale wstawienie lub usunięcie elementu ze środka zabiera więcej czasu.



Deklaracja, wstawianie do kolejki i zawartość kolejki:

```
deque<float> que;  
  
int n = 5;  
  
for(int i = 0; i < n; i++)    ← que = 12.56, 9.42, 6.28, 3.14, 0  
    que.push_front(i * 3.14);  
  
for(size_t i = 0; i < que.size(); i++)  
    cout << que[i] << " ";
```

Lista

Lista `list<>` przechowuje swoje elementy w liście dwukierunkowej.

W listach nie ma swobodnego dostępu do elementów kolekcji.

Dołączanie i usuwanie elementów na końcu i na początku listy jest bardzo szybkie, ale dostanie się do elementu ze środka zabiera dużo czasu.



Deklaracja, wstawianie do listy, użycie i zdejmowanie elementów listy:

```
list<char> lis;  
  
for(char c = 'a'; c <= 'f'; c++)  
    lis.push_back(c);  
  
while(!lis.empty())  
{  
    cout << lis.front() << " ";  
    lis.pop_front();  
}
```

Lista jednokierunkowa

Lista `forward_list<>` przechowuje swoje elementy w liście jednokierunkowej.

W listach nie ma swobodnego dostępu do elementów kolekcji.

Na listach jednokierunkowych można iterować tylko do przodu.

Dołączanie i usuwanie elementów na początku listy jest bardzo szybkie, ale dostanie się do elementu ze środka zabiera dużo czasu. Lista jednokierunkowa nie udostępnia operacji na końcu – nie przechowuje informacji o ostatnim elemencie.



Deklaracja, wstawianie do listy, zmiana wielkości listy i zawartość listy:

```
forward_list<long> flis = {2, 3, 5, 7, 11, 13};  
  
flis.resize(9);    ← flis = 2, 3, 5, 7, 11, 13, 0, 0, 0  
  
for(auto e: flis)  
    cout << e << " ";
```


Łańcuchy i tablice

Obiektów klas łańcuchowych, czyli `basic_string<>`, `string` i `wstring` można używać jak kontenerów sekwencyjnych.

Są bardzo podobne do wektorów.

Innym rodzajem kontenera może być **tablica**.

Nie jest to klasa i nie ma żadnych metod, ale konstrukcja STL umożliwia uruchamianie na tablicach różnych algorytmów (tak jak na kontenerach).

Kontenery uporządkowane

Kontenery uporządkowane wykonują automatycznie sortowanie swoich elementów. Kontenery uporządkowane są implementowane w postaci zrównoważonych drzew BST.

Asocjacyjne kontenery uporządkowane przechowują pary klucz-wartość (odpowiednio `first` i `second`) i sortowanie następuje po kluczach.

Domyślnie elementy lub klucze są porządkowane przy pomocy operatora `<`.

Wszystkie kontenery uporządkowane posiadają domyślny parametr wzorca służący sortowaniu (domyślnym jest operator `<`).

Rodzaje kontenerów:

- zbiory `set<>`,
- wielozbiory `multiset<>`,
- mapy `map<>`,
- multimapy `multimap<>`.

Deklaracja, wstawianie do wielozbioru i zawartość wielozbioru:

```
multiset<string, greater<string>> cities
{
    "Rzepin",
    "Ambrozow"
};

cities.insert({"Kielce", "Krakow"}); ← cities = Rzepin, Krakow, Kielce, Ambrozow

for(auto e: cities)
    cout << e << " ";
```

Kontenery nieuporządkowane

Kontenery nieuporządkowane przechowują elementy w sposób nieuporządkowany (brak kryterium sortowania).

Asocjacyjne kontenery nieuporządkowane przechowują pary klucz-wartość (odpowiednio `first` i `second`).

Kontenery nieuporządkowane są implementowane w postaci tablic z haszowaniem.

Rodzaje kontenerów:

- zbiory `unordered_set<>`,
- wielozbiory `unordered_multiset<>`,
- mapy `unordered_map<>`,
- multimapy `unordered_multimap<>`.

Deklaracja, wstawianie do mapy i zawartość mapy:

```
unordered_map<string, double> umap
{
    {"bolek", 2.1},
    {"lolek", 3.1}
};

for(pair<const string, double> &e: umap) ← umap = 9.61 4.41 lub 4.41 9.61
    e.second *= e.second;

for(pair<const string, double> &e: umap)
    cout << e.second << " ";
```

Kolejność elementów w kontenerze nieuporządkowanym zależy od funkcji mieszającej i nie jest określona – inny kompilator lub inna liczba elementów może dać inną kolejność. Program nie powinien na niej polegać.

Adaptery kontenerów

Adaptery kontenerów to kontenery wykorzystujące ogólną strukturę innych kontenerów do realizacji pewnych specyficznych potrzeb.

Adapterami kontenerów są stosy `stack<>`, kolejki `queue<>` i kolejki priorytetowe `priority_queue<>`.

Przy definiowaniu takich kontenerów można podać jako drugi parametr typ kontenera do realizacji struktury. Domyślnie stos i kolejka korzystają z `deque<>`, a kolejka priorytetowa z `vector<>`:

```
stack<int> sta;                ← stack<int, deque<int>>
stack<int, vector<int>> stv;    ← stos zbudowany na wektorze
```

```
queue<double> que;            ← queue<double, deque<double>>
```

```
priority_queue<string> pqe;    ← priority_queue<string, vector<string>>
priority_queue<string, deque<string>, greater<string>> pqmin; ← najmniejszy na szczycie
```

Iteratory

Iterator to specjalny obiekt, który potrafi iterować po elementach kolekcji.

Iterator daje możliwość dotarcia do składnika kontenera bez konieczności znajomości jego struktury.

Iterator ma zaimplementowaną **semantykę wskaźnika** – posiada operator **wyłuskania** elementu `*`.

Wszystkie kontenery udostępniają funkcje tworzące iteratory do nawigowania po ich elementach.

Funkcja `begin( )` zwraca iterator wskazujący na pozycję z pierwszym elementem w kolekcji.

Funkcja `end( )` zwraca iterator wskazujący pozycję za ostatnim elementem.

Każdy kontener definiuje dwa typy iteratorów:

- `kontener::iterator` przeznaczony do iterowania po elementach z możliwością odczytu i zapisu,
- `kontener::const_iterator` przeznaczony do iterowania po elementach tylko z możliwością odczytu.

Podział iteratorów

Iteratory wejścia

Taki iterator może odczytać wartość elementu, na który wskazuje, o ile nie wskazuje przekroczenia zakresu.

Musi posiadać domyślny konstruktor, konstruktor kopiujący oraz operatory: `=`, `==`, `!=`, `++`.

```
| value = *iterator;
```

```
| iterator++;
```

```
| ++iterator;
```

Iteratory wyjścia

Ten typ iteratora umożliwia tylko zapis wartości do danego składnika – odczyt jest niemożliwy.

Musi posiadać domyślny konstruktor, konstruktor kopiujący oraz operatory: `=`, `++`.

```
| *iterator = value;
```

Iterator przejścia w przód

Jest to połączenie iteratora wejścia i wyjścia.

Posiada domyślny konstruktor, konstruktor kopiujący oraz operatory: `=`, `==`, `!=`, `++`.

Możliwy jest zarówno zapis, jak i odczyt. Przesuwanie się po kolejnych składnikach tak jak poprzednio jest możliwe tylko w przód poprzez inkrementację iteratora.

Iterator dwukierunkowy

Różni się tylko możliwością dekrementacji iteratora od iteratorów przejścia w przód.

Iterator bezpośredniego dostępu

Ten typ dziedziczy wszystko po iteratorach dwukierunkowych, przy czym posiada możliwość dostępu bezpośredniego do wybranego składnika bez potrzeby skanowania struktury.

Z racji tego, że można przeskoczyć o większą liczbę składników niż jeden, iteratory te posiadają operatory: `+`, `+=`, `-`, `-=`, `[]`. A także dodatkowo operatory: `<`, `>`, `<=`, `>=`.

```
| iterator += 5;
```

```
| iterator++;
```

```
| iterator[n] = wartosc;
```

```
vector<int> arr = {1, 2, 3};  
  
vector<int>::iterator it;  
  
for(it = arr.begin(); it != arr.end(); it++)  
    cout<< *it << " ";           ← 1 2 3  
  
vector<int>::reverse_iterator rit;  
  
for(rit = arr.rbegin(); rit != arr.rend(); rit++)  
    cout<< *rit << " ";         ← 3 2 1
```

```
list<char> lis = {'a', 'b', 'c'};

list<char>::const_iterator cit;

for(cit = lis.cbegin(); cit != lis.cend(); cit++)
    cout << *cit << " ";           ← a b c

list<char>::iterator pit;

pit = lis.begin();
*pit = 'z';

for(pit = lis.begin(); pit != lis.cend(); pit++)
    cout << *pit << " ";           ← z b c
```

Algorytmy

Biblioteka STL definiuje też algorytmy, czyli odpowiednie wzorce funkcji, które mają wykonać pewne abstrakcyjne zadania na określonym kontenerze.

Przykładowym algorytmem jest `for_each`, który ma wywołać podany **funktor** na określonym zakresie elementów.

Innymi przykładami algorytmów są `reverse`, który odwraca kolejność elementów w kontenerze czy `find`, który wyszukuje określoną wartość.

Niektóre algorytmy wymagają iteratorów o **dostępie swobodnym**, a inne przeglądają sekwencje po kolei, więc wystarcza im **iterator jednokierunkowy**.

Zakresy i parametry algorytmów

Algorytmy pracują na kolekcjach i na tablicach.

Argumentami algorytmów STL są zakresy (iteratory w kolekcjach albo wskaźniki w tablicach).

Po stronie funkcji wywołującej leży obowiązek zapewnienia poprawności zakresów – oznacza to, że początek musi odnosić się do wcześniejszego lub tego samego elementu tego samego kontenera co koniec.

Algorytmy działają w trybie nadpisywania, a nie wstawiania – funkcja wywołująca musi więc zapewnić, aby zakresy docelowe posiadały odpowiedni rozmiar.

Niektóre algorytmy umożliwiają przekazanie operacji zdefiniowanych przez użytkownika, które są następnie przez nie wewnętrznie wywoływane.

Operacje te to funktory – mogą być zwykłymi funkcjami lub obiektami funkcyjnymi lub lambdaami.

Funktory służyć mogą do realizacji następujących zadań:

- predykat jednoargumentowy jako kryterium wyszukiwania lub wybierania elementów,
- predykat dwuargumentowy jako kryterium sortowania czy wyszukiwania w uporządkowanym zbiorze,
- funktor aplikowany do wszystkich elementów z podanego zakresu,
- funktor dla algorytmów numerycznych.

Klasyfikacja algorytmów

Algorytmy dzielą się na **niemodyfikujące** (tylko czytające dane) i **modyfikujące**. Przeznaczenie algorytmu można wywnioskować po jego nazwie.

Przyrostek/sufiks `_if` używany jest wtedy, gdy istnieją dwie postacie pewnego algorytmu posiadające tę samą liczbę parametrów, lecz jedna wymaga podania wartości (wersja bez przyrostka) a druga funkcji lub obiektu funkcyjnego (wersja z przyrostkiem).

Algorytm `find()` na przykład szuka elementu o określonej wartości, podczas gdy algorytm `find_if()` szuka elementu spełniającego podane kryterium.

Przyrostek/sufiks `_copy` wskazuje, że elementy podlegają nie tylko manipulacji, lecz również kopiowaniu do zakresu docelowego.

Algorytm `reverse()` na przykład odwraca kolejność elementów wewnątrz danego zakresu, podczas gdy algorytm `reverse_copy()` kopiuje elementy w odwrotnej kolejności do innego zakresu.

Algorytmy niemodyfikujące

Algorytmy niemodyfikujące nie zmieniają ani kolejności, ani wartości przetwarzanych elementów.

Algorytmy niemodyfikujące współpracują z iteratorami wejściowymi i postępującymi, można je więc wywołać dla wszystkich kontenerów standardowych.

Algorytm `for_each`

- Algorytm `for_each( )` wywołuje wobec każdego elementu operację podaną przez funkcję wywołującą.
- Algorytm `for_each( )` zwraca obiekt funkcyjny stosowany do elementów kolekcji.

```
void echo(short int n)
{
    cout << n << endl;
}

vector<short int> vec = {1, 3, 7, 9, 11, 13, 33, 44, 69};

for_each(vec.begin(), vec.end(), echo);
```

```
vector<short int> vec = {1, 3, 7, 9, 11, 13, 33, 44, 69};

auto print = [](const short int &n) {cout << n << " ";};

for_each(vec.begin(), vec.end(), print); ← 1 3 7 9 11 13 33 44 69
```

```
vector<short int> vec = {1, 3, 7, 9, 11, 13, 33, 44, 69};

int sum = 0;

for_each(vec.begin(), vec.end(), [&sum](const int &n) {sum += n;});

cout << "sum = " << sum << endl; ← sum = 190
```

Algorytmy wyszukiujące

- Funkcja `find( )` znajduje pierwsze wystąpienie zadanej wartości.
- Funkcja `find_end( )` znajduje ostatnie wystąpienie zadanego ciągu wartości.
- Funkcja `search( )` znajduje pierwsze wystąpienie zadanego ciągu wartości.
- Funkcja `min_element( )` znajduje element o najmniejszej wartości.
- Funkcja `max_element( )` znajduje element o największej wartości.

```
const int maxN = 10;
int arr[maxN] = {1, 3, 7, -9, 11, 13, -33, 44, 69};

cout << "min = " << *min_element(arr, arr + maxN) << endl;   ← min = -33
cout << "max = " << *max_element(arr, arr + maxN) << endl;   ← max = 69
```

```
const int maxN = 10;
int arr[maxN] = {1, 3, 7, -9, 11, 13, -33, 44, 69};

int n = 3;

auto res = find(begin(arr), end(arr), n);

if(res != end(arr))

    cout << "arr contains: " << n << endl;                     ← arr contains: 3

else

    cout << "does not contains: " << n << endl;
```

Algorytmy sprawdzające

- Funkcja `count( )` zlicza wystąpienia zadanej wartości w określonym zakresie.
- Funkcja `count_if( )` zlicza elementy spełniające zadany predykat.
- Funkcja `equal( )` sprawdza, czy wartości z podanych zakresów są sobie równe.
- Funkcja `mismatch( )` znajduje pierwsze wystąpienie różnicy w podanych ciągach wartości (wynikiem jest para iteratorów).
- Funkcja `is_permutation( )` sprawdza, czy zakres jest permutacją innego zakresu.
- Funkcja `is_sorted( )` sprawdza, czy zakres jest posortowany.


```
int arrA[] = {1, 2, 3};  
int arrB[] = {1, 3, 2};  
int arrC[] = {1, 2, 7};  
  
cout << "arrB is a perm. of arrA: " << boolalpha;  
cout << is_permutation(arrA, arrA + 3, arrB) << endl; ← arrB is a perm. of arrA: true  
  
cout << "arrC is a perm. of arrA: " << boolalpha;  
cout << is_permutation(arrA, arrA + 3, arrC) << endl; ← arrC is a perm. of arrA: false
```

Algorytmy modyfikujące

Algorytmy modyfikujące zmieniają wartość elementów. Mogą one bezpośrednio modyfikować elementy z danego zakresu lub modyfikować je podczas kopiowania do innego zakresu.

- Funkcja `fill()` zastępuje każdy element z zadanego zakresu podaną wartością.
- Funkcja `replace()` zastępuje elementy o określonej wartości z zadanego zakresu inną wartością.
- Funkcja `generate()` zastępuje każdy element z zadanego zakresu wartością wygenerowaną przez podaną funkcję bezargumentową.
- Funkcja `merge()` scala dwa zakresy.
- Funkcja `swap_ranges()` zamienia miejscami elementy z dwóch zakresów.
- Algorytm `for_each()` dopuszcza operację modyfikującą swój argument, argument ten musi być przekazywany przez referencję.

```

void power2(int &n)
{
    n = n * n;
}

vector<int> els2 = {1, 2, 3};

for_each(els2.begin(), els2.end(), power2);

for(const auto &e: els2)
    cout << e << " ";           ← 1 4 9

```

```

int power3(int n)
{
    return n * n * n;
}

vector<int> els3 = {1, 2, 3};

transform(els3.begin(), els3.end(), els3.begin(), power3);

for(const auto &e: els3)
    cout << e << " ";           ← 1 8 27

```

Algorytmy usuwające

Algorytmy usuwające są specjalną postacią algorytmów modyfikujących. Mogą usuwać elementy albo z pojedynczego zakresu, albo przy jednoczesnym kopiowaniu do innego zakresu.

Tak jak w przypadku algorytmów modyfikujących, jako kontenera docelowego nie można użyć kontenera asocjacyjnego ani nieuporządkowanego.

- Funkcja `remove( )` usuwa elementy o podanej wartości.
- Funkcja `remove_if( )` usuwa elementy spełniające zadany predykat.
- Funkcja `unique( )` usuwa elementy powtarzające się (sąsiednie).

Algorytmy operują na zakresach wskazanych iteratorami, a nie na kontenerach – **nie mogą zmienić rozmiaru kontenera**. *Usunięcie* polega na przesunięciu pozostawianych elementów na początek zakresu. Algorytm zwraca iterator wskazujący nowy, **logiczny koniec** zakresu, elementy za nim mają nieokreślone wartości.

```
vector<int> v = {1, 2, 3, 2, 4, 2};

auto newEnd = remove(v.begin(), v.end(), 2); ← v = 1, 3, 4, ?, ?, ? (rozmiar 6)

v.erase(newEnd, v.end()); ← v = 1, 3, 4 (rozmiar 3)
```

Aby faktycznie usunąć elementy z kontenera, wynik algorytmu przekazuje się metodzie `erase()` kontenera (*erase-remove idiom*):

```
v.erase(remove(v.begin(), v.end(), 2), v.end());

v.erase(remove_if(v.begin(), v.end(), [](int n) {return n % 2 == 0;}), v.end());

v.erase(unique(v.begin(), v.end(), v.end()); ← 1, 1, 2, 2, 2, 3, 1 -> 1, 2, 3, 1
```

Od C++20 dostępne są funkcje `erase()` i `erase_if()`, wykonujące obie operacje naraz:

```
erase(v, 2);
erase_if(v, [](int n) {return n % 2 == 0;});
```

Algorytmy mutujące

Algorytmy mutujące to algorytmy, które zmieniają kolejność elementów (a nie ich wartości) poprzez operacje przypisania i zamiany ich wartości.

- Funkcja `reverse()` odwraca kolejność elementów.
- Funkcja `rotate()` przesuwa cyklicznie elementy.
- Funkcja `shuffle()` losowo zmienia kolejność elementów, korzystając z podanego generatora liczb losowych (nagłówek `random`); starsza funkcja `random_shuffle()` została usunięta w C++17.

```
shuffle(v.begin(), v.end(), mt19937(random_device{}()));
```

- Funkcja `partition()` dzieli zakres na elementy spełniające predykat (na początku kolekcji) i te niespełniające (na końcu kolekcji).

Algorytmy sortujące

Algorytmy sortujące są specjalnym rodzajem algorytmu mutującego, ponieważ także zmieniają kolejność elementów.

Sortowanie jest jednak bardziej skomplikowane niż proste operacje mutujące i zabiera zwykle więcej czasu.

- Funkcja `sort()` sortuje elementy.
- Funkcja `stable_sort()` sortuje elementy w sposób stabilny.

Algorytmy do posortowanych danych

Algorytmy przeznaczone dla zakresów posortowanych wymagają, aby zakresy, na których one operują, były posortowane zgodnie z ich kryterium sortowania.

- Funkcja `binary_search()` sprawdza, czy dany zakres zawiera określony element.