Metody i języki programowania I

dr inż. Grzegorz Szwarc
mgr inż. Andrzej Grosser

andrzej.grosser@icis.pcz.pl

Instytut Informatyki Teoretycznej i Stosowanej

Ramowy program zajęć

- Zasięg deklaracji i czas trwania obiektów
- Przeciążenie funkcji
- Obsługa sytuacji wyjątkowych
- Klasa i obiekt klasy
- Przeciążenie operatorów i metod klas
- Dziedziczenie i wielodziedziczenie
- Polimorfizm i funkcje wirtualne
- Biblioteka wejścia – wyjścia

Wymagania podstawowe

- Stałe i zmienne (p-wartość, l-wartość)
- Typy danych: atomowe, wyliczeniowy, tablicowy, strukturalny
- Modyfikatory `static, volatile, register`
- Operatory (priorytety, łączność) i wyrażenia
- Instrukcje: proste, złożone, warunkowe, iteracyjne
- Funkcje:
  - ▷ Argumenty i wartości zwracane
  - ▷ Modyfikator `inline`
  - ▷ Funkcja `main()`
  - ▷ Rekurencja
- Typ referencyjny i wskaźnikowy
- Deklaracje i definicje obiektów i funkcji

Wymagania podstawowe

- Obsługa wejścia–wyjścia:
  - ▷ Standardowe strumienie wejścia-wyjścia
  - ▷ Manipulatory strumieni wejścia-wyjścia
  - ▷ Obsługa strumieni plikowych
- Operacje na wskaźnikach
- Dynamiczny przydział pamięci dla typów atomowych, strukturalnych oraz ich tablic
- Wskaźniki do funkcji
- Makrodefinicje preprocesora `#define, #include, #ifdef`

Literatura

- Literatura podstawowa
  - ▷ *Podstawy języka C++*, S. B. Lippman, J. Lajoie, Wydawnictwa Naukowo–Techniczne, Warszawa 2003
  - ▷ *Język C++*, B. Stroustrup, Wydawnictwa Naukowo–Techniczne, Warszawa 2004
- Literatura uzupełniająca
  - ▷ *Thinking in C++. Edycja polska*, B. Eckel, Wydawnictwo Helion, Warszawa
  - ▷ *Model obiektu C++*, S. B. Lippman, Wydawnictwa Naukowo–Techniczne, Warszawa
  - ▷ *C++ bez cholesterolu*, http://www.kis.p.lodz.pl/~sektor/cbx/
  - ▷ *Projektowanie i rozwój języka C++*, B. Stroustrup, Wydawnictwa Naukowo–Techniczne, Warszawa

Zasięg deklaracji

- Zasięg deklaracji – kontekst pozwalający rozróżniać rozmaite znaczenia nazwy (jednoznaczność symboli)
  - ▷ zasięg lokalny
  - ▷ zasięg przestrzeni nazw
  - ▷ zasięg klasy
- Zasięg lokalny – wyznacza go każdy blok instrukcji, a tym samym:
  - ▷ każda funkcja
  - ▷ każda instrukcja złożona

Zasięg deklaracji

```
int oblicz(double);
long i = 0;
float eps = 0.5;

int funkcja(const double& eps)
{
  for (int i = 0; i < 10; i++)
    { /*...*/ }
  int i = oblicz(eps);
  return i;
}
```

Zasięg deklaracji

- Globalna przestrzeń nazw – domyślnie każdy obiekt, funkcja i typ jest tworem globalnym; wymaga się by twory globalne miały jednoznaczną/unikalną nazwę
- Przestrzeń nazw deklarowana przez użytkownika – twory deklarowane wewnątrz to składowe przestrzeni nazw; nazwa tworu musi być jednoznaczna w tej przestrzeni nazw; nazwa tworu w przestrzeni nazw jest automatycznie kwalifikowana nazwą przestrzeni; przestrzeń nazw reprezentuje odrębny zasięg (zasięg przestrzeni nazw)

Zasięg deklaracji

```
// ---- pr1.h ----
namespace przyklad {
  double oblicz(double);
  extern const double eps;
}
```

```
// ---- pr1.cpp ----
#include "pr1.h"
namespace przyklad {
  const double eps = 0.34;

  double oblicz(double v)
  { return 2*v+cos(v); }
}
```

```
// ---- testowanie.cpp ----
#include "pr1.h"
int testowanie()
{
  double d = przyklad::oblicz(przyklad::eps);
  // ...
}
```

Zasięg deklaracji

```
// ---- pr1.h ----
namespace przyklad
{
  double oblicz(double);
  extern const double eps;
}
// ---- testowanie.cpp ----
#include "pr1.h"
double eps = 3.14;
double oblicz(double);

void testowanie()
{
  double eps = 2.71;
  double d = oblicz(eps);
  d = przyklad::oblicz(przyklad::eps);
  d = oblicz(::eps);
}
```

Zasięg deklaracji

- Deklaracja używania – odwoływanie się do składowej przestrzeni nazw za pomocą niekwalifikowanej nazwy; nazwa zasłania taką samą nazwę deklarowaną w zasięgu zewnętrznym; nazwa może być zasłaniana przez nazwę z zasięgu zagnieżdżonego w jej zasięgu

```
// ---- testowanie.cpp ----
#include "pr1.h"
double eps = 3.14;
double oblicz(double);
void testowanie()
{
  using przyklad::oblicz;
  double d = oblicz(eps);
  d = ::oblicz(::eps);
  double eps;
  using przyklad::eps; // błąd
  // ...
}
```

```
// ---- pr1.h ----
namespace przyklad {
  double oblicz(double);
  extern const double eps;
}
```

Zasięg deklaracji

- Dyrektywa używania – odwoływanie się do wszystkich składowych przestrzeni nazw za pomocą niekwalifikowanych nazw; sprawia, że nazwy składowych są windowane do zasięgu zawierającego definicję przestrzeni nazw

```
// ---- testowanie.cpp ----
#include "pr1.h"
double oblicz(double);

void testowanie()
{
  using namespace przyklad;
  double d = oblicz(eps); // błąd
  d = ::oblicz(eps);
  d = przyklad::oblicz(eps);
}
```

```
// ---- pr1.h ----
namespace przyklad {
  double oblicz(double);
  extern const double eps;
}
```

Czas trwania obiektów

- Obiekt globalny – istnieje przez cały czas wykonania programu; zasięg ogranicza się do bieżącego pliku źródłowego; odpowiednimi deklaracjami (`extern`) zasięg można rozszerzyć na inne pliki źródłowe; domyślnie inicjowany wartością zera
- Obiekt automatyczny – obiekt lokalny; przydział pamięci następuje automatycznie w chwili wywołania funkcji; czas trwania obiektu kończy się wraz z zakończeniem wykonania funkcji; zasięg ograniczony jest do bieżącego kontekstu; należy uważać na wskaźniki i referencje do obiektów lokalnych; obiekt domyślnie nie jest inicjowany
- Statyczny obiekt lokalny – deklaracja z modyfikatorem `static`; istnieje przez cały czas wykonania programu; wartość takiego obiektu przetrwa między kolejnymi wywołaniami funkcji; zasięg ograniczony jest do bieżącego kontekstu; domyślnie inicjowany wartością zera
- Obiekt z czasem trwania określanym przez programistę – obiekt z pamięcią przydzielaną dynamicznie; obiekt bez nazwy; identyfikowany pośrednio przez wskaźnik; (zawieszony wskaźnik – wskazujący na nie obsługiwany obszar pamięci (*wskaźnik zwisający*)); wyciek pamięci – obszar pamięci przydzielany dynamicznie na który nie wskazuje żaden wskaźnik

Czas trwania obiektów

- Dynamiczne zarządzanie pamięcią – operatory `new` i `delete`; w przypadku niewystarczających zasobów podczas przydziału pamięci zgłaszana jest sytuacja wyjątkowa `bad_alloc`

```
int* wsk = new int; // new (nothrow) int;
*wsk = 123;
delete wsk;

const int* cwsk = new const int(100);

if (wsk != 0) // czy konieczne?
  delete wsk;

int* wskt = new int[10];
delete [] wskt;
int* wsk1 = new int[obliczony_rozmiar()];
int (*wsk2)[124] = new int[obliczony_rozmiar()][124];
const int *cwskt = new const int[100]; //błąd
```

Pseudonimy przestrzeni nazw

- Długiej nazwie przestrzeni można nadać pseudonim
- Pseudonim można nadać nawet zagnieżdżonym przestrzeniom nazw
- Każda przestrzeń nazw może mieć wiele synonimów (pseudonimów)

```
namespace BardzoDlugaNazwa{
   namespace JeszczeDluzszaNazwaPrzestrzeniNazw{}
}
namespace BDN = BardzoDlugaNazwa;
namespace
   JDN = BardzoDlugaNazwa::JeszczeDluzszaNazwaPrzestrzeniNazw;
```

Zagnieżdżone przestrzenie nazw

- Przestrzenie nazw można zagnieżdżać, np. w celu lepszego wyodrębnienia jednostek logicznych programu

```
namespace Fabryka{
   extern int x;
   void fun(int x);
   namespace Montownia{
      extern double x;
      void fun(int x);
   }
   namespace Lakiernia{
      extern double x;
   }
}
//...
Fabryka::Montownia::fun(x1);
Fabryka::fun(x1);
//...
```

Nienazwane przestrzenie nazw

```
// ---- kod1.cpp ----
static void drukujWynik(int a, int b)
{std::cout << a + b << std::endl;}
//...
drukujWynik(a1,b1);
//...

// ---- kod2.cpp ----
namespace {
void drukujWynik(int a, int b)
{std::cout << a + b << std::endl;}
}
//...
drukujWynik(a2,b2);
//...
```

- Nazwy zadeklarowane w nienazwanej przestrzeni nazw są lokalne w obrębie jednego pliku
- Odnosząc się do składowych nienazwanych przestrzeni nazw nie trzeba używać operatora zasięgu

Wzorzec klasy `auto_ptr`

- Wzorzec klasy `auto_ptr` pozwala na zautomatyzowanie zarządzania obiektami umieszczonymi w pamięci dynamicznej, programista nie musi się sam troszczyć o zwalnianie zasobów przydzielonych operatorem `new`
- Obiekt `auto_ptr` jest właścicielem obiektu, na który wskazuje; dzięki czemu po zniszczeniu tegoż obiektu klasy `auto_ptr` automatycznie niszczony jest obiekt, na który wskazuje

Wzorzec klasy `auto_ptr`

- Do obiektów klasy `auto_ptr` można przypisywać tylko i wyłącznie wskaźniki wskazujące na pojedyńcze obiekty umieszczone w pamięci dynamicznej - nie ma wersji dla tablic
- Należy pamiętać żeby do obiektu klasy `auto_ptr` przypisywać tylko i wyłącznie adresy pamięci przydzielonej operatorem `new` - brak dbałości o to grozi niezdefiniowanym zachowaniem programu
- Trzeba uważać żeby nie powstały dwa obiekty klasy `auto_ptr` mające prawo własności do tego samego obiektu przypisywanego dynamicznie

Wzorzec klasy auto_ptr

```
#include <memory>

double *wsk = new double(10.0);
std::auto_ptr<double> a(wsk);
std::auto_ptr<double> b(new double);
std::auto_ptr<double> c;

c = b;//nie można się już odwoływać za pomocą b do obiektu
std::cout << *c << std::endl;
c.get();//przekazuje adres obiektu
c.release();//przekazuje adres obiektu i znosi prawo własności
c.reset(new double(3.0));//nadaje c adres nowo utworzonego obiektu,
                         //poprzednia wartość jest usuwana
```

Umieszczający operator new

- Trzecia postać wyrażenia new pozwala na utworzenie obiektu w już przydzielonej dynamicznie pamięci
- Powinno się go używać tylko do umieszczenia obiektów w określonym adresie pamięci
- Programista bierze całkowitą odpowiedzialność za to, że wskaźnik będący parametrem operatora new wskazuje na obszar pamięci odpowiednio wyrównany i wystarczający na umieszczenie nowego obiektu!

Umieszczający operator new

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

struct X
{
 int a; char b;
 X()
 { a = 1; b = 1; }
};

int main()
{
   char * buf = new char[sizeof(X) * 4];
   X *wx = new (buf) X[4];
   std::cout << wx[0].a << std::endl;
   delete[] buf;
}
```

Funkcje przeciążone

- Przeciążenie funkcji – nadanie takiej samej nazwy funkcjom wykonującym takie same (podobne) operacje na różnych typach argumentów; jednoznaczność listy argumentów formalnych ze względu na liczbę argumentów i ich typy

```
// funkcje przeciążone
int maks(int a, int b);
int maks(int tab[]);
int maks(int, int, int);

// błędna powtórna deklaracja
long maks(int, int);

// deklaracje tej samej funkcji
int maks(int a, int b);
int maks(int, int);
int maks(int, int = 100);
int maks(const int, int);
```

```
// typedef - to nie nowy typ
typedef int liczba;
int maks(int, liczba);

// deklaracje różnych funkcji
// przeciążenie
int maks(int, int);
int maks(int*, int);
int maks(const int*, int);
int maks(int&, int);
int maks(const int&, int);
```

Funkcje przeciążone

- Rozstrzyganie przeciążenia – wybór jednej funkcji ze zbioru funkcji przeciążonych na podstawie argumentów określonych w wywołaniu funkcji; przebiega w trzech etapach
  - ▷ zidentyfikowanie zbioru funkcji przeciążonych, który można rozważać dla danego wywołania funkcji oraz zidentyfikowanie właściwości listy argumentów aktualnych w wywołaniu funkcji – funkcje kandydujące
  - ▷ wybranie funkcji ze zbioru funkcji przeciążonych, które ze względu na liczbę i typy argumentów formalnych, można by wywołać z argumentami określonymi w wywołaniu funkcji – funkcje żywotne; klasyfikacja zgodności typów
    - ⋆ ścisła zgodność
    - ⋆ zgodność po przekształceniu typu
    - ⋆ niezgodność typu

Funkcje przeciążone

- wybranie funkcji, która jest ściśle zgodna z wywołaniem funkcji – funkcja najżywotniejsza; klasyfikacja przekształceń typów
  - przekształcenia typów zastosowane do argumentów nie są gorsze niż przekształcenia niezbędne do tego, by można było wywołać dowolną inną funkcję żywotną
  - przekształcenia typów zastosowane do pewnych argumentów są lepsze niż przekształcenia, które trzeba by było zastosować do tych samych argumentów przy wywołaniu pozostałych funkcji żywotnych

Funkcje przeciążone

- Ścisła zgodność – typ argumentu formalnego taki sam jak argumentu aktualnego; dopuszczalne również przekształcenia
  - ▷ przekształcenie l–wartości w p–wartość

```
int oblicz(int);
void drukuj(int&);
int wrt, wyn;
wrt = 5; // wrt - l-war.; 5 - p-war.
wyn = oblicz(wrt);
// wyn - l-wrt.;
// obiekt tymczasowy przechowujący wartość
// zwróconą przez oblicz() - p-war.
drukuj(wyn); // brak przek. l-war. na p-war.
```

Funkcje przeciążone

- przekształcenie tablicy we wskaźnik

```
int tab[3];
void drukuj(int*);
drukuj(tab);
```

- przekształcenie funkcji we wskaźnik

```
void zapisz(void (*drk)(int*));
zapisz(drukuj);
```

- przekształcenia kwalifikujące (modyfikatory)

```
int a = 5, *wsk = &a;
void fun1(const int*);
void fun2(const int);
void fun3(int* const);
fun1(wsk);
fun2(a); // brak przekształcenia
fun3(wsk); // brak przekształcenia
```

Funkcje przeciążone

- Zgodność po przekształceniu typu; trzy kategorie
  - ▷ awansowanie
    - ⋆ argument typu: `char, unsigned char, short` awansuje do typu `int`; jeżeli na danej maszynie rozmiar typu `int` jest większy niż rozmiar typu `short` to argument typu `unsigned short` awansuje do typu `int` – w przeciwnym razie awansuje do typu `unsigned int`

      ```
      void fun(int);
      fun('a');
      ```

    - ⋆ argument typu `float` awansuje do typu `double`

      ```
      void fun(double);
      float f = 1.0;
      fun(f);
      ```

Funkcje przeciążone

- argument typu wyliczeniowego awansuje do pierwszego z następujących typów, które mogą reprezentować wszystkie wartości stałych należących do danego wyliczenia: `int, unsigned int, long` lub `unsigned long`

```
enum stan {wygrana, porazka};
void fun(int);
void fun(long);
fun(wygrana); // awans do int
```

- argument typu `bool` awansuje do typu `int`

```
bool f = true;
fun(f); // awans do int
```

Funkcje przeciążone

- przekształcenie standardowe; wszystkie równorzędne (pokrewieństwo typów nie ma wpływu); pięć rodzajów
  - przekształcenia typów całkowitych: przekształcenia dowolnego typu całkowitego lub wyliczeniowego w dowolny inny typ całkowity (z wyłączeniem awansowań) – brak ostrzeżeń o konwersji

```
void fun(char);
int f = 100;
fun(f); // przek. int w char -- niebezpieczne
```

  - przekształcenia typów zmiennopozycyjnych: przekształcenia dowolnego typu zmiennopozycyjnego w dowolny inny typ zmiennopozycyjny (z wyłączeniem awansowań) – brak ostrzeżeń o konwersji
  - przekształcenia zmiennopozycyjno–całkowite: przekształcenia dowolnego typu zmiennopozycyjnego w dowolny typ całkowity lub przekształcenie dowolnego typu całkowitego w typ zmiennopozycyjny

```
void fun(int);
float f = 100;
fun(f); // przek. float w int -- niebezpieczne
```

Funkcje przeciążone

- ⋆ przekształcenia typów wskaźnikowych: przekształcenia wartości całkowitej zero w typ wskaźnikowy i przekształcenia wskaźnika dowolnego typu w typ `void*`

```
void fun(void*);
int f = 10, *wf = &f;
fun(wf); // przek. int* w void*
fun(0); // przek. literału w void*
```

- ⋆ przekształcenia typów logicznych: przekształcenia dowolnego typu całkowitego, wyliczeniowego lub wskaźnikowego w typ `bool`
- ▷ przekształcenia zdefiniowane przez użytkownika

- Referencje jako parametry formalne funkcji przeciążonych
  - ▷ argument aktualny jest odpowiednim inicjatorem dla argumentu formalnego; ścisła zgodność

```
void zamien(int&, int&);
int a = 2, b = 3;
zamien(a, b);
```

Funkcje przeciążone

- argument aktualny nie może inicjować argumentu formalnego; brak ścisłej zgodności

```
void oblicz(double&);
int f = 2;
oblicz(f); // błąd - obiekt tymczasowy
// rozwiązaniem jest modyfikator const
void oblicz(const double&);

void oblicz(int&);
void oblicz(int);
int f = 2;
int &rf = f;
oblicz(f); // błąd - niejednoznaczne
oblicz(rf); // błąd - niejednoznaczne
oblicz(86); // wywołanie oblicz(int)
```

Funkcje przeciążone

- Wskaźnik do funkcji przeciążonej – ścisła zgodność z typem wskaźnika

```
void drukuj(const string&);
void drukuj(double);

void (*prn0)(double) = drukuj;
void (*prn1)(int) = drukuj; // błąd
double (*prn2)(double) = drukuj; // błąd
```

Przeciążenie a zasięg

- Funkcja zdeklarowana lokalnie nie przeciąża funkcji zdeklarowanych w zasięgu globalnym tylko je przysłania

```
void drukuj(const string&);
void drukuj(double);

void test(int i)
{
  void drukuj(int);// lokalna deklaracja
  drukuj("wartość: "); // błąd
  drukuj(i);
}
```

- Funkcje zdeklarowane w różnych przestrzeniach nazw nie przeciążają się – inny zasięg

Przeciążenie a zasięg

- Funkcja wprowadzona do bieżącego zasięgu przez definicję używania przeciąża inne deklaracje funkcji

```
namespace drk {
  void drukuj(const string&);
  void drukuj(double);
}

void drukuj(int);
using namespace drk;

void test(int i)
{
  drukuj("wartość: ");
  drukuj(i);
}
```

Przeciążenie a zasięg

- Funkcja wprowadzona do bieżącego zasięgu przez dyrektywę używania przeciąża inne deklaracje funkcji

```
namespace drk {
  void drukuj(const string&);
  void drukuj(double);
}
```

```
void drukuj(int);
using drk::drukuj;

void test(int i)
{
  drukuj("wartość: ");
  drukuj(i);
}

void drukuj(double); // błąd
```

```
// --------===##===--------
void drukuj(double);

void test(int i)
{
  //przysłonięcie
  using drk::drukuj;
  drukuj("wartość: ");
  drukuj(i);
}
```

Obsługa sytuacji wyjątkowych

- Obsługa sytuacji wyjątkowych – mechanizm pozwalający na komunikację dwóch fragmentów programu w celu obsłużenia “nienormalnego” zdarzenia (sytuacji wyjątkowej); mechanizm (w C++) niewznawialny; po zgłoszeniu wyjątku i jego ewentualnym obsłużeniu, sterowanie nie może być przekazane do miejsca zgłoszenia wyjątku; zgłaszanie wyjątku – słowo kluczowe `throw` poprzedzające wyrażenie, którego typem jest zgłaszany wyjątek; obsługa wyjątków – blok `try` (zawierający instrukcje zgłaszające wyjątki) oraz jednej lub kilku klauzul `catch` z argumentem typu wyjątku i blokiem instrukcji jego obsługi

```
double odw(int x)
{
  return 1.0/x;
}

for (int i = -2; i < 2; ++i)
  cout << odw(i);    // -0.5 -1 inf 1
```

Obsługa sytuacji wyjątkowych

```
double odw(int x)
{
  if (x == 0) throw int(2);
  return 1.0/x;
}

double ptl(int a, int b)
{
  double s = 0;
  try {
    for (int i = a; i < b; ++i)
      s += odw(i);
  }
  catch (int w) {
    cerr << "błąd␣nr␣" << w << '\n' ;
    cerr << "wartość␣s:" << s << '\n';
  }
  return s;
}
```

```
// --------===##===--------

cout << ptl(-2, 2);
        // błąd nr 2
        // wartość s:-1.5
        // -1.5
```

Obsługa sytuacji wyjątkowych

- blok `try` funkcji – powiązanie grupy klauzul `catch` z treścią funkcji

```
double ptl(int a, int b)
try {
  double s = 0;
  for (int i = a; i < b; ++i)
    s += odw(i);
  return s;
}
catch (int w) {
  cerr << "błąd␣nr␣" << w ;
}
```

Obsługa sytuacji wyjątkowych

- Zwijanie stosu – proces, w którym wychodzi się z instrukcji złożonych i definicji funkcji w poszukiwaniu klauzuli `catch` przeznaczonej do obsługi zgłoszonego wyjątku; gwarancja wywołania destruktorów obiektów lokalnych podczas opuszczania ich kontekstu wymuszonego zgłoszeniem sytuacji wyjątkowej; jeśli nie uda się znaleźć odpowiedniej klauzuli zostanie wywołana funkcja biblioteczna `terminate()`

```
double odw(int x)
{
  if (x == 0) throw int(2);
  if (x > 1000) throw  string("zły arg.");
  return 1.0/x;
}

try {
  cout << ptl(998, 1002);  // zły arg.
}
catch (string e) { cerr << e << '\n'; }
```

Obsługa sytuacji wyjątkowych

- Wychwytywanie wszystkich wyjątków – klauzule `catch` przeglądane są w kolejności występowania w programie; po znalezieniu właściwej pozostałe nie są sprawdzane; klauzula wychwytująca wszystkie wyjątki powinna być zapisywana jako ostatnia

```
enum Err {niezle, zle, bardzozle};

double odw(int x)
{
  if (x == 0) throw int(2);
  if (x > 1000) throw string("zły␣arg.");
  if (x < -1000) throw Err(bardzozle);
  return 1.0/x;
}

try { cout << ptl(-1002, 998); }    // nieznany błąd
catch (string e) { cerr << e << '\n'; }
catch (...) { cerr << "nieznany␣błąd\n"; }
```

Obsługa sytuacji wyjątkowych

- Ponowne zgłoszenie wyjątku – klauzula `catch` może przekazać wyjątek do innej klauzuli znajdującej się wyżej na liście wywołań funkcji

```
double ptl(int a, int b)
{
  double s = 0;
  try {
    for (int i = a; i < b; ++i)
      s += odw(i);
  }
  catch (int& w) {
    cerr << "błąd␣nr␣"
         << w++ << '\n' ;
    cerr << "wartość␣s:"
         << s << '\n';
    throw;
  }
  return s;
}
```

```
try { cout << ptl(-2, 2); }
catch (int w) {
  cerr << "ponowny␣błąd␣nr␣"
       << w << '\n';
}
catch (string e) {
  cerr << e << '\n';
}
catch (...) {
  cerr << "nieznany␣błąd\n";
}

// --------===##===--------
// błąd nr 2
// wartość s:-1.5
// ponowny błąd nr 3
```

Obsługa sytuacji wyjątkowych

- Specyfikacje wyjątków – dołączenie do deklaracji funkcji listy wyjątków które może zgłosić; gwarancja nie zgłaszania przez funkcję wyjątków z poza listy; listę zgłaszanych wyjątków umieszczoną w parze nawiasów okrągłych umieszcza się za listą parametrów formalnych funkcji poprzedzając ją słowem kluczowym `throw`; pusta lista wyjątków – gwarancja, że funkcja nie zgłosi żadnego wyjątku; jeśli funkcja zgłosi wyjątek z poza listy – wywołanie funkcji `unexpected()` standardowo wywołującej `terminate()`

```
double odw(int x) throw (int, string, Err)
{
  if (x == 0) throw int(2);
  if (x > 1000) throw string("zły␣arg.");
  if (x < -1000) throw Err(bardzozle);
  return 1.0/x;
}
```

Obsługa sytuacji wyjątkowych

```
double ptl(int a, int b) throw (string, Err)
{
  double s = 0;
  try {
    for (int i = a; i < b; ++i)
      s += odw(i);
  }
  catch (int w) {
    cerr << "błąd␣nr␣" << w << '\n' ;
  }
  return s;
}

try { cout << ptl(-2, 2); }
catch (string e) { cerr << e << '\n'; }
catch (...) { cerr << "nieznany␣błąd\n"; }
```

Obsługa sytuacji wyjątkowych

- Specyfikacje wyjątków i wskaźniki do funkcji – specyfikacja wyjątków dla wskaźnika do funkcji użytego jako wartość inicjująca lub jako p–wartość musi być co najmniej tak samo restrykcyjna jak specyfikacja wskaźnika inicjowanego lub występującego po lewej stronie operatora przypisania

```
double odw(int x) throw (int, string, Err)
{
  if (x == 0) throw int(2);
  if (x > 1000) throw string("zły␣arg.");
  if (x < -1000) throw Err(bardzozle);
  return 1.0/x;
}

double (*odw_w0)(int) = odw; // ok
double (*odw_w1)(int) throw(int, string, Err) = odw; // ok
double (*odw_w2)(int) throw(int, string, Err, double) = odw; // ok
double (*odw_w3)(int) throw(int, string) = odw; // błąd
double (*odw_w4)(int) throw() = odw; // błąd
```

Obsługa sytuacji wyjątkowych

- Obsługa sytuacji wyjątkowej braku wystarczających zasobów pamięci – zgłaszane przez operator new; wyjątek typu bad_alloc

```
int main()
try {
  int* w = new int[192*1024*1024];
  // ...
  delete [] w;
  return 0;
}
catch (bad_alloc) { cerr << "bleee\n"; }
```

Klasa

- Klasa – typ zdefiniowany przez użytkownika; służy do definiowania pojęć abstrakcyjnych, których nie można bezpośrednio wyrazić za pomocą typów wbudowanych
- Nagłówek klasy – nazwa klasy poprzedzona słowem kluczowym `class`
- Treść klasy – lista składowych klasy ujęta w parę nawiasów klamrowych; treść klasy stanowi oddzielny zasięg; trzy rodzaje składowych
  - ▷ deklaracje pól (zmiennych)
  - ▷ deklaracje metod (funkcji)
  - ▷ specyfikatory dostępu do pól i metod (do *inwariantów*)
- Definicja klasy – nagłówek klasy + treść klasy + ; każda definicja klasy wprowadza nowy typ danych
- Deklaracja klasy – nagłówek klasy + ;

```
class Pierwsza
{
  int i;
  double d;
};

class Druga
{
  int i;
  double d;
};

class Pierwsza ob1;
Druga ob2;
ob1 = ob2; // błąd; ob1 != ob2
```

Klasa

- Pole – dana, składowa klasy; deklarowana tak jak zmienne; pola inicjowane są za pośrednictwem konstruktora; polami mogą być obiekty typów prostych, złożonych, obiekty innych klas, wskaźniki/referencje do nich, a także wskaźniki/referencje do obiektów klasy definiowanej

```
class Punkt
{
  unsigned char kolor;
  double wspX, wspY;
};
```

- Metoda – funkcja składowa klasy; w definicji klasy można umieszczać deklaracje i definicje funkcji operujących na polach klasy

```
class Punkt
{
  unsigned char kolor;
  double wspX, wspY;
  void ustawKolor(unsigned char pKolor)
  { kolor = pKolor; }
  unsigned char podajKolor()
  { return kolor; }
  double wsp(int i);
  void wsp(double& x, double& y);
  void ustaw(double x=0.0, double y=0.0);
};

void Punkt::ustaw(double x, double y)
{
  wspX = x;
  wspY = y;
}
```

- Hermetyzacja – wyodrębnianie i ukrywanie prywatnej wewnętrznej reprezentacji i implementacji (*enkapsulacja, kapsułkowanie*); publiczna część klasy stanowi jej interfejs; możliwość dostępu do składowych klasy zależy od miejsca zadeklarowania oznaczone odpowiednim specyfikatorem dostępu
  - `public` – składowa publiczna dostępna w każdym miejscu bieżącego zasięgu oraz w odpowiednich zasięgach zagnieżdżonych
  - `private` – składowa prywatna dostępna tylko w zasięgu (treści) tej klasy; aby zapewnić hermetyzacje wszystkie pola klasy winny być zadeklarowane jako prywatne; składowe prywatne niedostępne dla klas potomnych; domyślny specyfikator dostępu
  - `protected` – składowa chroniona jest niedostępna publicznie (tak jak prywatna) natomiast dostęp (publiczny) do tej składowej mają klasy potomne

Klasa

```
class Punkt
{
private:
  unsigned char kolor;
  double wspX, wspY;
public:
  void ustawKolor(unsigned char pKolor);
  unsigned char podajKolor();
  double wsp(int i);
  void wsp(double& x, double& y);
  void ustaw(double x, double y);
};
```

- Zaprzyjaźnianie – zezwolenie pewnym funkcjom, metodom innych klas lub całym klasom na dostęp do składowych prywatnych; uprawnienie do dostępu do składowych niepublicznych; deklaracja zaprzyjaźnienia występuje we wnętrzu definicji klasy; specyfikator dostępu nie wpływa na zaprzyjaźnienie

```
class Punkt
{
private:
  unsigned char kolor;
  //...
friend
  ostream& operator<<(ostream&, const Punkt&);
};
```

- Struktura – klasa, której wszystkie składowe są domyślnie publiczne; zapis

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

równoważny jest zapisowi

```
class s { public: /*....*/ };
```

Obiekt klasy

- Obiektami klas definiowanych przez użytkownika rządzą takie same zasady jak obiektami typów wbudowanych
  - ▷ posiadają czas trwania (obiekt globalny, automatyczny, statyczny, dynamiczny)
  - ▷ obiekty tego samego typu można wzajemnie inicjować i przypisywać; domyślnie wykonywane jest kopiowanie wszystkich pól

```
Punkt a, c, b = a;
c = b;
```

  - ▷ posiadają zasięg – zależny od kontekstu użycia

```
Punkt p;
{ Punkt p = ::p; }
```

  - ▷ można deklarować wskaźniki i referencje do obiektów

```
Punkt a, *c = &a, &b = a;
c = new Punkt;
delete c;
```

Obiekt klasy

- można tworzyć tablice obiektów i wskaźników do nich

```
Punkt p[5], *wsk[4];
```

- można użyć obiektu jako argument funkcji (domyślnie przekazywany przez wartość) lub jego wskaźnika/referencji; może być wartością zwracaną przez funkcję

```
Punkt przesun(Punkt& a, const Punkt& delta);
```

- można użyć obiektu jako pola innej klasy

```
class Kwadrat
{
  Punkt wierzcholki[4];
public:
  void rysuj();
};
```

- może być obiektem typu `const` i `volatile`

```
const Kwadrat kwadrat;
```

Obiekt klasy

- Dostęp do składowych
  - operator kropkowy `.` – dostęp do składowych obiektu i referencji do obiektu

```
Punkt p;
p.ustawKolor(127);
Punkt &rp = p;
double x = rp.wsp(0);
```

  - operator strzałkowy `->` – dostęp do składowych przez wskaźnik do obiektu

```
Punkt *wp = &p;
wp->ustawKolor(255);
double x = wp->wsp(0);
double y = (*wp).wsp(1);
```

- Metody specjalne – inicjowanie obiektów (konstruktor), przypisywanie wartości, zarządzanie pamięcią, przekształcenia typu oraz niszczenie obiektów (destruktor)

Metody

- Metody z atrybutem `inline`
  - ▷ metody definiowane w definicji klasy są automatycznie kwalifikowane jako funkcje rozwijane w miejscu wywołania
  - ▷ metody definiowane poza definicją klasy wymagają jawnego użycia specyfikatora `inline`; jest to dozwolone dla deklaracji w definicji klasy, jak i w definicji metody po za definicją klasy, a także w obu tych miejscach

```
class Punkt
{
private:
  unsigned char kolor;
  double wspX, wspY;
public:
  void ustawKolor(unsigned char pKolor) { kolor = pKolor; }
  inline unsigned char podajKolor() { return kolor; }
  double wsp(int i);
  void wsp(double& x, double& y);
  void kopiuj(const Punkt& punkt);
  inline void ustaw(double x=0.0, double y=0.0);
};

inline void Punkt::ustaw(double x, double y)
{
  wspX = x;
  wspY = y;
}
```

- Metody z atrybutem `const` – dla obiektu definiowanego jako `const` można wykonać tylko metody zadeklarowane jako stałe (modyfikatorem `const`); słowo kluczowe `const` umieszcza się w deklaracji metody i w jej definicji (za listą argumentów); funkcje tego typu można przeciążyć funkcją bez modyfikatora

```
class Punkt
{
private:
  unsigned char kolor;
  double wspX, wspY;
public:
  void ustawKolor(unsigned char pKolor) { kolor = pKolor; }
  inline unsigned char podajKolor() const { return kolor; }
  double wsp(int i) const;
  void wsp(double& x, double& y);
  void kopiuj(const Punkt& punkt);
  inline void ustaw(double x=0.0, double y=0.0);
};
```

Metody

- Obiekt klasy zadeklarowany jako `const` uważa się za obiekt z atrybutem `const` jedynie od chwili zakończenia konstruowania obiektu do chwili rozpoczęcia niszczenia go
- Metoda zadeklarowana jako stała gwarantuje nie zmienianie wartości pól obiektu, natomiast może zmieniać wartości obiektów do których zadeklarowano wskaźniki

```
class Tekst
{
public:
  void zla(const string& str) const
  {
    tekst = str.c_str(); // błąd
    for (int i = 0; i < str.size(); i++)
      tekst[i] = str[i]; // OK! ale...
  }
private:
  char* tekst;
};
```

- Pola z atrybutem `mutable` – zezwolenie na modyfikowanie pola obiektu nawet wówczas gdy zmiany dotyczą obiektu z atrybutem `const`

```
class Tekst
{
public:
  void dobra(const string& str) const
  { tekst = str.c_str(); }
private:
  mutable char* tekst;
};
```

- Metody z atrybutem `volatile` – metody deklarowane jako ulotne (analogicznie jak dla modyfikatora `const`) są wywoływane dla obiektu zdefiniowanego jako ulotny

- Wskaźnik `this` – niejawnie zdefiniowana składowa każdej (niestatycznej) metody klasy przechowujące adres obiektu; wskaźnik pozwalający kojarzyć pola obiektu z jego metodami; przekształcenia kodu źródłowego do jawnego użycia wskaźnika
  - ▷ redefinicja metod klasy, tak by każda metoda otrzymała dodatkowy argument – wskaźnik `this`

```
void Punkt::ustaw(double x, double y)
{
   wspX = x;
   wspY = y;
}

// pseudo kod
void ustaw(Punkt* this, double x, double y)
{
   this->wspX = x;
   this->wspY = y;
}
```

- przebudowa wywołania metod klasy; dodanie nowego argumentu aktualnego – adres obiektu dla którego wywołuje się daną metodę

```
punkcik.ustaw(2.1, 3.4);

// pseudo kod
ustaw(&punkcik, 2.1, 3.4);
```

- typ wskaźnika `this` zależy od atrybutów metody (`const`, `volatile`); jeżeli metoda ma atrybut `const` to pierwszy niejawny argumenty formalny metody też jest typu stałego lub ulotnego;
- jawne wykorzystanie pola `this` w metodzie klasy

```
void Punkt::kopiuj(const Punkt& punkt)
{
  if (this != &punkt) {
    kolor = punkt.kolor;  wspX = punkt.wspX; wspY = punkt.wspY;
  }
}
```

- Pole statyczne – pole klasowe; tworzenie jednej wspólnej kopi pola do której mają dostęp wszystkie obiekty klasy; mimo globalnej natury pole tego typu należy do przestrzeni nazw klasy; obowiązują mechanizmy ukrywania informacji; inicjowanie na zewnątrz definicji klasy; definicja pola statycznego należy do zasięgu klasy

```
class Punkt
{
private:
  static unsigned char kolor; // dekl.
  double wspX, wspY;
  public:
  // ...
};

//wymagana definicja pola statycznego
unsigned char Punkt::kolor = 255;
```

```
class Punkt
{
private:
  static const unsigned char kolor;
  //static const unsigned char kolor = 16; // domyślna wartość
  double wspX, wspY;
public:
  // ...
};

//wymagana definicja pola statycznego
const unsigned char Punkt::kolor = 16;
//const unsigned char Punkt::kolor;
```

- Dostęp do pól statycznych – wewnątrz metod składowych klasy dostęp jest taki sam jak do innych pól (nie wymaga kwalifikowania); dla pól statycznych publicznych oraz zaprzyjaźnionych funkcji/metod/klas; dwa sposoby dostępu
  - ▷ operatory dostępu (kropkowy i strzałkowy)
  - ▷ operator zasięgu klasy; kwalifikowanie nazwy pola statycznego nazwą klasy
- Szczególne własności pola statycznego
  - ▷ typem pola statycznego może być jego klasa; dla pola niestatycznego można zadeklarować je jedynie jako wskaźnik lub referencję do obiektu tej klasy
  - ▷ pola statycznego można użyć jako wartości domyślnej argumentu metody klasy, podczas gdy nie można tego zrobić z polem niestatycznym
- Metody statyczne – dostęp do pól statycznych (niepublicznych) klasy nawet gdy nie zdefiniowano ani jednego obiektu klasy; nie może być `const` ani `volatile`; nie może zawierać odwołania do wskaźnika `this` ani do pól niestatycznych klasy

Klasa

```
class Punkt
{
private:
  static unsigned char kolor;
  double wspX, wspY;
public:
  static void Kolor(unsigned char pKolor)
  { kolor = pKolor; }
  static unsigned char Kolor();
  // ...
};

unsigned char Punkt::Kolor()
{ return kolor; }

Punkt::Kolor(200);
Punkt p;
cout << p.Kolor();
```

- Wskaźnik do składowej klasy – deklaracja wskaźnika wymaga uwzględnienia typu klasy

```
//wskaźnik do pola
double Punkt::* wpole;
//wskaźnik do metody
void (Punkt::* wmetoda)(double, double)
```

- Korzystanie ze wskaźnika do składowej – wskaźnik do składowej należy najpierw związać z obiektem lub wskaźnikiem do takiego obiektu; dostęp do składowej poprzez operatory
  - ▷ dla obiektu lub jego referencji `.*`
  - ▷ dla wskaźnika do obiektu `->*`

```
class Tablica
{
public:
  int liczba_e;
  void kopiuj(const Tablica& tab);
  //...
};

int Tablica::* w_licz = &Tablica::liczba_e;
void (Tablica::* w_kop)(const Tablica&) = &Tablica::kopiuj;

Tablica tab, *wtab;

//bezpośrednio
if (tab.liczba_e == wtab->liczba_e) wtab->kopiuj(tab);

//za pośrednictwem wskaźników do składowych
if (tab.*w_licz == wtab->*w_licz) (wtab->*w_kop)(tab);
```

- Wskaźnik do składowych statycznych – składowe statyczne są obiektami lub funkcjami globalnymi należącymi do danej klasy i do wskazywania na nie służą zwykłe wskaźniki

```
class Tablica
{
public:
  static int liczba_e;
  static void kopiuj(const Tablica& tab);
  //...
};

int* w_licz = &Tablica::liczba_e;
void (*w_kop)(const Tablica&) = Tablica::kopiuj;
```

- Zasięg klasy – definicja klasy + definicje pól statycznych klasy + definicje metod klasy

```
class Napis
{
public:
  typedef int index_t;
  typedef char char_t;
  char_t& znak(index_t elem);
  void czysc(char_t = tlo);
  static char_t ustaw_tlo(char_t znak);
private:
  char_t* napis;
  static char_t tlo;
};

Napis::char_t Napis::tlo = ustaw_tlo('#');
Napis::char_t& Napis::znak(index_t elem)
{ return napis[elem]; }
```

- Rozstrzyganie znaczenia nazw w zasięgu klasy; trzy etapy
  1. deklaracje zapisane w zasięgach lokalnych definicji metody
  2. deklaracje wszystkich składowych klasy
  3. deklaracje zapisane w zasięgu przestrzeni nazw przed definicją metody

```
int b;
class A
{
public:
  void zmien(int b)
  { b = 0; }; // ::b; A::b; this->b
private:
  int b;
};
```

- Klasy jako składowe przestrzeni nazw

```
// plik.h
namespace B {
  class A
  {
  public:
    void zmien(int pb);
  private:
    int b;
    static int c;
  };
}

//plik.cpp
#include "plik.h"
namespace B {
  void A::zmien(int pb) { b = pb; }
  int A::c = 10;
}
```

```
//plik.cpp -- inaczej
#include "plik.h"
void B::A::zmien(int pb)
{ b = pb; }
int B::A::c = 10;

// main.cpp
#include "plik.h"
int main()
{
  B::A ob;
  ob.zmien(22);
  return 0;
}
```

Unia

- Unia – specjalny rodzaj klasy, w której każde pole ma ten sam adres – obszary pamięci zajmowane przez poszczególne pola nakładają się; rozmiar unii to rozmiar jej największej składowej
- Unia może zawierać metody, nie może zawierać składowych statycznych oraz referencji
- W danej chwili można odwołać się tylko do jednego pola
- Unia nie może mieć pól z konstruktorami i destruktorami, może natomiast posiadać wskaźniki do takich pól

```
 union Wartosc{
    int wc;
    double wr;
    char* wz;
 };
//...
Wartosc w;
w.wc = 100;
std::cout << w.wr << std::endl; //kompilator nie zglosi zastrzeżen, ale
```

Pola bitowe

- Szczególny rodzaj pola klasy pozwalający na oszczędnośc pamięci to pole bitowe
- Typem pola bitowego musi być liczba całkowita (ze znakiem lub bez znaku)
- Nie można pobrać adresu pola bitowego

```
 class Plik{
 public:
    unsigned modyfikacja:1;
    unsigned tryb:2;
 };
enum {ODCZYT = 01, ZAPIS = 02};
//...
Plik plik;
plik.tryb |= ZAPIS;
```

Klasy zagnieżdzone

- Klasę można zdefiniować wewnątrz innej klasy, zdefiniowana w ten sposób klasa to klasa zagnieżdżona
- Definicja klasy zagnieżdzonej może znaleźć się w każdej sekcji klasy zawierającej
- Klasa zagnieżdzona nie ma żadnych dodatkowych prac do sekcji niepublicznych klasy zawierającej

```
   class Okno{
       class Szyba{
          public:
              int szerokosc, wysokosc;
              void rysuj();
       };
   };
void Okno::Szyba::rysuj()
{
//...
}
```

Klasy lokalne

- Klasę można również zdefiniować wewnątrz funkcji - klasa lokalna
- Klasa lokalna nie może zawierać składowych statycznych

```
void funkcja()
{
  class Lokalna{
    public:
      int a;
      double metoda(double x)
      {
        return 2 * x;
      }
  };
  Lokalna x;
}
```

Konstruktor klasy

- Konstruktor klasy – specjalna metoda klasy służąca do inicjowania obiektów klasy; jeżeli jest zdefiniowana będzie automatycznie zastosowana do każdego obiektu klasy przed jego użyciem; metoda wywoływana automatycznie (niejawnie) zaraz po tym jak obiektowi zostanie przydzielona pamięć; konstruktor definiuje się nadając mu nazwę taką samą jak klasy; nie określa się dla niego wartości przekazywanej (zwracanej); można definiować wiele konstruktorów dla jednej klasy z różnymi parametrami (przeciążenie konstruktora); konstruktor podlega zasadom hermetyzacji

```
class Rachunek
{
public:
  Rachunek(const char* nazwa, double saldo = 0.0);
  // ...
private:
  char* nazwa_;
  unsigned int numer_;
  double saldo_;
};
```

Konstruktor klasy

```
inline Rachunek::Rachunek(const char* nazwa, double saldo)
{
  nazwa_ = new char[strlen(nazwa)+1];
  strcpy(nazwa_, nazwa);
  numer_ = generuj_numer_rachunku();
  saldo_ = saldo;
}
```

```
Rachunek rach2("Wacław␣Czyż", 100);
Rachunek rach3a("Ala␣Kotowska");
Rachunek rach3b = Rachunek("Ala␣Kotowska");
Rachunek rach3c = "Ala␣Kotowska";
Rachunek* wrach4 = new Rachunek("Olek␣Psiarski");
Rachunek trach[10]; // błąd; brak konstr. domyśl.
```

Konstruktor klasy

- Konstruktor domyślny – konstruktor nie wymagający podawania parametrów; posiadający pustą listę argumentów formalnych lub każdy z parametrów formalnych konstruktora ma określoną wartość domyślną

```
class Rachunek
{
public:
  Rachunek();
  // ...
};
```

```
inline Rachunek::Rachunek()
{
  nazwa_ = 0;
  numer_ = 0;
  saldo_ = 0.0;
}
```

```
Rachunek rach5;
Rachunek rach6(); // OK, ale... jest to deklaracja funkcji
```

Konstruktor klasy

- Konstruktor domyślny generowany przez kompilator

```
class B
{
public:
  B() { cout << "ctor␣B\n"; }
};

class A
{
private:
  B b1;
  int i;
  B b2;
};

A a; // ctor B
     // ctor B
```

Konstruktor klasy

- Lista inicjowania składowych – umieszczona między listą argumentów formalnych, a treścią konstruktora; poprzedzona dwukropkiem; zawierająca nazwy pól rozdzielone przecinkami oraz ich wartości początkowe; każde pole może znaleźć się tylko raz na liście; kolejność inicjowania pól zależy od deklaracji tych pól w definicji klasy

```
class ConstRef
{
public:
  ConstRef(int a);
private:
  int i;
  const int ci;
  int &ri;
};
```

```
ConstRef::ConstRef(int a)
{
   i = a;  // przypisanie
   ci = a; // błąd
   ri = i; // błąd
}

ConstRef::ConstRef(int a)
  : ci(a), ri(i) // inicjowanie
{
  i = a; // przypisanie
}
```

Konstruktor klasy

```
inline Rachunek::Rachunek()
  : nazwa_(0), numer_(0), saldo_(0.0)
{ }

inline Rachunek::Rachunek(const char* nazwa, double saldo)
  : saldo_(saldo)
{
  nazwa_ = new char[strlen(nazwa)+1];
  strcpy(nazwa_, nazwa);
  numer_ = generuj_numer_rachunku();
}

// inna wersja
inline Rachunek::Rachunek(const char* nazwa, double saldo)
  : nazwa_(new char[strlen(nazwa)+1]),
    numer_(generuj_numer_rachunku()), saldo_(saldo)
{  strcpy(nazwa_, nazwa); }
```

Konstruktor klasy

- Kolejność wywołania konstruktorów obiektów składowych

```
class B
{
public:
  B() { cout << "ctord␣B\n"; }
  B(int b) { cout << "ctor␣B␣"
                  << b
                  << '\n'; }
};

class C
{
public:
  C() { cout << "ctord␣C\n"; }
  C(int c) { cout << "ctor␣C␣"
                  << c
                  << '\n'; }
};
```

```
class A
{
public:
  A() : c2(2), b2(2)
  { cout << "ctord␣A\n"; }
private:
  B b1;
  C c1;
  B b2;
  C c2;
};

A a; // ctord B
     // ctord C
     // ctor B 2
     // ctor C 2
     // ctord A
```

Konstruktor klasy

- Kostruktory dla obiektu typu `const` – konstruktor nie może być metodą `const;` zastosowanie właściwego konstruktora nie zależy od tego czy obiekt jest typu `const`; ustalenie atrybutu `const` dla obiektu następuje po zainicjowaniu obiektu (po wykonaniu konstruktora) i obowiązuje do chwili wywołanie destruktora tego obiektu; te same zasady odnoszą się także do obiektów zadeklarowanych jako `volatile`

Destruktor klasy

- Destruktor klasy – specjalna metoda klasy wywoływana zawsze gdy obiekt ma być usunięty z pamięci; metoda wywoływana automatycznie (niejawnie) tuż przed usunięciem obiektu; po wykonaniu destruktora następuje zwolnienie pamięci; destruktorowi nadaje się nazwę zgodną z nazwą klasy poprzedzoną znakiem ~; destruktor nie przekazuje żadnej wartości oraz nie przyjmuje żadnych argumentów (nie można go przeciążyć); może być tylko jeden destruktor; podlega hermetyzacji; destruktor można wywołać jawnie

```
class Rachunek
{
public:
  ~Rachunek();
  // ...
};
```

```
inline Rachunek::~Rachunek()
{
  delete[] nazwa_;
  zwolnij_numer_rachunku(numer_);
}
```

Destruktor klasy

- Kolejność wywołania destruktorów obiektów składowych

```
class B
{
public:
  B() {}
  B(int b) {}
  ~B()
  { cout << "dtor␣B\n"; }
};

class C
{
public:
  C() {}
  C(int c) {}
  ~C()
  { cout << "dtor␣C\n"; }
};
```

```
class A
{
public:
  A() : c2(2), b2(2)
  { cout << "ctord␣A\n"; }
  ~A() { cout << "dtor␣A\n"; }
private:
  B b1;
  C c1;
  B b2;
  C c2;                 // ctord A
};                      // dtor B
                        // dtor A
A a;                    // dtor C
B* wb = new B(1);       // dtor B
delete wb;              // dtor C
                        // dtor B
```

Konstruktor klasy

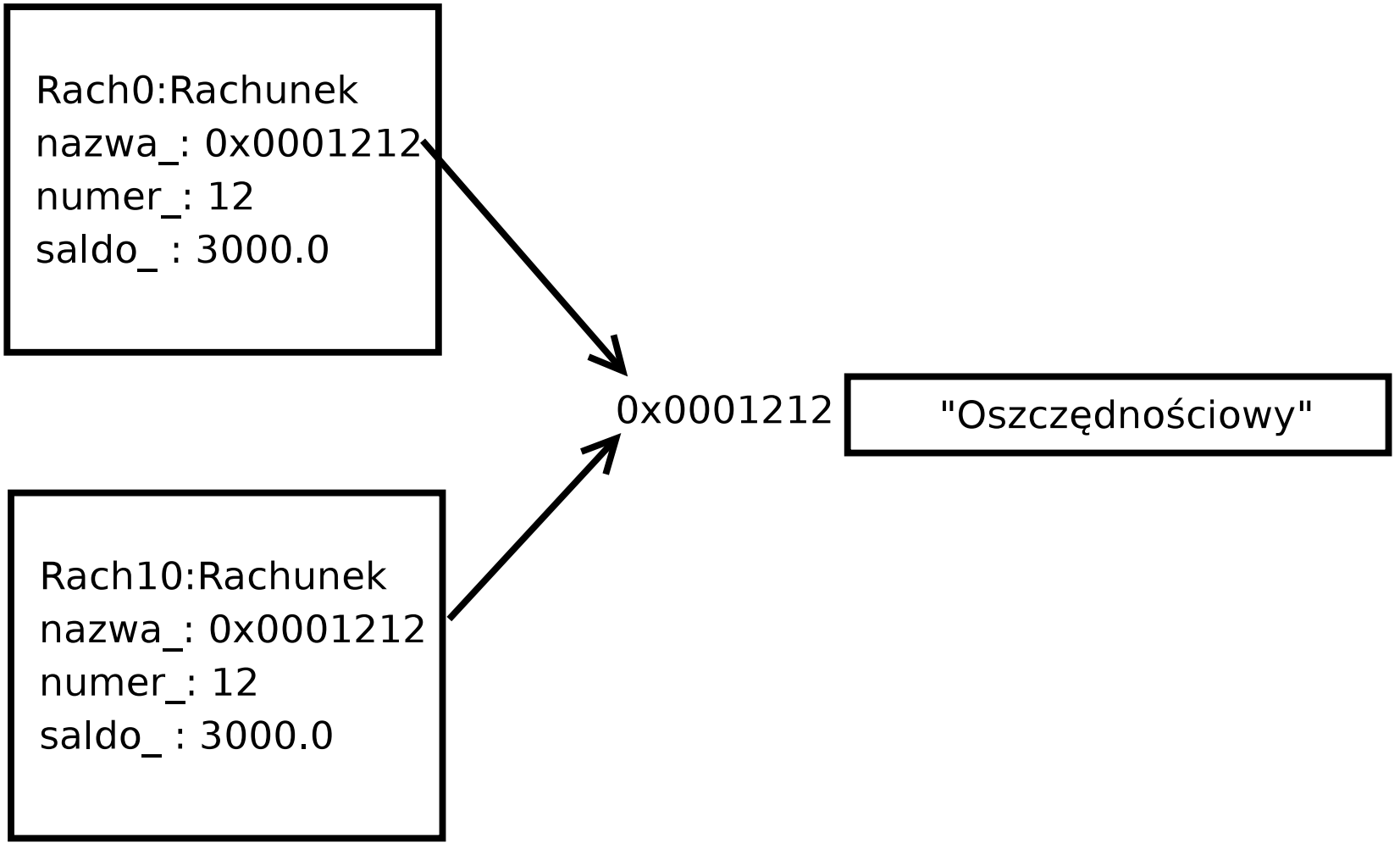
- Konstruktor kopiujący – inicjowanie jednego obiektu klasy wartością innego obiektu klasy; domyślny mechanizm (gdy programista nie zdefiniował konstruktora kopiującego) to inicjowanie składowa po składowej; wykorzystanie konstruktora kopiującego:
  - ▷ jawne inicjowanie jednego obiektu klasy wartością innego obiektu
  - ▷ przekazanie obiektu klasy jako argumentu funkcji (przez wartość)
  - ▷ przekazanie obiektu klasy jako wyniku wywołania funkcji
  - ▷ definicja niepustej kolekcji uporządkowanej
  - ▷ wstawienie obiektu do kolekcji

```
class Rachunek
{
public:
  Rachunek(const Rachunek& rach);
  // ...
};
```

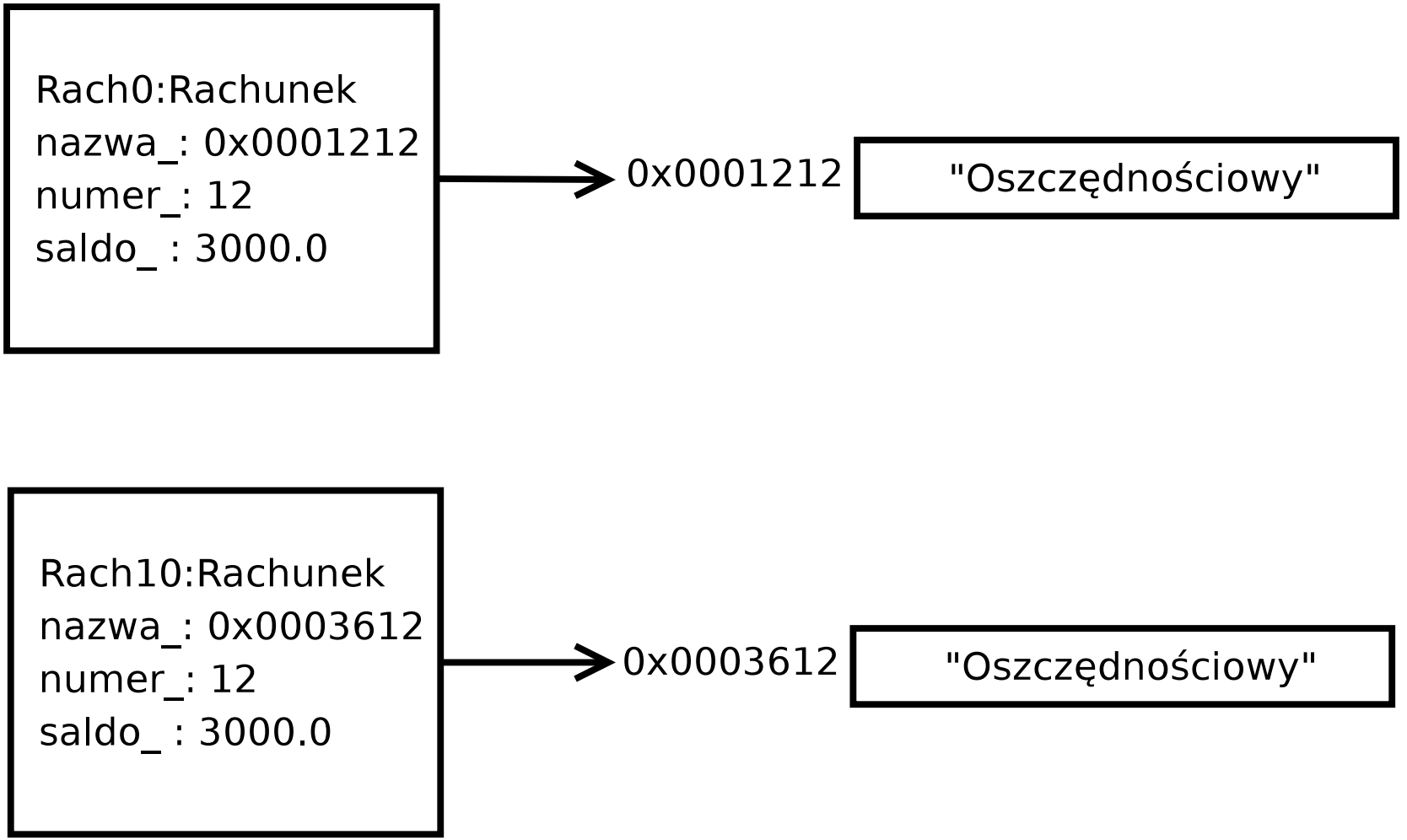
Konstruktor klasy

```
// inicjowanie składowa po składowej (domyślne)
inline Rachunek::Rachunek(const Rachunek& rach)
  : nazwa_(rach.nazwa_),
    numer_(rach.numer_),
    saldo_(rach.saldo_)
{ }

// poprawne kostrukcja obiektu
inline Rachunek::Rachunek(const Rachunek& rach)
  : numer_(rach.numer_), saldo_(rach.saldo_)
{
  nazwa_ = new char[strlen(rach.nazwa_)+1];
  strcpy(nazwa_, rach.nazwa_);
}

Rachunek rach10(rach0);
Rachunek rach11 = rach0;
const Rachunek rach12(rach0);
const Rachunek rach13 = rach0;
```

Konstruktor klasy



Rysunek 1: Mechanizm kopiowania składowa po składowej



Rysunek 2: Konstruktor kopiujący

Konstruktor klasy

- Wywołania konstruktorów kopiujących

```
class A
{
public:
  A() { cout << "ctord" << endl; }
  A(const char* s) { cout << "ctor" << s << endl; }
  A(const A& a) { cout << "cctor" << endl; }
};

void parametr(A a) {}
A wartosc() { A a("kot"); return a; }

A a1("ala");    // ctor ala
A a2 = a1;      // cctor
A a3(a1);       // cctor
parametr(a1);   // cctor
a3 = wartosc(); // ctor kot
                // cctor
```

Konstruktor klasy

- Wywołania konstruktorów kopiujących obiektów składowych

```
class B
{
public:
  B() { }
  B(const B&)
  { cout << "cctor␣B\n"; }
};

class C
{
public:
  C() { }
  C(const C&)
  { cout << "cctor␣C\n"; }
};
```

```
class A
{
  B b1;
  C c1;
  B b2;
  C c2;
};

A a1;
A a2(a1); // cctor B
          // cctor C
          // cctor B
          // cctor C
```

Konstruktor klasy

- Zamiast definiować konstruktor kopiujący, można w ogóle zabronić inicjowania składowa po składowej
  - ▷ deklarując konstruktor kopiujący jako metodę prywatną klasy dając tym możliwość inicjowania składowa po składowej tylko w treści metod klasy oraz funkcjom/metodom/klasom zaprzyjaźnionym
  - ▷ aby uniemożliwić inicjowanie składowa po składowej w również treści metod oraz funkcjom/metodom/klasom zaprzyjaźnionym należy rozmyślnie zrezygnować z definicji konstruktora kopiującego zadeklarowanego w sekcji prywatnej klasy; użycie takiego konstruktora będzie powodowało błędy konsolidacji (linkowania) programu

Przypisywanie

- Przypisanie obiektowi wartości innego obiektu – realizowany przez operator =; domyślny mechanizm to przypisanie składowa po składowej; w sytuacji gdy implementacja klasy wymaga jawnej definicji konstruktora kopiującego na pewno wymagane będzie jawne zdefiniowanie operatora przypisania

```
class Rachunek
{
public:
  inline Rachunek& operator = (const Rachunek& rach);
  // ...
};

// przypisanie składowa po składowej (domyślne) nie zawsze poprawne
inline Rachunek& Rachunek::operator = (const Rachunek& rach)
{
  nazwa_ = rach.nazwa_;
  numer_ = rach.numer_;
  saldo_ = rach.saldo_;
  return *this;
}
```

Przypisywanie

```
inline Rachunek& Rachunek::operator = (const Rachunek& rach)
{
  if (this != &rach) {
    delete[] nazwa_;
    nazwa_ = new char[strlen(rach.nazwa_)+1];
    strcpy(nazwa_, rach.nazwa_);
    numer_ = rach.numer_;     saldo_ = rach.saldo_;
  }
  return *this;
}
// alternatywne rozwiązanie
inline Rachunek& Rachunek::operator = (const Rachunek& rach)
{
  char* tmp = new char[strlen(rach.nazwa_)+1];
  strcpy(tmp, rach.nazwa_);
  delete[] nazwa_;
  nazwa_ = tmp;   numer_ = rach.numer_;    saldo_ = rach.saldo_;
  return *this;
}

Rachunek lech("Lech", 100), czech("Czech");
Rachunek rus = lech; // inicjowanie
rus = czech; // przypisanie
```

Przypisywanie

- Aby całkowicie zapobiec wykonywaniu przypisania składowa po składowej stosuje się ten sam zabieg co dla inicjowania składowa po składowej: operator przypisania deklaruje się jako prywatny i nie definiuje go
- Wywołanie operatorów przypisania obiektów składowych

```
class B
{
public:
  B& operator = (const B& b)
  {
    cout << "op=␣B\n";
    return *this;
  }
};

class A
{
  B b1, b2;
};

A a1, a2;
a2 = a1;   // op= B
           // op= B
```

Tablica obiektów klasy

- Tablica obiektów klasy – definiowana identycznie jak dla typów wbudowanych; każdy element tablicy jest oddzielnie inicjowany konstruktorem domyślnym; można jawnie określić argumenty przekazywane do konstruktora na liście inicjowania; brak możliwości inicjowania elementów tablicy umieszczanej w pamięci dynamicznej innym konstruktorem niż domyślny

```
Rachunek tab[10];

Rachunek rodzina[] =
  {"tata", "mama", "dziecko"};

Rachunek rodzina[] = {
  Rachunek("tata", 1000),
  Rachunek(), // użycie konstr. domyś.
  Rachunek("mama", 800)};
```

```
Rachunek* wrach
  = new Rachunek[10];
delete[] wrach;
```

Operatory przeciążone

- Przeciążanie operatorów – definiowanie wersji operatorów (zdefiniowanych pierwotnie w języku C++) przeznaczonych dla argumentów będących obiektami klas; deklaracja tak samo jak dla zwykłej metody/funkcji; nawa składa się ze słowa kluczowego `operator` poprzedzającego symbol operatora; metodę/funkcję operatora można wywołać jawnie

operatory przeciążalne								
+	-	*	/	%	^	&	|	~
!	,	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&	||	+=	-=	/=
%=	^=	&=	|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]			

operatory nieprzeciążalne								
::	.	.*	?:	sizeof				

Operatory przeciążone

```
class Napis
{
public:
  Napis(const char* = 0);
  Napis(const Napis&);
  ~Napis();
  Napis& operator = (const Napis&);
  int rozmiar() const { return rozmiar_; }
  const char* nps_c() const { return napis_; }
private:
  int rozmiar_;
  char* napis_;
};
```

Operatory przeciążone

```
class Napis
{
public:
  //...
  Napis& operator += (const Napis&);
  Napis& operator += (const char*);
};

Napis& Napis::operator += (const Napis& nps)
{
  if (nps.napis_) {
    Napis tmp(*this);
    rozmiar_ += nps.rozmiar_;
    delete[] napis_;
    napis_ = new char[rozmiar_ + 1];
    strcpy(napis_, tmp.napis_); strcat(napis_, nps.napis_);
  }
  return *this;
}
```

Operatory przeciążone

```
Napis& Napis::operator += (const char* nps)
{
  if (nps) {
    Napis tmp(*this);
    rozmiar_ += strlen(nps);
    delete[] napis_;
    napis_ = new char[rozmiar_ + 1];
    strcpy(napis_, tmp.napis_); strcat(napis_, nps);
  }
  return *this;
}

Napis ala("Ala"), kot("ma␣kota"), haslo;
ala += "␣"; // "Ala "
ala += kot; // "Ala ma kota"
```

Operatory przeciążone

```
class Napis
{
public:
  //...
  Napis operator + (const Napis&);
  Napis operator + (const char*);
};

Napis Napis::operator + (const Napis& nps)
{
  Napis tmp(*this);
  return tmp += nps;
}

Napis Napis::operator + (const char* nps)
{
  Napis tmp(*this);
  return tmp += nps;
}
```

Operatory przeciążone

```
Napis operator + (const char* nps1, const Napis& nps2)
{
   Napis tmp(nps1);
   return tmp += nps2;
}

// dla zapewnienia symetryczności - alternatywne rozwiązanie
Napis operator + (const Napis& nps1, const char* nps2)
{
   Napis tmp(nps1);
   return tmp += nps2;
}

haslo = ala + "␣"; // "Ala "
haslo = "␣" + kot; // " ma kota"
haslo = ala + "␣" + kot; // "Ala ma kota"
```

Operatory przeciążone

```
class Napis
{
public:
  Napis& operator = (const char*);
  bool operator == (const Napis&);
  //...
};

Napis& Napis::operator = (const char* nps)
{
  delete[] napis_;
  rozmiar_ = 0; napis_ = 0;
  if (nps) {
    rozmiar_ = strlen(nps);
    napis_ = new char[rozmiar_+1];
    strcpy(napis_, nps);
  }
  return *this;
}
```

Operatory przeciążone

```
bool Napis::operator == (const Napis& nps)
{
  return !strcmp(napis_, nps.napis_);
}

bool operator == (const char* nps1, const Napis& nps2)
{
  return !strcmp(nps1, nps2.nap_c());
}

Napis ala("Ala"), kot;
kot = "ma␣kota";

bool pytanie = ala == kot; // czy ala zawiera to samo co kot?
pytanie = "Ala" == ala; // czy ala zawiera Ala?
```

Operatory przeciążone

```
class Napis
{
public:
  const char& operator [] (int) const;
  char& operator [] (int);
  //...
};

const char& Napis::operator [] (int i) const
{
  assert(i > -1 && i < rozmiar_); // #include <cassert>
  return napis_[i];
}

char& Napis::operator [] (int i)
{
  assert(i > -1 && i < rozmiar_);
  return napis_[i];
}
```

Operatory przeciążone

```
char a = ala[0]; // a == 'A'
const Napis pies("Pimpek");
a = pies[0]; // a == 'P';

ostream& operator << (ostream& os, const Napis& n)
{
  return os << n.nps_c();
}

istream& operator >> (istream& is, Napis& n)
{
  char buf[256];
  is >> buf; // co w sytuacji gdy napis > 255?
  n = buf;
  return is;
}

cout << ala << "␣" << kot << '\n';
cin >> ala;
```

Operatory przeciążone

```
class Bezwzgledna
{
public:
  int operator () (int wrt)
  {
    return wrt < 0 ? -wrt : wrt;
  }
};

Bezwzgledna funktor; // obiekt funkcji
int a = -1;
int b = funktor(-3), c;
c = funktor(a);
```

Operatory przeciążone

```
class WskNapis
{
public:
  WskNapis(Napis& nps )
    : wsk_(&nps)
  {}
  Napis& operator * () { return *wsk_; }
  Napis* operator -> () { return wsk_; }
private:
  Napis* wsk_;
};

WskNapis wa(ala);
cout << wa->nps_c();
cout << (*wa).nps_c();
```

Operatory przeciążone

```
class WskNapis
{
public:
  WskNapis(Napis& nps, int roz = 0 )
    : wsk_(&nps), rozmiar_(roz), przesuniecie_(0) {}
  Napis& operator * () { return *wsk_; }
  Napis* operator -> () { return wsk_; }
  Napis* operator ++ (); // przedrostkowy
  Napis* operator -- ();
  Napis* operator ++ (int); // przyrostkowy
  Napis* operator -- (int);
private:
  int rozmiar_, przesuniecie_;
  Napis* wsk_;
};

Napis tab[3] = {"ala", "␣ma␣", "kota" };
WskNapis w(*tab, 3);
```

Operatory przeciążone

```
Napis* WskNapis::operator ++ ()
{
  assert(przesuniecie_+1 < rozmiar_);
  przesuniecie_++;
  return *++wsk_;
}

Napis* WskNapis::operator -- (int)
{
  assert(przesuniecie_-1 > -1);
  przesuniecie_--;
  return *wsk_--;
}

cout << w->nps_c();
for (int i = 0; i < 2; i++)
   cout << (++w)->nps_c();
cout << '\n'; // ala ma kota
```

Operatory przeciążone

- Jeżeli operator przeciążony jest zdefiniowany jako metoda klasy, to ta metoda wywoływana jest tylko wtedy, kiedy argument po *lewej* stronie operatora jest obiektem tej klasy. Gdy trzeba użyć operatora przeciążonego w operacji, której lewy argument jest innego typu, wówczas operator przeciążony *musi* być funkcją składową przestrzeni nazw (być w zasięgu)
- Przeciążone operatory: przypisania =, indeksowania `[]`, wywołania funkcji `()`, przyrostkowe i przedrostkowe operatory inkrementacji i dekrementacji `++, --` oraz wyboru składowej `->` muszą być definiowane jako metody klasy

- Materiał do samodzielnego opanowania (nieobowiązkowe)
  - ▷ przeciążanie operatorów `new`, `delete`
  - ▷ przeciążanie operatorów `new[]`, `delete[]`

Funkcje przekształcenia typu

- Funkcja przekształcenia typu – specjalny rodzaj metody, określający przekształcenie definiowane przez użytkownika; deklaracja w treści klasy słowem kluczowym `operator` poprzedzającym nazwę typu, który ma być wynikiem przekształcenia; wynik przekształcenia nie musi być typem wbudowanym; metoda ta ma pustą listę argumentów formalnych oraz nie specyfikuje się typu wartości zwracanej

```
class MalaInt
{
public:
  MalaInt(int wrt) : wartosc_(wrt) {}
  MalaInt operator + (int wrt)
  { return wartosc_ + wrt; }
private:
  int wartosc_;
};

MalaInt mi(3);
mi + 3.14; //  == 6; jaki typ wyr.?
```

Funkcje przekształcenia typu

```
class MalaInt
{
public:
  MalaInt(int wrt) : wartosc_(wrt) {}
  operator int() { return wartosc_; }
  // MalaInt operator + (int wrt)
  // { return wartosc_ + wrt; }
private:
  int wartosc_;
};

mi + 3.14; //  == 6.14;  jaki typ wyr.?
```

Funkcje przekształcenia typu

```
class Leksem
{
public:
  Leksem(const string& nzw, int wrt)
    : nazwa_(nzw), wartosc_(wrt) { }
  operator MalaInt() { return wartosc_; }
  operator string() { return nazwa_; }
  operator int() { return wartosc_; }
private:
  string nazwa_;
  MalaInt wartosc_;
};
```

Funkcje przekształcenia typu

```
Leksem lek("rudy", 102);

string s = lek; // s == "rudy"
int i = lek; // i == 102;
MalaInt mi = lek; // mi.wartosc_ = 102;

void wylicz(int);
wylicz(lek); // przekształcenie użytkownika

void oblicz(double);
oblicz(lek); // przekształcenie użytk. + stand.
```

Funkcje przekształcenia typu

- Konstruktor jako funkcja przekształcenia typu – zestaw konstruktorów jednoargumentowych zdefiniowanych dla klasy określa zbiór niejawnych przekształceń wartości argumentów formalnych konstruktora w wartości obiektów klasy; wyłączenie konwersji przez konstruktor modyfikatorem `explicit`

```
void oblicz(MalaInt);
int a = 4;
oblicz(a); // niejawna konwersja

class MalaInt
{
public:
  explicit MalaInt(int wrt);
  //...
};
```

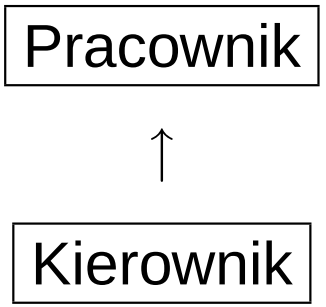
Klasy pochodne i dziedziczenie

- Dziedziczenie – mechanizm pozwalający definiować nową klasę (klasę pochodną) w oparciu o klasę już istniejącą (klasę podstawową, bazową); określane jest przez listę dziedziczenia klasy (`: poziom_dostępu klasa_bazowa`)
- Klasa pochodna – klasa zdefiniowana przez dodawanie “udogodnień” do istniejącej klasy bez ponownego programowania i kompilowania klasy podstawowej; klasa pochodna dziedziczy wszystkie pola i metody swojej klasy podstawowej; klasa pochodna jest podtypem klasy bazowej (w przypadku dziedziczenia publicznego)

Klasy pochodne i dziedziczenie

- Polimorfizm – relacja między typem a podtypem, dzięki której bez interwencji programisty wskaźnik lub referencja do obiektu klasy podstawowej mogą odnosić się do obiektu dowolnego podtypu klasy; polimorfizm istnieje tylko wewnątrz poszczególnych hierarchii klas; korzystanie z polimorfizmu za pomocą mechanizmów
  - ▷ niejawnych przekształceń typu wskaźników do klas pochodnych, referencji do wskaźników lub referencji do publicznego typu podstawowego
  - ▷ funkcji wirtualnych
  - ▷ operacji rzutowania dynamicznego (operator `dynamic_cast`)
- Klasa abstrakcyjna – “niepełna” klasa, która staje się mniej lub bardziej ukończona w wyniku każdego kolejnego dziedziczenia po niej przez inną klasę; zapewnienie wspólnego interfejsu dla kilku różnych klas tworzących hierarchię dziedziczenia; klasa nie tworzy obiektów

Klasy pochodne i dziedziczenie




```
class Pracownik
{
protected:
  string imie, nazwisko;
  Data zatrudniony_od;
  short dzial;
};
```

```
class Kierownik : public Pracownik
{
protected:
  Pracownik* zespol;
  short poziom;
};

Kierownik* wk, kier;
Pracownik* wp, prac;
wp = &kier;
wk = &prac; // błąd
```

Dostęp do składowych klasy podstawowej

- Obiekt klasy pochodnej składa się z:
  - ▷ podobiektu utworzonego z pól niestatycznych klasy podstawowej (pola odziedziczone)
  - ▷ części pochodnej składającej się z pól niestatycznych klasy pochodnej (pola deklarowane w klasie pochodnej)

  W treści klasy pochodnej można się odnosić bezpośrednio do składowych podobiektu odziedziczonych po klasie podstawowej, tak jak gdyby to były składowe danej klasy pochodnej; głębokość łańcucha dziedziczenia nie ogranicza dostępu do tych składowych ani nie zwiększa kosztu tego dostępu; metody klasy podstawowej wywołujemy tak jak by to były metody klasy pochodnej

Dostęp do składowych klasy podstawowej

```
class AB
{
public:
  AB() : a_(1), b_(2.2) {}
  void wyswietlAB(ostream& os);
  void wyswietlAB();
protected:
  int a_;
  double b_;
};

void AB::wyswietlAB(ostream& os)
{ os << a_ << ' ' << b_; }

void AB::wyswietlAB()
{ cout << a_ << ' ' << b_; }
```

```
class CD : public AB
{
public:
  CD() : c_(3), d_(4.4) {}
  void wyswietlCD(ostream& os);
  void wyswietlABCD(ostream& os);
  void wyswietlABCD2(ostream& os);
protected:
  int c_;
  double d_;
};

void CD::wyswietlCD(ostream& os)
{ os << c_ << ' ' << d_; }

void CD::wyswietlABCD(ostream& os)
{ os << a_ << ' ' << b_ << ' '
     << c_ << ' ' << d_; }
```

Dostęp do składowych klasy podstawowej

```
void CD::wyswietlABCD2(ostream& os);
{
  wyswietlAB(os);
  os << ' ';
  wyswietlCD(os);
}

AB ob1;
ob1.wyswietlAB(cout); // 1 2.2

CD ob2;
ob2.wyswietlCD(cout);   // 3 4.4
ob2.wyswietlABCD(cout); // 1 2.2 3 4.4
ob2.wyswietlABCD2(cout); // 1 2.2 3 4.4
ob2.wyswietlAB(cout); // 1 2.2
```

Dostęp do składowych klasy podstawowej

- Przysłanianie składowych z klasy podstawowej – dwa różne zasięgi nazw; brak niejawnego przeciążenia nazw z klasy podstawowej w klasie pochodnej

```
class CD : public AB
{
public:
  CD() : c_(3), d_(4.4), a_("jeden") {}
  // ...
  void wyswietlAB()
  { cout << a_ << ' ' << c_ << ' ' << d_; }
  void wyswietlABCD2(ostream& os);
protected:
  // ...
  string a_;
};
```

Dostęp do składowych klasy podstawowej

```
void CD::wyswietlABCD2(ostream& os)
{
  AB::wyswietlAB(os);
  os << ' ';
  wyswietlCD(os);
}

CD ob3;
ob3.wyswietlCD(cout);   // 3 4.4
ob3.wyswietlABCD(cout); // jeden 2.2 3 4.4
ob3.wyswietlABCD2(cout); // 1 2.2 3 4.4
ob3.wyswietlAB(); // jeden 3 4.4
ob3.AB::wyswietlAB(cout); // 1 2.2
```

Dostęp do składowych klasy podstawowej

- Jawne przeciążenia metod w klasie pochodnej metodami z klasy podstawowej

```
class CD : public AB
{
public:
  // ...
  void wyswietlAB();
  void wyswietlABCD2(ostream& os);
  using AB::wyswietlAB;
  // ...
};
```

```
void CD::wyswietlABCD2(ostream& os)
{
  wyswietlAB(os);
  os << ' ';
  wyswietlCD(os);
}
```

```
CD ob4;
ob4.wyswietlABCD2(cout); // 1 2.2 3 4.4
ob4.wyswietlAB(); // jeden 3 4.4
ob4.wyswietlAB(cout); // 1 2.2
```

Dostęp do składowych klasy podstawowej

- Klasa pochodna ma dostęp do składowych chronionych i prywatnych innych obiektów własnej klasy

```
double CD::test1(const CD& cd)
{ return d_ + cd.d_; }
```

- Klasa pochodna ma dostęp do składowych chronionych swojego podobiektu klasy podstawowej

```
double CD::test2(const CD& cd)
{ return b_ + cd.d_; }
```

- Klasa pochodna ma dostęp do składowych chronionych klasy podstawowej w innych obiektach własnej klasy

```
double CD::test3(const CD& cd)
{ return b_ + cd.b_; }
```

Dostęp do składowych klasy podstawowej

- Klasa pochodna nie ma dostępu do składowych chronionych niezależnego obiektu klasy podstawowej

```
double CD::test4(const AB& ab)
{ return b_ + ab.b_; } // błąd
```

- Klasa pochodna ma dostęp do składowych chronionych i prywatnych niezależnego obiektu klasy podstawowej jeśli została z nią zaprzyjaźniona

```
class AB
{
  friend class CD;
  //...
};
```

Dostęp do składowych klasy podstawowej

- Wskaźnik do obiektu klasy podstawowej ma dostęp do pól i metod (w tym i metod wirtualnych) deklarowanych (lub odziedziczonych) w jego klasie, niezależnie od typu tego obiektu na jaki faktycznie może wskazywać; klasa wskaźnika określa interfejs obiektu pod wskaźnikiem

```
AB *wab = new CD;
```

  - ▷ jeżeli w klasie podstawowej i pochodnej jest deklarowana metoda niewirtualna mająca taką samą nazwę, to za pomocą wskaźnika będzie wywołana wersja metody klasy podstawowej

```
wab->wyswietlAB(); //wołanie AB::wyswietlAB()
```

  - ▷ jeżeli w klasie podstawowej i pochodnej jest deklarowane pole mające taką samą nazwę to wskaźnik będzie wskazywał na wersję pola z klasy podstawowej

```
wab->a_; //dostęp do AB::a_ typu int
```

Dostęp do składowych klasy podstawowej

- jeżeli w klasie pochodnej jest deklarowana metoda niewirtualna (lub pole) o nazwie która nie występuje w klasie podstawowej to za pomocą wskaźnika nie można jej wywołać bezpośrednio

```
//próba wołanie CD::wyswietlABCD()
wab->wyswietlABCD(); // błąd
static_cast<CD*>(wab)->wyswietlABCD();

//próba dostępu do CD::d_
wab->d_; // błąd
static_cast<CD*>(wab)->d_;
```

Konstruktory i destruktor klasy pochodnej

- Klasa pochodna nie dziedziczy konstruktorów klasy podstawowej; każda klasa pochodna musi mieć zdefiniowany własny zbiór konstruktorów
- Konstruktor klasy pochodnej ma prawo wywołać tylko konstruktor bezpośredniej klasy podstawowej (wyjątek: dziedziczenie wirtualne)
- Konstruktor klasy pochodnej nie powinien bezpośrednio przypisywać wartości polu klasy podstawowej, lecz przekazywać wartość do odpowiedniego konstruktora klasy podstawowej; unikanie ścisłego powiązania (implementacyjnego) między klasą pochodną i podstawową, by nie utrudniać modyfikowania lub rozszerzania klasy podstawowej; należy zadbać o odpowiedni zbiór konstruktorów w klasie bazowej

```
class AB
{
public:
  AB() : a_(0), b_(0.0) {}
  AB(int a, double b) : a_(a), b_(b) {}
  //...
};
```

Konstruktory i destruktor klasy pochodnej

```
class CD : public AB
{
public:
  CD()
    : c_(0), d_(0.0) {}
  CD(int c, double d, const string& a)
    : c_(c), d_(d), a_(a) {}
  CD(int a0, double b, int c, double d, const string& a)
    : AB(a0, b), c_(c), d_(d), a_(a) {}
protected:
  // ...
  string a_;
};
```

Konstruktory i destruktor klasy pochodnej

```
CD ob5;
ob5.wyswietlABCD(cout); //   0 0 0
ob5.wyswietlAB(cout); // 0 0

CD ob6(3, 4.4, "jeden");
ob6.wyswietlABCD(cout); // jeden 0 3 4.4
ob6.wyswietlAB(cout); // 0 0

CD ob7(1, 2.2, 3, 4.4, "jeden");
ob7.wyswietlABCD(cout); // jeden 2.2 3 4.4
ob7.wyswietlAB(cout); // 1 2.2
```

Konstruktory i destruktor klasy pochodnej

- Kolejność wywoływania konstruktorów
  - ▷ konstruktor klasy podstawowej; jeżeli jest kilka klas podstawowych, to konstruktory są wywoływane w kolejności występowania tych klas w liście dziedziczenia
  - ▷ konstruktor obiektu klasy składowej; jeżeli jest kilka obiektów klas składowych, to konstruktory są wywoływane w kolejności występowania deklaracji tych obiektów w definicji klasy
  - ▷ konstruktor klasy pochodnej
- Wywołanie destruktorów przebiega w odwrotnej kolejności niż konstruktorów

Konstruktory i destruktor klasy pochodnej

```
class KS // klasa składowa
{
public:
  KS() : p_(0)
  { cout << "ctord: KS" << p_ << endl; }
  KS(int p) : p_(p)
  { cout << "ctor: KS" << p_ << endl; }
  ~KS()
  { cout << "dtor: KS" << p_ << endl; }
  const  int& p() { return p_; }
protected:
  int p_;
};
```

Konstruktory i destruktor klasy pochodnej

```
class KB // klasa bazowa
{
public:
  KB()
  { cout << "ctord:␣KB" << p0_.p() << p1_.p() << endl; }
  KB(int p1) : p1_(p1)
  { cout << "ctor:␣KB" << p0_.p() << p1_.p() << endl; }
  ~KB()
  { cout << "dtor:␣KB" << p0_.p() << p1_.p() << endl; }
protected:
   KS p0_, p1_;
};
```

Konstruktory i destruktor klasy pochodnej

```
class KP : public KB  // klasa pochodna
{
public:
  KP()
  { cout << "ctord: KP" << p0_.p() << p1_.p() << p2_.p() << endl; }
  KP(int p1, int p2) : KB(p1), p2_(p2)
  { cout << "ctor: KP" << p0_.p() << p1_.p() << p2_.p() << endl; }
  ~KP()
  { cout << "dtor: KP" << p0_.p() << p1_.p() << p2_.p() << endl; }
protected:
  KS p2_;
};
```

Konstruktory i destruktor klasy pochodnej

```
KP o;
ctord: KS0
ctord: KS0
ctord: KB00
ctord: KS0
ctord: KP000

dtor: KP000
dtor: KS0
dtor: KB00
dtor: KS0
dtor: KS0
```

```
KP o(1, 2);
ctord: KS0
ctor: KS1
ctor: KB01
ctor: KS2
ctor: KP012

dtor: KP012
dtor: KS2
dtor: KB01
dtor: KS1
dtor: KS0
```

Konstruktory i destruktor klasy pochodnej

- Konstruktor kopiujący i operator przypisania klasy pochodnej

```
class A
{
public:
  explicit A(const string& nps = "")
    : nps_(new string(nps))
  { }
  A(const A& a);
  A& operator = (const A& a);
  ~A() { delete nps_; }
protected:
  string* nps_;

friend
  ostream& operator << (ostream& os, const A& a);
};
```

Konstruktory i destruktor klasy pochodnej

```
inline A::A(const A& a)
  : nps_(new string(*a.nps_));
{ }

inline A& A::operator = (const A& a)
{
  if (this != &a)   // to nie jest konieczne
    *nps_ = *a.nps_;
  return *this;
}

ostream& operator << (ostream& os, const A& a)
{
  return os << *a.nps_;
}
```

Konstruktory i destruktor klasy pochodnej

```
class B : public A
{
public:
  explicit B(int b, string nps = "")
    : A(nps), b_(new int(b)) {}
  B(const B& b);
  B& operator = (const B& b);
  ~B() { delete b_; }
protected:
  int* b_;
friend
  ostream& operator << (ostream& os, const B& b);
};

ostream& operator << (ostream& os, const B& b)
{
  return os << static_cast<const A&>(b) << '\t' << *b.b_;
}
```

Konstruktory i destruktor klasy pochodnej

```
// domyślny mechanizm inicjowania składowa po składowej
inline B::B(const B& b)
  : A(b), b_(b.b_)
{ }

// domyślny mechanizm przypisania składowa po składowej
inline B& B::operator = (const B& b)
{
  if (this == &b) return *this;
  this->A::operator = (b);
  b_ = b.b_;
  return *this;
}
```

Konstruktory i destruktor klasy pochodnej

```
inline B::B(const B& b)
  : A(b), b_(new int(*b.b_))
{ }

inline B& B::operator = (const B& b)
{
  if (this == &b) return *this; // to nie jest konieczne
  this->A::operator = (b);   // *static_cast<A*>(this) = b;
  *b_ = *b.b_;
  return *this;
}

B b(3, "ala");
cout << b << endl;    // ala    3
B b1 = b, b2(2);
b2 = b;
cout << b1 << ' ' << b2 << endl;   // ala    3 ala    3
```

Klasa pochodna

- Dziedziczenie publiczne – dziedziczenie typu; klasa pochodna jest pod typem klasy podstawowej i zastępuje implementację wszystkich metod zależnych od typu, zachowując implementację metod wspólnych
- Dziedziczenie prywatne – dziedziczenie implementacyjne; składowe publiczne klasy podstawowej stają się składowymi prywatnymi klasy pochodnej; klasa pochodna nie obsługuje bezpośrednio interfejsu publicznego klasy podstawowej; dostarcza własny interfejs publiczny; korzysta z implementacji zdefiniowanej dla klasy podstawowej

```
class A
{
public:
  A(int a_) : a(a_) {}
protected:
  int a;
};
```

```
class B : private A
{
public:
  B(int a_, int b_) : A(a_), b(b_) {}
protected:
  int b;
};
```

```
A* a = new B(3, 3); // błąd
```

Klasa pochodna

- Dziedziczenie chronione – jest również dziedziczeniem implemenetacyjnym; składowe publiczne klasy podstawowej stają się składowymi chronionymi klasy pochodnej, dostępnymi dla klas dalej wyprowadzanych z klasy pochodnej
- Uwalnianie poszczególnych składowych – przywrócenie poziomu dostępu składowej z niepublicznej klasy podstawowej; np. udostępnianie składowych chronionych prywatnej klasy podstawowej dla klas dalej wyprowadzanych z klasy pochodnej

```
class A
{
public:
  A(int a_) : a(a_) {}
  void pokaz()
  { cout << a; }
protected:
  int a;
};
```

```
class B : private A
{
public:
  B(int a_, int b_) : A(a_), b(b_) {}
  using A::pokaz;
  void pokazuj() { pokaz(); cout << b; }
protected:
  int b;
};
```

```
B b(2, 3); b.pokazuj(); b.pokaz(); // 232
```

Metody wirtualne

- Metody klasy są domyślnie niewirtualne; metodę wirtualną specyfikuje się w definicji klasy podstawowej słowem kluczowym `virtual`
- Klasa wprowadzająca (podstawowa) metodę wirtualną musi ją albo definiować albo deklarować jako metodę czysto wirtualną; jeżeli dla klasy pochodnej jest zdefiniowana wersja metody wirtualnej to zastępuje ona wersje dziedziczoną po klasie podstawowej; “wirtualność” metod pozwala na zmianę jej działania w klasie pochodnej niezależnie od kontekstu wywołania
- Aby wersja metody wirtualnej dla klasy pochodnej mogła zastępować wersję jej klasy podstawowej, musi być spełniony warunek ścisłej zgodności prototypów tych metod

Metody wirtualne

```
class A
{
public:
  void go() { virt(); nvirt(); }
protected:
  virtual void virt() { cout << "virtA "; }
  void nvirt() { cout << "nvirtA "; }
};

class B : public A
{
protected:
  virtual void virt() { cout << "virtB "; }
  void nvirt() { cout << "nvirtB "; }
};

B b;
b.go(); // virtB nvirtA
```

Metody wirtualne

- Polimorfizm metod występuje tylko wtedy gdy do podtypu klasy pochodnej odnosimy się pośrednio za pomocą albo referencji, albo wskaźnika do obiektu klasy podstawowej; mechanizm wirtualny metod działa tylko wtedy, gdy używamy wskaźników lub referencji
- Dynamiczne (późne) wiązanie – technika zmiany nazwy występującej w kodzie źródłowym programu na wartość (adres) w czasie wykonania programu
- Statyczne (wczesne) wiązanie – technika w której wiązanie identyfikatorów (nazw) z wartościami następuje przed uruchomieniem programu, w czasie kompilacji i konsolidacji

Metody wirtualne

```
class AB
{
public:
  virtual void wyswietl()
  { cout << a_ << ' ' << b_; }
  //...
};
```

```
class CD : public AB
{
public:
  void wyswietl()
  { cout << c_ << ' ' << d_; }
  //...
};
```

```
// statyczne wywołanie metody wirtualnej
AB o1(1, 2.2);
o1.wyswietl(); // 1 2.2
CD o2(1, 2.2, 3, 4.4, "jeden");
o2.wyswietl(); // 3 4.4

AB* wab = &o2;
o2.wyswietlAB(); // jeden 3 4.4
wab->wyswietlAB(); // 1 2.2
// dynamiczne wywołanie metody wirtualnej
wab->wyswietl(); // 3 4.4
```

Metody wirtualne

```
// Biblioteczna Klasa Usługowa
class BKU
{
public:
  BKU() : p_(0) { }
  BKU(int p) : p_(p) { }
  virtual int licz() const { return p_; }
  int licz_nov() const { return p_; }
protected:
  int p_;
};
```

Metody wirtualne

```
// Biblioteczna Klasa Główna
class BKG
{
public:
  BKG(int v, BKU* kbu = 0)
    : v_(v), kbu_(kbu ? kbu : new BKU(v))
  { }
  ~BKG() { delete kbu_; }
  void oblicz(int k)
  { cout << kbu_->licz() * k * v_ << '\t'
         << kbu_->licz_nov() * k * v_ << endl; }
protected:
  BKU* kbu_;
  int v_;
};

BKG m00(3);
m00.oblicz(2); // 18 18
```

Metody wirtualne

```
// Moja Klasa Usługowa
class MKU : public BKU
{
public:
  MKU() : z_(0) { }
  MKU(int p, int z) : BKU(p), z_(z) { }
  virtual int licz() const { return z_ + p_; }
  int licz_nov() const { return z_ + p_; }
protected:
  int z_;
};

BKG m01(3, new BKU(3));
m01.oblicz(2); // 18 18

BKG m1(3, new MKU(3, 2));
m1.oblicz(2); // 30 18
```

Metody wirtualne

- Statyczne wywołanie metody wirtualnej – jawny wybór wersji metody wirtualnej przy pomocy operatora zasięgu unieważnia mechanizm wirtualny i rozstrzyganie odbywa się w sposób statyczny podczas kompilowania programu

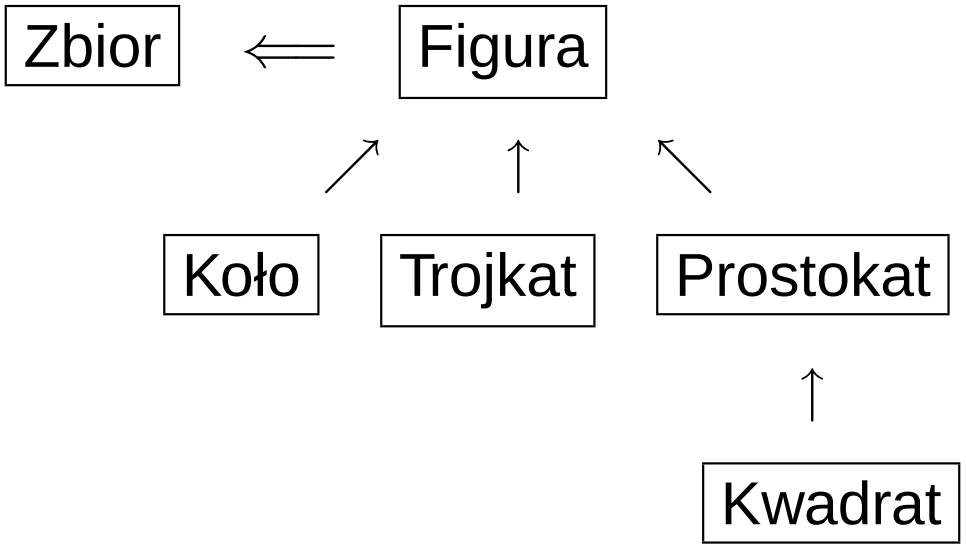
```
wab->AB::wyswietl(); // 1 2.2
```

- Metody czysto wirtualne – konstrukcja składniowa pozwalająca zaznaczyć że dana metoda wirtualna jest stworzona jako interfejs do następnych zastępujących ją metod podtypów (klas pochodnych); służy do zajmowania miejsca w interfejsie klasy, które następnie będzie zajmowane przez kolejno tworzone wersje metod dla typów pochodnych; próba wywołania metody czysto wirtualnej zakończy się błędem; deklaracja metody czysto wirtualnej jest zakończona przypisaniem wartości `0`; można definiować metody czysto wirtualne

```
class AB
{
public:
  virtual void wyswietl(ostream&) = 0;
  //...
};
```

Metody wirtualne

- Klasa zawierająca (lub dziedzicząca) co najmniej jedną metodę czysto wirtualną jest rozpoznawana jako klasa abstrakcyjna; próba utworzenia obiektu takiej klasy sygnalizowana jest błędem kompilatora; obiekt klasy abstrakcyjnej może występować jedynie jako podobiekt obiektów klas pochodnych



Metody wirtualne

```
class Figura
{
public:
  Figura(const string& nazwa)
    : nazwa_(nazwa)
  { }
  virtual double pole() const = 0;
  const string& nazwa() const { return nazwa_; }
protected:
  string nazwa_;
};
```

Metody wirtualne

```
class Kolo : public Figura
{
public:
  Kolo(double r) : Figura("koło"), r_(r) { }
  double pole() const { return M_PI * r_ * r_; }
protected:
  double r_;
};

class Trojkat : public Figura
{
public:
  Trojkat(double b, double h)
    : Figura("trójkąt"), b_(b), h_(h) { }
  double pole() const { return 0.5 * b_ * h_; }
protected:
  double b_, h_;
};
```

Metody wirtualne

```
class Prostokat : public Figura
{
public:
  Prostokat(double a, double b)
    : Figura("prostokąt"), a_(a), b_(b) { }
  double pole() const { return a_ * b_; }
protected:
  Prostokat(const string& nazwa, double a, double b)
    : Figura(nazwa), a_(a), b_(b) { }
  double a_, b_;
};

class Kwadrat : public Prostokat
{
public:
  Kwadrat(double a)
    : Prostokat("kwadrat", a, a) { }
};
```

Metody wirtualne

```
class Zbior
{
public:
  void dodaj(Figura* figura)
  { zbior_.push_back(figura); }
  void pokaz();
  ~Zbior();
protected:
  vector<Figura*> zbior_; // #include <vector>
private:
  Zbior(const Zbior&); // tylko deklaracja
  Zbior& operator = (const Zbior&); // tylko deklaracja
};

Zbior::~Zbior()
{
  for (unsigned i = 0; i < zbior_.size(); ++i)
    delete zbior_[i];
}
```

Metody wirtualne

```
void Zbior::pokaz()
{
  for (unsigned i = 0; i < zbior_.size(); ++i)
    cout << "oto␣" << zbior_[i]->nazwa()
         << "␣jego␣pole␣to␣"
         << zbior_[i]->pole() << '\n';
}

Zbior figury;
figury.dodaj(new Kolo(2.52313));
figury.dodaj(new Kwadrat(3.16228));
figury.dodaj(new Trojkat(3.030303, 3.3));
figury.dodaj(new Prostokat(2.753, 0.9081));
figury.pokaz();

// oto koło jego pole to 20
// oto kwadrat jego pole to 10
// oto trójkąt jego pole to 5
// oto prostokąt jego pole to 2.5
```

Metody wirtualne

- Argumenty domyślne metod wirtualnych – statyczne podstawienie argumentów

```
class A
{
public:
  virtual void pokaz(char p = 'a')
  { cout << "Wersja A:" << p << '\n'; }
};

class B : public A
{
public:
  void pokaz(char p = 'b')
  { cout << "Wersja B:" << p << '\n'; }
};
```

```
B* b = new B;
b->pokaz(); // Wersja B:b
A* a = b;
a->pokaz(); // Wersja B:a
```

Metody wirtualne

- Destruktor wirtualny – jeżeli w klasie podstawowej (dla danej hierarchii dziedziczenia) zadeklarowana jest co najmniej jedna metoda wirtualna to destruktor klasy podstawowej powinien być również zadeklarowany jako wirtualny; może być czysto wirtualny, ale wymaga defincji

```
KP o(1, 2);
```

```
ctord: KS0
ctor: KS1
ctor: KB01
ctor: KS2
ctor: KP012
dtor: KP012
dtor: KS2
dtor: KB01
dtor: KS1
dtor: KS0
```

```
KB* k = new KP(1, 2);
delete k;
```

```
ctord: KS0
ctor: KS1
ctor: KB01
ctor: KS2
ctor: KP012
dtor: KB01
dtor: KS1
dtor: KS0
```

Metody wirtualne

```
// deklaracja destruktora
// jako wirtualnego

class KB
{
public:
  virtual ~KB();
  //...
};
```

```
KB* k = new KP(1, 2);
delete k;

ctord: KS0
ctor: KS1
ctor: KB01
ctor: KS2
ctor: KP012
dtor: KP012
dtor: KS2
dtor: KB01
dtor: KS1
dtor: KS0
```

Metody wirtualne

- Metoda wirtualna w konstruktorze/destruktorze – wersja metody wirtualnej, która jest wywoływana w konstruktorze/destruktorze klasy podstawowej jest zawsze wersją wirtualną, aktywną w klasie podstawowej

```
class A
{
public:
  A() : a(1) { cout << "ctor␣A:"; pokaz(); }
  virtual ~A() { cout << "dtor␣A:"; pokaz(); }
  virtual void pokaz()
  { cout << "wersja␣A:" << a << '\n'; }
protected:
  int a;
};
```

Metody wirtualne

```
class B : public A
{
public:
  B() : b(2) { cout << "ctor␣B\n"; }
  ~B() { cout << "dtor␣B\n"; }
  void pokaz()
  { cout << "wersja␣B:"<< a << b << '\n'; }
protected:
  int b;
};

A* b = new B;              // ctor A:wersja A:1
b->pokaz();                // ctor B
delete b;                  // wersja B:12
                           // dtor B
                           // dtor A:wersja A:1
```

Metody wirtualne

- “Wirtualne konstruktory” – klonowanie i tworzenie obiektów bez dokładnej znajomości typu tworzonego obiektu; wirtualne metody składowe zwracające wskaźnik do obiektu utworzonego (w pamięci dynamicznej) w oparciu o konstruktor kopiujący lub domyślny

```
class Figura
{
public:
  //...
  virtual ~Figura() { }
  virtual Figura* klonuj() const = 0;
  //...
};

class Kolo : public Figura
{
public:
  //...
  Kolo* klonuj() const { return new Kolo(*this); }
};
```

Metody wirtualne

```
class Trojkat : public Figura
{
public:
  //...
  Trojkat* klonuj() const { return new Trojkat(*this); }
};

class Prostokat : public Figura
{
public:
  //...
  Prostokat* klonuj() const { return new Prostokat(*this); }
};

class Kwadrat : public Prostokat
{
  //...
  Kwadrat* klonuj() const { return new Kwadrat(*this); }
};
```

Metody wirtualne

```
class Zbior
{
public:
  Zbior() { }
  Zbior(const Zbior& z);
  Zbior& operator = (const Zbior& z);
  //...
};

Zbior::Zbior(const Zbior& z)
{
  for (unsigned i = 0; i < z.zbior_.size(); ++i)
    dodaj(z.zbior_[i]->klonuj());
}
```

Metody wirtualne

```
Zbior& Zbior::operator = (const Zbior& z)
{
  if (this == &z) return *this;

  for (unsigned i = 0; i < zbior_.size(); ++i)
    delete zbior_[i];
  zbior_.clear();

  for (unsigned i = 0; i < z.zbior_.size(); ++i)
    dodaj(z.zbior_[i]->klonuj());

  return *this;
}
```

- RTTI – identyfikacja typu obiektu podczas wykonania programu (*Run–Time Type Identification*); mechanizm wykorzystywany w sytuacjach gdy typ obiektów można określić jedynie podczas wykonywania programu; dostępne dla obiektów klas posiadających co najmniej jedną metodę wirtualną; dynamiczne określanie typów obiektów; statyczny system określania typów (podczas kompilacji) jest bezpieczniejszy i wydajniejszy
- Operator `typeid` – określenie typu obiektu; określenie typu wyrażenia; zwraca obiekt klasy `type_info`; wymaga deklaracji pliku nagłówkowego `<typeinfo>`

```
int ii = 0, *iw = &ii;
cout << typeid(ii).name() << ’␣’
     << typeid(iw).name() << ’\n’;          // i Pi
double dd = 0.0, *dw = &dd;
cout << typeid(dd).name() << ’␣’
     << typeid(dw).name() << ’\n’;          // d Pd
```

```
cout << typeid(123).name() << ’␣’
     << typeid(3.14).name() << ’\n’;     // i d

int ii = 0, *iw = &ii;
cout << (typeid(int) == typeid(ii));     // 1
cout << (typeid(int) == typeid(iw));     // 0
cout << (typeid(int) != typeid(1/3));    // 0
cout << (typeid(int) != typeid(1.0/3));  // 1
```

```
class A
{
public:
  virtual void met() const = 0;
};
```

```
class B : public A
{
public:
  void met() const
  { cout << "metB\n";}
};
```

```
class C : public A
{
public:
  void met() const
  { cout << "metC\n";}
};
```

```
void pokaz(const A& a)
{
  cout << typeid(a).name() << ' '
       << typeid(&a).name() << ' ';
  a.met();
}

void pokaz(const A* a)
{
  cout << typeid(a).name() << ' '
       << typeid(*a).name() << ' ';
  a->met();
}
```

```
B b;
C c;
pokaz(b); // 1B PC1A metB
pokaz(c); // 1C PC1A metC
pokaz(&b); // PC1A 1B metB
pokaz(&c); // PC1A 1C metC
```

- Operator `dynamic_cast` – rzutowanie typu w trakcie wykonania programu; przekształcenie wskaźnika wskazującego na obiekt klasy we wskaźnik do obiektu innej klasy w ramach tej samej hierarchii (ew. przekształcenie l–wartości obiektu w referencję); przy niepowodzeniu rzutowania dla wskaźnika zostanie zwrócona wartość `0`, dla referencji zgłoszona zostanie sytuacja wyjątkowa `bad_cast`

```
class C : public A
{
public:
  C() : c(3.14) { }
  void met() const { cout << "metC\n";}
  void inna() const { cout << c << "␣innaC\n";}
protected:
  double c;
};
```

```
void drukuj(A* a)
{
  a->met();
  if (C* w = dynamic_cast<C*>(a)) w->inna();
}

void drukujZle(A* a)
{
  a->met();
  if (C* w = static_cast<C*>(a)) w->inna();
}

B b; C c;
drukuj(&c); // metC 3.14 innaC
drukuj(&b); // metB
drukujZle(&c); // metC 3.14 innaC
drukujZle(&b); // metB 3.83854e-308 innaC
```

Wielodziedziczenie

- Wielodziedziczenie – tworzenie klasy pochodnej wyprowadzanej z więcej niż jednej bezpośredniej klasy podstawowej (*dziedziczenie wielobazowe*, *dziedziczenie wielorakie*)
- Kolejność wywoływania konstruktorów/destruktorów

```
class A
{
public:
  A() { cout << "ctorA "; }
  ~A() { cout << "dtorA "; }
};

class B
{
public:
  B() { cout << "ctorB "; }
  ~B() { cout << "dtorB "; }
};
```

```
class C : public B, public A
{
public:
  C() { cout << "ctorC "; }
  ~C() { cout << "dtorC "; }
};

C c; // ctorB ctorA ctorC
     // dtorC dtorA dtorB

A* a = &c;
B* b = &c;
```

Wielodziedziczenie

- Zasięg klasy pochodnej (wielodziedziczącej) – jednoczesne przeszukiwanie wszystkich bezpośrednich klas podstawowych; możliwość wieloznacznej interpretacji nazwy w razie dziedziczenia składowej o tej samej nazwie z więcej niż jednej klasy podstawowej

```
class A
{
public:
  A(int a_) : a(a_) {}
  void drukuj() { cout << a; }
  void metodaA()
  { cout << "metodaA\n"; }
protected:
  int a;
};
```

```
class B
{
public:
  B(double b_) : b(b_) {}
  void drukuj() { cout << b; }
  void metodaB()
  { cout << "metodaB\n"; }
protected:
  double b;
};
```

```
class C : public B, private A
{
public:
  C(int a_, double b_) : B(b_), A(a_) {}
  void drukuj()
  { A::drukuj(); cout << '␣';  B::drukuj(); cout << '\n'; }
  void pokaz()
  { cout << a << '␣' << b << '␣'; metodaA(); }
};

C c(1, 2.2);
c.drukuj(); // 1 2.2
c.pokaz(); // 1 2.2 metodaA
c.metodaB(); //  metodaB
c.metodaA(); // błąd

B* b = &c;
A* a = &c; // błąd
```

Dziedziczenie wirtualne

- Dziedziczenie wirtualne – definiowanie klasy za pośrednictwem dziedziczenia zapewniającego składanie przez referencję (domyślnym mechanizmem dziedziczenia jest składanie przez wartość); klasa dziedzicząca wirtualnie otrzymuje tylko jeden wspólny podobiekt klasy podstawowej niezależnie od liczby wystąpień tej klasy w hierarchii dziedziczenia; obiekt wirtualnej klasy podstawowej – wspólny podobiekt klasy podstawowej
- Inicjowanie podobiektu klasy wirtualnej odbywa się w kostruktorach najbardziej pochodnej klasy

```
class A
{
public:
  A(int a_) : a(a_) {}
  void metodaA() { cout << "metodaA\n"; }
protected:
  int a;
};
```

Dziedziczenie wirtualne

```
class B : public A
{
public:
  B(int a, double b_) : A(a), b(b_) {}
  void pokaz() { cout << a << ’␣’ << b << ’\n’; }
  void metodaB() { cout << "metodaB\n"; }
protected:
  double b;
};

class C : public A
{
public:
  C(int a, float c_) : A(a), c(c_) {}
  void pokaz() { cout << a << ’␣’ << c << ’\n’; }
  void metodaC() { cout << "metodaC\n"; }
protected:
  float c;
};
```

Dziedziczenie wirtualne

```
class D : public B, public C
{
public:
  D(int a1, double b, int a2, float c)
    : B(a1, b), C(a2, c) {}
  void pokaz() { B::pokaz(); C::pokaz(); }
};

D d(1, 2.2, 3, 4.4);
d.pokaz(); // 1 2.2
           // 3 4.4
d.metodaB(); // metodaB
d.metodaC(); // metodaC
d.metodaA(); // błąd d.B::metodaA();
```

Dziedziczenie wirtualne

```
class A { /* bez zmian */ };

class B : public virtual A { /* bez zmian */ };

class C : public virtual A { /* bez zmian */ };

class D : public B, public C
{
public:
  D(int a1, double b, int a2, float c)
    : A(10), B(a1, b), C(a2, c) {}
  /* bez zmian */
};
```

```
D d(1, 2.2, 3, 4.4);
d.pokaz(); // 10 2.2
           // 10 4.4
```

```
d.metodaB(); // metodaB
d.metodaC(); // metodaC
d.metodaA(); // metodaA
```

Dziedziczenie wirtualne

```
class A
{
public:
  A() { cout << "ctorA "; }
  ~A() { cout << "dtorA "; }
};

class B : public A
{
public:
  B() { cout << "ctorB "; }
  ~B() { cout << "dtorB "; }
};
```

```
class C : public A
{
public:
  C() { cout << "ctorC "; }
  ~C() { cout << "dtorC "; }
};

class D : public B, public C
{
public:
  D() { cout << "ctorD "; }
  ~D() { cout << "dtorD "; }
};
```

```
D d; // ctorA ctorB ctorA ctorC ctorD
     // dtorD dtorC dtorA dtorB dtorA
```

Dziedziczenie wirtualne

```
class A
{
public:
  A() { cout << "ctorA "; }
  ~A() { cout << "dtorA "; }
};

class B : public virtual A
{
public:
  B() { cout << "ctorB "; }
  ~B() { cout << "dtorB "; }
};
```

```
class C : public virtual A
{
public:
  C() { cout << "ctorC "; }
  ~C() { cout << "dtorC "; }
};

class D : public B, public C
{
public:
  D() { cout << "ctorD "; }
  ~D() { cout << "dtorD "; }
};
```

```
D d; // ctorA ctorB ctorC ctorD
     // dtorD dtorC dtorB dtorA
```

Obsługa sytuacji wyjątkowych

- Zgłaszanie wyjątku będącego obiektem klasy – ograniczenia na rodzaj klas, których można użyć do tworzenia obiektów wyjątków;
  - ▷ posiada odpowiedni dostępny konstruktor do utworzenia obiektu wyjątku
  - ▷ posiada dostępny konstruktor kopiujący
  - ▷ posiada dostępny destruktor
  - ▷ klasa nie jest klasą abstrakcyjną
- Etapy zgłoszenie wyjątku będącego obiektem klasy
  1. wyrażenie `throw` tworzy obiekt tymczasowy – wywołanie odpowiedniego konstruktora klasy
  2. utworzenie kopii obiektu tymczasowego – (wywołanie konstruktora kopiującego klasy) utworzenie obiektu reprezentującego obiekt wyjątku w celu przekazania go do procedury obsługi
  3. usunięcie obiektu tymczasowego – (wywołanie destruktora) wykonywane przed przystąpieniem do szukania procedury obsługi

Obsługa sytuacji wyjątkowych

```
class A
{
public:
  A() : a_(++c) { cerr << "ctorA:" << a_ << ' '; }
  A(const A& aa) : a_(++c) { cerr << "cctorA:" << a_ << ' '; }
  ~A() { cerr << "dtorA:" << a_ << ' '; }
  int a() const { return a_; }
protected:
  int a_;
  static int c;
};

int A::c = 0;

void fun1a() { throw A(); }

try { fun1a(); }
catch (const A& a) { cerr << "wyjątekA:" << a.a() << ' '; }

// ctorA:1 cctorA:2 dtorA:1 wyjątekA:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Wyjątek należący do klasy pochodnej może być obsłużony przez klauzulę `catch` przeznaczoną dla wyjątków klasy podstawowej

```
class B : public A
{
public:
  B() { cerr << "ctorB:" << a_ << ' '; }
  B(const B& bb) : A(bb) { cerr << "cctorB:" << a_ << ' '; }
  ~B() { cerr << "dtorB:" << a_ << ' '; }
};

void fun1b() { throw B(); }

try { fun1b(); }
catch (const A& a) { cerr << "wyjątekA:" << a.a() << ' '; }
catch (const B& b) { cerr << "wyjątekB:" << b.a() << ' '; }

// ctorA:1 ctorB:1 cctorA:2 cctorB:2 dtorB:1 dtorA:1
//    wyjątekA:2 dtorB:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Wybór klauzuli obsługi wyjątku według zasady *pierwszego dopasowania* – obsługi dokona pierwsza napotkana klauzula mogąca obsłużyć wyjątek, a nie najlepiej dopasowana; najpierw klauzula dla klasy pochodnej

```
try { fun1b(); }
catch (const B& b) { cerr << "wyjątekB:" << b.a() << '␣'; }
catch (const A& a) { cerr << "wyjątekA:" << a.a() << '␣'; }

// ctorA:1 ctorB:1 cctorA:2 cctorB:2 dtorB:1 dtorA:1
//   wyjątekB:2 dtorB:2 dtorA:2
```

- Podczas tworzenia obiektu wyjątku nie bada się aktualnego typu obiektu

```
void fun2() { B b; A* a = &b; throw *a; }

try { fun2(); }
catch (const B& b) { cerr << "wyjątekB:" << b.a() << '␣'; }
catch (const A& a) { cerr << "wyjątekA:" << a.a() << '␣'; }

// ctorA:1 ctorB:1 cctorA:2 dtorB:1 dtorA:1 wyjątekA:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Ponowne zgłoszenie wyjątku powoduje zgłoszenie pierwotnego obiektu reprezentującego wyjątek

```
void fun3()
{
  try { fun1b(); }
  catch (const A& a) { cerr << "wyjątekA:" << a.a() << '␣'; throw; }
}

try { fun3(); }
catch (const B& b) { cerr << "wyjątekB:" << b.a() << '␣'; }
catch (const A& a) { cerr << "wyjątekA:" << a.a() << '␣'; }

// ctorA:1 ctorB:1 cctorA:2 cctorB:2 dtorB:1 dtorA:1
//   wyjątekA:2 wyjątekB:2 dtorB:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Wykorzystanie metod wirtualnych obiektów reprezentujących wyjątki

```
class A
{
  // ...
public:
  virtual void nocotam() const { cerr << "WyjątekA:" << a_ << ' '; }
};

class B : public A
{
  // ...
public:
  virtual void nocotam() const { cerr << "WyjątekB:" << a_ << ' '; }
};

try { fun1b(); }
catch (const A& a) { a.nocotam(); }

// ctorA:1 ctorB:1 cctorA:2 cctorB:2 dtorB:1 dtorA:1
//   WyjątekB:2 dtorB:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Podczas *zwijania stosu* w poszukiwaniu klauzuli `catch` przed opuszczeniem bieżącego zasięgu (bloku instrukcji, funkcji) automatycznie wywoływane są destruktory obiektów

```
class C
{
public:
  C() : c(++cs) { cerr << "ctorC:" << c << '␣'; }
  ~C() { cerr << "dtorC:" << c << '␣'; }
protected:
  int c;
  static int cs;
};

int C::cs = 0;
```

```
void fun4sub()
{
  C c2;
  fun1a();
  C c3;
}

void fun4()
{
  C c1;
  fun4sub();
  C c4;
}

try { fun4(); }
catch (const A& a) { a.nocotam(); }

// ctorC:1 ctorC:2 ctorA:1 cctorA:2 dtorA:1 dtorC:2 dtorC:1
//   WyjątekA:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Specyfikacja wyjątków dla metody wirtualnej klasy pochodnej musi być taka sama lub zawężona niż dla klasy podstawowej – gwarancja nie naruszenia specyfikacji wyjątków metody klasy bazowej w trakcie wywołania za pośrednictwem wskaźnika (referencji)

```
class Eb
{ public: virtual void met() throw( A ) { fun1a(); } };

class Ep : public Eb
{ public: virtual void met() throw( A ) { fun1b(); } };

Eb* e = new Ep;
try { e->met(); }
catch (const A& a) { a.nocotam(); }
delete e;

// ctorA:1 ctorB:1 cctorA:2 cctorB:2 dtorB:1 dtorA:1
//   WyjątekB:2 dtorB:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Blok `try` funkcji jest niezbędny do ochrony listy inicjowania składowych w konstruktorach klas

```
int fun5(int i)
{
  if (!i) throw A();
  return 2*i;
}

class D
{
public:
  D(int d0)
  try
    : d(fun5(d0))
  { fun1b(); }
  catch (const A& a)
  { cerr << "ctorD-"; a.nocotam(); }
  D(int d0, int)
    : d(fun5(d0))
  {
    try { fun1b(); }
    catch (const A& a)
    { cerr << "ctorD-"; a.nocotam(); }
  }
protected:
  int d;
};
```

Obsługa sytuacji wyjątkowych

```
try { D d(1,0); }
catch (const A& a) { cerr << "main()-"; a.nocotam(); }
// ctorA:1 ctorB:1 cctorA:2 cctorB:2 dtorB:1 dtorA:1
//   ctorD-WyjątekB:2 dtorB:2 dtorA:2

try { D d(1); }
catch (const A& a) { cerr << "main()-"; a.nocotam(); }
// ctorA:1 ctorB:1 cctorA:2 cctorB:2 dtorB:1 dtorA:1
//   ctorD-WyjątekB:2 main()-WyjątekB:2 dtorB:2 dtorA:2

try { D d(0,0); }
catch (const A& a) { cerr << "main()-"; a.nocotam(); }
// ctorA:1 cctorA:2 dtorA:1 main()-WyjątekA:2 dtorA:2

try { D d(0); }
catch (const A& a) { cerr << "main()-"; a.nocotam(); }
// ctorA:1 cctorA:2 dtorA:1
//   ctorD-WyjątekA:2 main()-WyjątekA:2 dtorA:2
```

Obsługa sytuacji wyjątkowych

- Hierarchia klasy wyjątków w bibliotece standardowej C++; pliki nagłówkowe `<exception>, <stdexcept>`
  - ▷ klasa bazowa `exception`, klasy pochodne `logic_error, runtime_error, bad_cast, bad_alloc`
  - ▷ błędy logiczne: klasa bazowa `logic_error`, klasy pochodne `invalid_argument, out_of_range, length_error, domain_error`
  - ▷ błędy wykonania: klasa bazowa `runtime_error`, klasy pochodne `range_error, overflow_error, underflow_error`

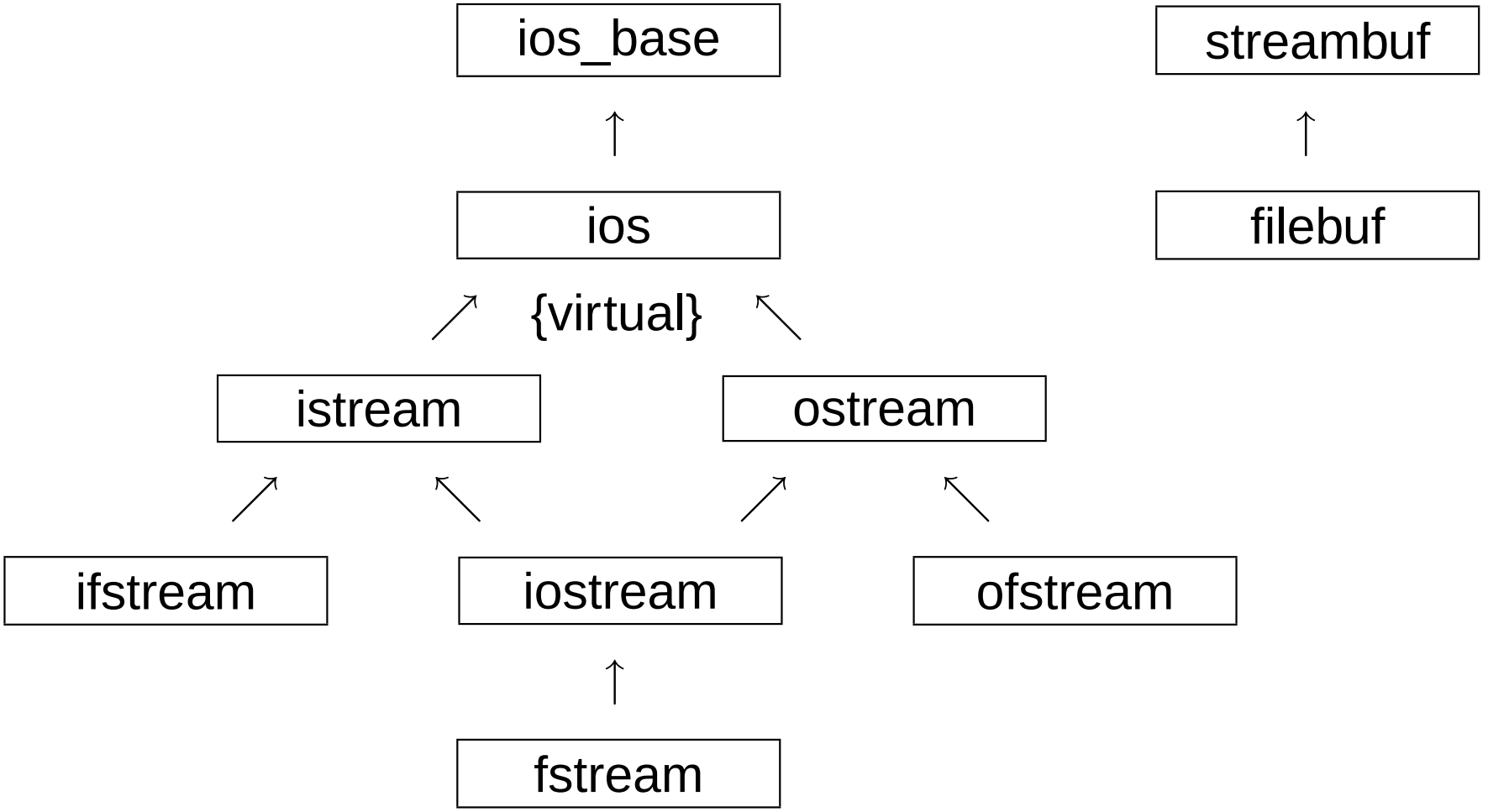
```
class exception
{
public:
  exception() throw() { }
  virtual ~exception() throw();
  virtual const char* what() const throw();
};
```

Obsługa sytuacji wyjątkowych

```
class tab
{
public:
  explicit tab(int val = 0)
  { for( int i = 0; i < 3; ++i) tab_[i] = val; }
  int& operator[] (int i)
  {
    if (i < 0 || i > 2)
      throw out_of_range("zły␣indeks␣tab");
    return tab_[i];
  }
protected:
  int tab_[3];
};

try { tab t(3); t[1] = t[4]; }
catch (const exception& e) { cerr << e.what(); }
// zły indeks tab
```

Biblioteka wejścia – wyjścia

- Obsługa wejścia – wyjścia nie jest częścią języka C++, lecz biblioteki standardowej
- Uproszczona hierarchia dziedziczenia klas biblioteki `IOStream`



Biblioteka wejścia – wyjścia

- `ios_base` – klasa bazowa określająca składowe wszystkich klas strumieniowych niezależnych od typu znakowego; znaczniki stanu oraz formatu strumienia
- `ios` – `basic_ios<char>` – definiuje składowe zależne od typu znakowego; definicja bufora strumienia `streambuf` (`basic_streambuf<char>`)
- `istream` – `basic_istream<char>` – klasa strumienia wejścia
- `ostream` – `basic_ostream<char>` – klasa strumienia wyjścia
- `iostream` – `basic_iostream<char>` – klasa strumienia wejścia-wyjścia
- `ifstream` – `basic_ifstream<char>` – klasa strumienia pliku wejścia
- `ofstream` – `basic_ofstream<char>` – klasa strumienia pliku wyjścia
- `fstream` – `basic_fstream<char>` – klasa strumienia pliku wejścia-wyjścia

Biblioteka wejścia – wyjścia

- Predefiniowane obiekty
  - ▷ `cin` – obiekt klasy `istream`; standardowe wejście; pobieranie danych z terminala użytkownika
  - ▷ `cout` – obiekt klasy `ostream`; standardowe wyjście; odsyłanie danych do terminala użytkownika
  - ▷ `cerr` – obiekt klasy `ostream`; standardowe wyjście diagnostyczne; odsyłanie komunikatów o błędach (domyślnie na standardowe wyjście)
  - ▷ `clog` – obiekt klasy `ostream`; standardowe wyjście logowań; odsyłanie komunikatów o stanie programu (domyślnie na standardowe wyjście)
- Operatory pobierania i odsyłania do strumienia – przeciążone operatory przesunięć bitowych
  - ▷ `>>` – pobieranie ze strumienia (przesunięcie w prawo) – *pobieracz*
  - ▷ `<<` – odesłanie do strumienia (przesunięcie w lewo) – *wstawiacz*

Biblioteka wejścia – wyjścia

- Wyjście formatowane

```
int ii = 1024, jj = 2048, *iw = &ii;
const char* ala = "Ala";

cout << ii << '␣'   // 1024
     << *iw << '␣'  // 1024
     << iw << '␣'   // 0xbffff888
     << &ii << '␣'  // 0xbffff888
     << &iw << '␣'  // 0xbffff880
     << (ii > jj ? ii : jj) << '␣' // 2048
     << ala << '␣'  // Ala
     << &ala << '\n'; // 0xbffff87c
```

Biblioteka wejścia – wyjścia

- Formatowanie strumienia wyjściowego – podstawowe manipulatory
  - ▷ `dec, hex, oct` – podstawa zapisu liczb

```
int i = 16;
cout << hex << i; // 10
cout << oct << i; // 20
cout << dec << i; // 16
```

  - ▷ `showbase, noshowbase` – dodawanie przedrostka oznaczającego podstawę zapisu liczb

```
cout << showbase;
cout << hex << i; // 0x10
cout << oct << i; // 020
```

  - ▷ `showpos, noshowpos` – dopisywanie znaku + do liczb nieujemnych

```
double d = 3.14, b = -10;
cout << showpos;
cout << d << '␣' << b; // +3.1400 -10.0000
```

Biblioteka wejścia – wyjścia

- `showpoint, noshowpoint` – kropka dziesiętna we wszystkich liczbach zmiennopozycyjnych

```
cout << showpoint;
cout << d << ' ' << b; // 3.1400 -10.0000
cout << noshowpoint;
cout << d << ' ' << b; // 3.14 -10
```

- `fixed, scientific` – dziesiętny i wykładniczy zapis liczb zmiennopozycyjnych

```
cout << scientific;
cout << d << ' ' << b; // 3.1400e+00 -1.0000e+01
```

- `boolalpha, noboolalpha` – reprezentacja wartości prawda i fałsz jako napisów albo jako liczby 1 i 0

```
bool f = false;
cout << f << ' ' << boolalpha << f; // 0 false
```

Biblioteka wejścia – wyjścia

- `uppercase, nouppercase` – napis `0X` dla liczb szesnastkowych, `E` dla notacji wykładniczej
- `left, right, internal` – dodanie znaków wypełniających na prawo od wartości, na lewo od wartości, między znakiem a wartością
- `setw(w)` – szerokość zapisu wartości lub pobieranych wartości

```
double d = -10;
cout << setw(5) << left << d;     // -10
cout << setw(5) << right << d;    //   -10
cout << setw(5) << internal << d; // -  10
```

- `setfill(z)` – wypełnienie odstępów znakiem `z`

```
cout << setfill('%');
cout << setw(5) << left << d;     // -10%%
cout << setw(5) << right << d;    // %%-10
cout << setw(5) << internal << d; // -%%10
```

Biblioteka wejścia – wyjścia

- `setprecision(n)` – ustawienie dokładności n dla zapisu zmiennopozycyjnego

```
cout << M_PI; // 3.14159
cout << setprecision(12)
     << M_PI; // 3.14159265359
```

- `flush, ends, endl` – opróżnianie bufora wyjściowego, dodanie znaku pustego i opróżnianie, dodanie znaku nowego wiersza i opróżnianie

- metody składowe klasy `ios` – `setf(), unsetf(), width(), fill(), precision()`

```
cout.precision(5);
cout.width(10);
cout.fill('$');
cout.setf(ios::right);
cout << M_PI << endl; // $$$$3.1416
```

Biblioteka wejścia – wyjścia

- Wejście formatowane

```
int i = 16;
double d = -10;
cin >> i >> d; // 2 3.1
cout << i << d; // 2 3.1

int suma = 0;
while (cin >> i)  // 1 2 3
  suma += i;
cout << suma;   // 6

cin >> i ;  // 23.45
cout << i << static_cast<bool>(cin); // 23 1
cin >> d;
cout << d << static_cast<bool>(cin) // 0.45 1
```

Biblioteka wejścia – wyjścia

```
string txt;
cin >> txt;   // ala makota
cout << txt;  // ala
cin >> setw(5) >> txt;
cout << txt;  // makot

int i = 0, j = 0;
cin >> hex >> i >> oct >> j; // 10 20
cout << i << '␣' << j;       // 16 16
```

- Nieformatowane operacje wejścia/wyjścia
  - ▷ Metody klasy `istream` – `get(), getline(), read(), ignore(), gconut(), peek(), putback()`
  - ▷ Metody klasy `ostream` – `put(), write()`

Biblioteka wejścia – wyjścia

- Stan strumienia
  - ▷ Badanie stanu strumienia
    - ⋆ Metoda `eof()` – prawda, jeśli napotkano znacznik końca pliku
    - ⋆ Metoda `bad()` – prawda, jeśli wykryto próbę wykonania niedozwolonej operacji; nieokreślony stan “zepsucia” strumienia, np. próba przesunięcia poza znacznik końca pliku
    - ⋆ Metoda `fail()` – prawda, jeśli nie udało się otworzyć strumienia lub napotkano nieprawidłowy format
    - ⋆ Metoda `good()` – prawda, jeśli wszystkie powyższe metody zwracają fałsz
    - ⋆ Metoda `rdstate()` – stan strumienia
- Ustawianie stanu strumienia
  - ▷ Metoda `setstate()` – ustawienie stan jednego lub kilku warunków stanu strumienia
  - ▷ Metoda `clear()` – przywrócenie dobrego stanu strumienia

Biblioteka wejścia – wyjścia

```
cin.setstate(ios::badbit);
ios::iostate stan = cin.rdstate();
cout << stan;          // 1
cin.setstate(ios::failbit);
cout << cin.rdstate(); // 5
cin.clear();
cout << cin.rdstate(); // 0
cin.clear(stan);
cout << cin.rdstate(); // 1
```

- Stan strumienia i sytuacje wyjątkowe – możliwość generowania wyjątku zależnego od stanu strumienia; mechanizm domyślnie wyłączony; metoda składowa `exceptions()`; generowany wyjątek typu `ios::failure`

```
cin.exceptions(ios::failbit);
int i;
try { while (cin >> i) cout << i; }
catch (ios::failure) { cerr << "błąd strumienia"; }
```

Biblioteka wejścia – wyjścia

```
int sumator(istream& str)
{
  ios::iostate st = str.exceptions();
  str.exceptions(ios::failbit|ios::badbit);
  int v, sum = 0;
  try {
    while (str >> v)
      sum += v;
  }
  catch (...) {
    if (!str.eof()) { str.exceptions(st); throw; }
  }
  str.exceptions(st);
  return sum;
}

try { cout << sumator(cin); }
catch (ios::failure) {cerr << "błąd␣strumienia"; }
```

Biblioteka wejścia – wyjścia

- Operator odbierania ze strumienia >>

```
class A
{
public:
  A() : a(0), b(0) {}
protected:
  int a;
  double b;
friend
  istream& operator >> (istream& is, A& ob)
  { return is >> ob.a >> ob.b; }
friend
  ostream& operator << (ostream& os, const A& ob)
  { return os << ob.a << '␣' << ob.b;  }
};

A o;
cin >> o;  // 23.45
cout << o; // 23 0.45
```

Biblioteka wejścia – wyjścia

- Strumienie plikowe – plik nagłówkowy `<fstream>`
  - ▷ plikowy strumień wyjściowy – obiekt klasy `ofstream`; domyślnie otwierany w trybie wyjściowym (`ios::out`, znacznik ustawiony na początek pliku); możliwość otwarcia w trybie dołączającym (`ios::app`, znacznik ustawiony na koniec pliku); zamykanie strumienia metodą składową klasy lub automatycznie przez destruktor
  - ▷ plikowy strumień wejściowy – obiekt klasy `ifstream`; domyślnie otwierany w trybie do odczytu (`ios::in`); znacznik ustawiony na początku pliku
  - ▷ plikowy strumień wejścia–wyjścia – obiekt klasy `fstream`; wymagane jawne określenie trybu pracy; znacznik ustawiany odpowiednio do trybu pracy strumienia

Biblioteka wejścia – wyjścia

```
int i = 10;
double d = 3.14;
ofstream plik;
plik.open("nazwa_pliku"); // ios::out
if (!plik) { // nie udało się otworzyć pliku
  // ...
}
plik << i << '\t' << d << '\n';
// ...
plik.close();
// ...
ofstream inny("inna_nazwa");
// ...
plik.open("nazwa_pliku", ios::app);
plik << ++i << '\t' << (2*d) << '\n';
```

Biblioteka wejścia – wyjścia

```
int i;
double d;
ifstream plik("nazwa_pliku");
if (!plik) { // nie udało się otworzyć pliku
  // ...
}
plik >> i >> d;
// ...

//---===###===---

// plik do odczytu
fstream plik("nazwa_pliku", ios::in);
// plik do pisania
fstream inny("inna_nazwa", ios::out);
// plik do odczytu i do dopisywania
fstream nowy("nowa_nazwa", ios::in|ios::app);
```

Biblioteka wejścia – wyjścia

- tryby otwarcia strumieni plikowych
  - ▷ `ios::in` – odczyt
  - ▷ `ios::out` – zapis, znacznik na początku, istniejący plik przycinany
  - ▷ `ios::app` – zapis, znacznik na końcu pliku, istniejący plik nie przycinany
  - ▷ `ios::ate` – odczyt/zapis, znacznik na końcu pliku
  - ▷ `ios::trunc` – przycinanie istniejącego pliku, jego zawartość zostanie utracona, domyślnie ustawiany dla trybu `ios::out`
  - ▷ `ios::binary` – tryb binarny, domyślnie tekstowy
  - ▷ `ios::nocreate` – zapis, nowy plik nie zostanie utworzony (nie standardowy)
  - ▷ `ios::noreplace` – zapis, istniejący plik nie zostanie przycięty (nie standardowy)

Biblioteka wejścia – wyjścia

- bieżące położenia znacznika
  - ▷ `tellg()` – pozycja pobierania z pliku
  - ▷ `tellp()` – pozycja wstawiania do pliku
- zmiana położenia znacznika – zmiana bieżącej pozycji pliku względem początku pliku lub względem określonej pozycji w pliku
  - ▷ `seekg()` – ustalenie pozycji pobierania z pliku
  - ▷ `seekp()` – ustalenie pozycji wstawiania do pliku
- predefiniowane pozycje w pliku
  - ▷ `ios::beg` – początek pliku
  - ▷ `ios::cur` – bieżąca pozycja w pliku
  - ▷ `ios::end` – koniec pliku

Biblioteka wejścia – wyjścia

```
string s;
ifstream f("plik.txt");     // abc def ghi jkl
f >> s;  cout << s << '\n';          // abc
streampos z = f.tellg();
f >> s;  cout << s << '\n';          // def
f.seekg(z);
f >> s;  cout << s << '\n';          // def
f.seekg(-2, ios::cur);
f >> s;  cout << s << '\n';          // ef
f.seekg(2, ios::cur);
f >> s;  cout << s << '\n';          // hi
f.seekg(-4, ios::end);
f >> s;  cout << s << '\n';          // jkl
f.seekg(1);
f >> s;  cout << s << '\n';          // jkl
f.clear();
f >> s;  cout << s << '\n';          // bc
```

Biblioteka wejścia – wyjścia

- Strumienie wewnętrzne – możliwość wykonywania operacji charakterystycznych dla strumieni na napisach (obiektach klasy `string`) przechowywanych w pamięci wewnętrznej; plik nagłówkowy `<sstream>`
  - ▷ obiekt klasy `ostringstream` – `basic_ostringstream<char>` – odsyłanie znaków do napisu
  - ▷ obiekt klasy `istringstream` – `basic_istringstream<char>` – pobieranie znaków z napisu
  - ▷ obiekt klasy `stringstream` – `basic_stringstream<char>` – pobieranie/odsyłanie znaków z/do napisu

```
ostringstream os;
int psy = 3;
os << "Ala␣ma␣" << 1 << "␣kota";
cout << os.str() << endl;    // Ala ma 1 kota
os << ",␣a␣Boguś␣ma␣" << psy << "␣psy" << endl;
cout << os.str();            // Ala ma 1 kota, a Boguś ma 3 psy
```

Biblioteka wejścia – wyjścia

```
string Suffix(string name, unsigned i)
{
  ostringstream tmp;
  tmp << '.' << setw(3) << setfill('0') << i;

  string::size_type p = name.find('.');
  if (p != string::npos)
    name.insert(p, tmp.str());
  else
    name += tmp.str();

  return name;
}

cout << Suffix("wyniki.dat", 4) << endl;   // wyniki.004.dat
cout << Suffix("dane", 11) << endl;        // dane.011
```

Biblioteka wejścia – wyjścia

```
  const char* nps = "Włodek␣ma␣14␣rybek";
  istringstream is(nps);
  string s[3];
  int ile;
  is >> s[0] >> s[1] >> ile >> s[2];
  for (int i = 0; i < 3; ++i)
    cout << s[i] << '␣';
  cout << ile << endl;  // Włodek ma rybek 14

// ---===###===---

  string pi = "3.141592653589793238␣46";
  double d = 1.1, b = 1.2;
  istringstream iss(pi);
  cout << d << '␣' << b << '␣' << iss.rdstate();  // 1.1 1.2 0
  iss >> d >> b;
  cout << d << '␣' << b << '␣' << iss.rdstate();  // 3.14159 1.2 3
```