Sprawy organizacyjne

- rejestracja,
- godziny wykładu,
- co umiemy,
- co będzie na wykładach,
- zasady zaliczania,
- literatura,
- historia języka C++,
- wykład.

Co umiemy

- Programowałem w C++,
- Programowałem w języku obiektowym ,
- Programowałem w C,
- Programowałem w innym języku,
- Nigdy nie programowałem,

Co będzie na wykładach

- elementy C w C++ (2-3),
- programowanie obiektowe w C++ (klasy, dziedziczenie, metody wirtualne, szablony, ...),

- Smalltalk, Delphi, Java, C# (1 jeśli starczy czasu),
- … .

Czemu C++?

- bardzo popularny,
- dostępny w wielu implementacjach,
- łatwo dostępna literatura.

Zastrzeżenia

- C++ nie jest jedynym językiem obiektowym,
- C++ nie jest najlepszym językiem obiektowym,
- w C++ można pisać programy nie mające nic wspólnego z programowaniem obiektowym,
- C++ jest trudnym językiem, z bardzo rozbudowaną składnią i subtelną semantyką.

Czego **nie** będzie na wykładach

- Opisu poszczególnych implementacji C++,
- Opisu poszczególnych nakładek na język C++.

Zasady zaliczania

Zaliczenie ćwiczeń:

- obecności,
- klasówka,
- program zaliczeniowy.

Egzamin na koniec wykładu:

- trzeba będzie napisać fragment programu w C++,
- 1.5 - 2 godz.

Wstęp do programowania obiektowego

Programowanie obiektowe

- Wszyscy o nim mówią, wszyscy go używają i nikt nie wie co to jest.
- Podejście obiektowe swym znaczeniem wykracza daleko poza programowanie (ogólnie: opis skomplikowanych systemów).
- Podejście obiektowe to inny sposób widzenia świata:
- Agenci do których wysyła się komunikaty, zobowiązani do realizacji pewnych zadań, realizujący je wg pewnych metod postępowania. Te metody są ukryte przed zlecającym wykonanie zadania. Przykład: poczta.
- Programowanie obiektowe jest czymś więcej niż jedynie dodaniem do języka programowania kilku nowych cech, jest innym sposobem myślenia o procesie podziału problemów na podproblemy i tworzeniu programów.
- Na programowanie obiektowe można patrzeć jako na symulowanie rozważanego świata.
- W programowaniu obiektowym mamy do czynienia z zupełnie innym modelem obliczeń,

zamiast komórek pamięci mamy obiekty (agentów), komunikaty i zobowiązania.

- Obiekt ma swój własny stan i swoje własne zachowanie (operacje). Każdy obiekt jest egzemplarzem pewnej klasy. Zachowanie obiektu jest określone w klasie tego obiektu. Z każdym obiektem jest związany zbiór zobowiązań (responsibilities) - protokół.
- Zachowanie obiektu można zaobserwować wysyłając do niego komunikat. W odpowiedzi obiekt wykona swoją metodę, związaną z tym komunikatem. To jakie akcje zostaną wykonane zależy od obiektu - obiekt innej klasy może wykonać w odpowiedzi na ten sam komunikat zupełnie inne akcje (polimorfizm).
- Przesłanie komunikatu jest podobne do wywołania procedury, istotna różnica między nimi polega na tym, że to jakie akcje zostaną wykonane po przesłaniu komunikatu zależy od tego, do kogo ten komunikat został wysłany. Wiązanie nazwy komunikatu z realizującą go metodą odbywa się dopiero podczas wykonywania, a nie podczas kompilowania, programu (metody wirtualne, wczesne i późne wiązanie metod).

- Programowanie obiektowe polega także na tym, że staramy się co tylko się da zrzucić na innych (na agentów), a nie staramy się wszystkiego robić samemu (nawyk z programowania imperatywnego). Umożliwia to ponowne wykorzystywanie (reuse) oprogramowania. „Ważnym pierwszym krokiem w ponownym wykorzystywaniu komponentów jest chęć spróbowania zrobienia tego.”
- Programowanie obiektowe nie ma większej mocy wyrażania, ale ułatwia rozwiązywanie problemów w sposób właściwy dla tworzenia dużych systemów.
- Programowanie obiektowe stanowi następny etap ewolucji mechanizmów abstrakcji w programowaniu:
  - procedury,
  - bloki,
  - moduły,
  - ATD,
  - programowanie obiektowe.
- W kontekście programowania obiektowego mówimy o projektowaniu sterowanym zobowiązaniami (RDD Responsibility-Driven Design). Przerzucanie zobowiązań na innych wiąże się z dawaniem im większej

samodzielności, dzięki czemu komponenty oprogramowania stają się mniej od siebie zależne, co z kolei ułatwia ich ponowne wykorzystywanie.

- Podsumowanie własności programowania obiektowego (Alan Kay, 1993):
  - Wszystko jest obiektem.
  - Obliczenie realizują obiekty przesyłając między sobą komunikaty.
  - Obiekt ma swoją pamięć zawierającą obiekty.
  - Każdy obiekt jest egzemplarzem klasy.
  - Klasa stanowi repozytorium zachowania obiektu.
  - Klasy są zorganizowane w hierarchię dziedziczenia.

Dziedziczenie

- Jedna z fundamentalnych własności podejścia obiektowego.
- Klasy obiektów można kojarzyć w hierarchie klas (prowadzi to do drzew lub DAGów dziedziczenia). Dane i zachowanie związane z klasami z wyższych poziomów tej hierarchii są dostępne w klasach po nich dziedziczących (pośrednio lub bezpośrednio).

- Mówimy o nadklasach (klasach bazowych) i podklasach (klasach pochodnych).
- W czystej postaci dziedziczenie odzwierciedla relację *is-a* (*jest*). Bardzo często ta relacja jest mylona z relacją *has-a* (*ma*) dotyczącą składania.
- Czasami chcemy wyrazić w hierarchii klas wyjątki (pingwin), można to uzyskać dzięki przedefiniowywaniu metod (method overriding).
- Zasada podstawialności: zawsze powinno być możliwe podstawienie obiektów podklas w miejsce obiektów nadklas.
- Możliwe zastosowania dziedziczenia:
  - specjalizacja (Kwadrat < Prostokąt).
  - specyfikacja (klasy abstrakcyjne).
  - rozszerzanie (Kolorowe Okno < Czarno-białe Okno).
  - Ograniczanie (Należy unikać zastępując składaniem, Kolejka < Lista).
- Zalety programowania obiektowego
  - Fakt, że w podejściu obiektowym każdy obiekt jest całkowicie odpowiedzialny za swoje zachowanie, powoduje że tworzone zgodnie z tym podejściem oprogramowanie w naturalny sposób jest podzielone na (w dużym stopniu) niezależne od siebie komponenty. Jest niezwykle ważna zaleta, gdyż takie

komponenty można projektować, implementować, testować, modyfikować i opisywać niezależnie od reszty systemu.
    - Dzięki temu, że oprogramowanie obiektowe składa się z wielu (w dużym stopniu) niezależnych od siebie komponentów, łatwo jest te komponenty ponownie wykorzystywać (reusability).
    - Tworzenie oprogramowania w metaforze porozumiewających się między sobą agentów skłania do bardziej abstrakcyjnego myślenia o programie: w kategoriach agentów, ich zobowiązań i przesyłanych między nimi komunikatów, z pominięciem tego jak są realizowane obsługujące te komunikaty metody.
- Zalety dziedziczenia:
    - Dziedziczenie pozwala pogodzić ze sobą dwie sprzeczne tendencje w tworzeniu oprogramowania:
        - chcemy żeby stworzone systemy były zamknięte,
        - chcemy żeby stworzone systemy były otwarte.

- Możliwość ponownego wykorzystywania (nie trzeba od nowa pisać odziedziczonych metod i deklaracji odziedziczonych zmiennych).
- Ponowne wykorzystywanie zwiększa niezawodność (szybciej wykrywa się błędy w częściej używanych fragmentach programów).
- Ponowne wykorzystywanie pozwala szybciej tworzyć nowe systemy (budować je z klocków).
- Zgodność interfejsów (gdy wiele klas dziedziczy po wspólnym przodku).
- Ukrywanie informacji (użytkownika klasy interesuje tylko interfejs należących do niej obiektów (komunikaty i ich znaczenie), a nie zawartość tych obiektów (metody i dane).

Narzuty związane z programowaniem obiektowym

- Szybkość wykonywania (programowanie obiektowe zachęca do tworzenia uniwersalnych narzędzi, rzadko kiedy takie narzędzia są równie efektywne, co narzędzia stworzone do jednego, konkretnego zastosowania).

- Rozmiar programów (programowanie obiektowe zachęca do korzystania z bibliotek gotowych komponentów, korzystanie z takich bibliotek może zwiększać rozmiar programów).
- Narzut związany z przekazywaniem komunikatów (wiązanie komunikatu z metodą odbywa się dopiero podczas wykonywania programu).
- Problem jo-jo (nadużywanie dziedziczenia może uczynić czytanie programu bardzo żmudnym procesem).

Podstawy języka C++

Literatura:

O języku C++:

- Język C++; Bjarne Stroustrup; WNT 2000; wyd. 5te i zmienione, 976 str.
- Język C++, ćwiczenia i rozwiązania; David Vandevoorde; WNT 2001, 287 str.
- The C++ Standard: Incorporating Technical Corrigendum No. 1, British Standards Institute 782 pages, 2nd ed.,19 December 2003, 782 str.
- Podstawy języka C++; Stanley B. Lippman, Josee Lajoie; WNT 2001, 1206 str.
- Podstawy języka C++, ćwiczenia i rozwiązania; Clovis L. Tondo; WNT 2001, 431 str.
- Istota języka C++, zwięzły opis; Stanley B. Lippman, WNT 2004, 326 str.
- C++ Biblioteka standardowa, podręcznik programisty; Nicolai M. Josuttis; Helion 2003, 726 str.
- Projektowanie i rozwój języka C++, Bjarne Stroustrup, WNT 1996, 512 str.
- Internet (draft proposed International Standard for Information Systems – Programming

language C++, X3J16/96-0225, 2 XII 1996; 699 stron w tym 370 język, 329 biblioteki).

O programowaniu obiektowym:

- Programowanie obiektowe, Peter Coad, Jill Nicola, Read Me 1993,
- Metody Obiektowe w teorii i praktyce; Ian Graham; WNT 2004, 908 str.

Historia C++

- **Algol 60** (13-to osobowy zespół, 1960-63),
- **Simula** 67 (Ole-Johan Dahl, Bjorn Myhrhaug, Kristen Nygaard, Norweski Ośrodek Obliczeniowy w Oslo, 1967),
- **C** (Dennis M. Ritchie, Bell Laboratories , New Jersey, 1972),
- **C** z klasami (Bjarne Stroustrup, Bell Laboratories, New Jersey, 1979-80),
- **C++** (j.w., 1983),
- Komisja X3J16 powołana do standaryzacji C++ przez ANSI (ANSI C++) (1990-??),

Elementy C w C++

- Uwaga: to nie jest opis języka C!
- C++ jest językiem ze statycznie sprawdzaną zgodnością typów,
- Program w C++ może się składać z wielu plików (zwykle pełnią one rolę modułów).

Notacja

- elementy języka (słowa kluczowe, separatory) zapisano są pismem prostym, pogrubionym (np. **{)**
- elementy opisujące konstrukcje języka zapisano pismem pochyłym, bez pogrubienia (np. *wyrażenie*),
- jeżeli dana konstrukcja może w danym miejscu wystąpić lub nie, to po jej nazwie jest napis $_{opc}$ (umieszczony jako indeks),
- jeżeli dana konstrukcja może w danym miejscu wystąpić 0 lub więcej razy, to po jej nazwie jest napis $_{0}$ (umieszczony jako indeks),
- jeżeli dana konstrukcja może wystąpić w danym miejscu raz lub więcej razy, to po jej nazwie jest napis $_{1}$ (umieszczony jako indeks),

Instrukcje języka C++

instrukcja:
instrukcja_etykietowana
instrukcja_wyrażeniowa
blok
instrukcja_warunkowa
instrukcja_wyboru
instrukcja_pętli
instrukcja_deklaracyjna
instrukcja_próbuj
instrukcja_skoku

- instrukcja etykietowana

instrukcja_etykietowana:
identyfikator **:** *instrukcja*
case *stałe_wyrażenie* **:** *instrukcja*
default : *instrukcja*

- pierwszy rodzaj instrukcji etykietowanej dotyczy instrukcji goto i nie będzie tu omawiany,
- instrukcje etykietowane case i default mogą wystąpić jedynie wewnątrz instrukcji wyboru,
- wyrażenie stałe stojące po etykiecie case musi być typu całkowitego.

- instrukcja wyrażeniowa

  instrukcja_wyrażeniowa:
  $wyrażenie_{opc}$ **;**

  - efektem jej działania są efekty uboczne wyliczania wartości wyrażenia (sama wartość po jej wyliczeniu jest ignorowana),
  - zwykle instrukcjami wyrażeniowymi są przypisania i wywołania funkcji, np.:

```
i = 23*k +1;
wypisz_dane(Pracownik);
```

  - szczególnym przypadkiem jest instrukcja pusta:
    ;
    użyteczna np. do zapisania pustej treści instrukcji pętli.

- instrukcja złożona (blok):

  blok:
  { $instrukcja_0$ **}**

- służy do zgrupowania wielu instrukcji w jedną,
- nie ma żadnych separatorów oddzielających poszczególne instrukcje,
- deklaracja też jest instrukcją,
- instrukcja złożona wyznacza zasięg widoczności.

• Instrukcja warunkowa

instrukcja_warunkowa:
if (*warunek* **)** *instrukcja*
if (*warunek* **)** *instrukcja* **else** *instrukcja*
warunek:
wyrażenie

- wyrażenie musi być typu arytmetycznego lub wskaźnikowego,
- jeśli wartość wyrażenia jest różna od zera, to wyrażenie jest uważane za prawdziwe, wpp. za fałszywe,
- dyndający else dotyczy ostatnio spotkanego if bez else,
- instrukcja nie może być deklaracją (ale może zawierać deklaracje),

- uwaga: w nowym projekcie języka przewidziane są zmiany:
  warunek może być także deklaracją (z pewnymi ograniczeniami), której zasięgiem jest dana instrukcja,
  ta deklaracja musi zawierać część inicjalizującą,
  instrukcja może być deklaracją (jej zasięgiem zawsze jest tylko ta instrukcja),
  wartość warunku jest niejawnie przekształcana na (nowo dodany) typ bool.

- Instrukcja wyboru

instrukcja_wyboru:
switch (*warunek* **)** *instrukcja*

- powoduje przekazanie sterowania do jednej z podinstrukcji występujących w instrukcji, o etykiecie wyznaczonej przez wartość warunku.
- warunek musi być typu całkowitego,
- podinstrukcje (wewnątrz instrukcji) mogą być etykietowane jedną (kilkoma) etykietami przypadków:
  case *wyrażenie_stałe* **:**

- wszystkie stałe przypadków muszą mieć różne wartości,
- może wystąpić (co najwyżej jedna) etykieta:
  default :

  Nie musi być ostatnią etykietą, bo i tak zawsze najpierw są analizowane etykiety case.
- podczas wykonywania instrukcji switch oblicza się wartość warunku, a następnie porównuje się ją ze wszystkimi stałymi występującymi po case. Jeśli któraś z nich równa się wartości warunku, to sterowanie jest przekazywane do instrukcji poprzedzonej etykietą z tą wartością, wpp. jeśli jest etykieta default, do instr. poprzedzonej tą etykietą, wpp. sterowanie przechodzi bezpośrednio za instr. switch. Ponieważ sterowanie przekracza etykiety case i default, prawie zawsze trzeba jawnie kończyć wykonywanie obsługi przypadku instrukcją break!
- instrukcja nie może być deklaracją (ale może zawierać deklaracje),
- uwaga: w nowym projekcie języka przewidziane są zmiany:
  - warunek może być także deklaracją (z pewnymi ograniczeniami), której zasięgiem jest dana instrukcja,

- ta deklaracja musi zawierać część inicjalizującą,
- instrukcja może być deklaracją, (jej zasięgiem zawsze jest tylko ta instrukcja),
- wartość wyrażenia jest niejawnie przekształcana na typ całkowity lub wyliczeniowy.

- instrukcja pętli

instrukcja_pętli:
while (*warunek* **)** *instrukcja*
do *instrukcja* **while (** *wyrażenie* **)**
for (*inst_ini_for* $warunek_{opc}$ **;** $wyrażenie_{opc}$ **)** *instrukcja*

inst_ini_for:
instrukcja_wyrażeniowa
prosta_deklaracja

Instrukcja while:
- warunek musi być typu arytmetycznego lub wskaźnikowego,
- jeśli wartość warunku jest różna od zera, to warunek uważa się za prawdziwe, wpp. za fałszywy,

- dopóki wartość warunku jest różna od zera, wykonujemy instrukcję. Warunek wylicza się przed każdym wykonaniem instrukcji.

Instrukcja do:

- powtarzamy wykonywanie instrukcji aż wyrażenie będzie miało wartość 0. Wyrażenie wylicza się po każdym wykonaniu instrukcji.
- wyrażenie musi być typu arytmetycznego lub wskaźnikowego.

Instrukcja for:

- jest równoważna ciągowi instrukcji:

```
{
 inst_ini_for
 while ( warunek ) {
   instrukcja
   wyrażenie ;
 }
}
```

(różnica: jeśli w instrukcji wystąpi continue, to wyrażenie będzie obliczone przed obliczeniem warunku.)
warunek jak zwykle,

- pominięcie warunku jest traktowane jako wpisanie 1,

- jeśli instrukcja instr_inic jest deklaracją, to zasięg zadeklarowanych nazw sięga do końca bloku
- uwaga: w nowym projekcie języka przewidziane są zmiany:
  - warunek może być także deklaracją (z pewnymi ograniczeniami), której zasięgiem jest dana instrukcja.
  - ta deklaracja musi zawierać część inicjalizującą. Zauważmy, że w pętli do nie ma warunku, jest tylko wyrażenie.
  - w pętli for zasięg nazw zadeklarowanych w inst_ini_for jest taki sam, jak zasięg nazw zadeklarowanych w warunku,
  - pominięcie warunku w pętli for jest równoważne wpisaniu tam true,
  - instrukcja może być deklaracją (jej zasięgiem zawsze jest tylko ta instrukcja). Przy każdym obrocie pętli sterowanie wchodzi i opuszcza ten lokalny zasięg.
  - wartość warunku jest niejawnie przekształcana na (nowo dodany) typ bool. Dotyczy to także wyrażenia występującego w pętli do.

- instrukcje skoku

instrukcja_skoku:
break ;
continue ;
return *wyrażenie*$_{opc}$ **;**
goto *identyfikator* **;**

Przy wychodzeniu z zakresu następuje niszczenie obiektów automatycznych w kolejności odwrotnej do ich deklaracji.

Instrukcja break:
- może się pojawić jedynie wewnątrz pętli lub instrukcji wyboru i powoduje przerwanie najciaśniej ją otaczającej takiej instrukcji,
- sterowanie przechodzi bezpośrednio za przerwaną instrukcję.

Instrukcja continue:
- może się pojawić jedynie wewnątrz instrukcji pętli i powoduje zakończenie bieżącego obrotu (najciaśniej otaczającej) pętli.

Instrukcja return:

- służy do kończenia wykonywania funkcji i (ewentualnie) do przekazywania wartości wyniku funkcji,

Instrukcja goto:

- Instrukcja deklaracji

instrukcja_deklaracji:
blok_deklaracji

- wprowadza do bloku nowy identyfikator,
- ten identyfikator może przesłonić jakiś identyfikator z bloku zewnętrznego (do końca tego bloku),
- inicjowanie zmiennych (auto i register) odbywa się przy każdym wykonaniu ich instrukcji deklaracji. Zmienne te giną przy wychodzeniu z bloku,

Komentarze

W C++ mamy dwa rodzaje komentarzy:

- Komentarze jednowierszowe zaczynające się od //.

- Komentarze (być może) wielowierszowe, zaczynające się od /* i kończące */. Te komentarze nie mogą się zagnieżdżać.

Stałe

- Stałe całkowite:
  - dziesiętne (123543). Ich typem jest pierwszy z wymienionych typów, w którym dają się reprezentować: int, long int, unsigned long int (czyli nigdy nie są typu unsigned int!).
  - szesnastkowe (0x3f, 0x4A). Ich typem jest pierwszy z wymienionych typów, w którym dają się reprezentować: int, unsigned int, long int, unsigned long int.
  - ósemkowe (0773). Typ j.w.
  - przyrostki U, u, L i l do jawnego zapisywania stałych bez znaku i stałych long, przy czym znów jest wybierany najmniejszy typ (zgodny z przyrostkiem), w którym dana wartość się mieści.
- Stałe zmiennopozycyjne:
  - mają typ double (o ile nie zmienia tego przyrostek)
  - - 1.23, 12.223e3, -35E-11,
  - - przyrostek f, F (float), l, L (long double),

- stałe znakowe (typu char)
  - znak umieszczony w apostrofach (‘a’)
  - niektóre znaki są opisane sekwencjami dwu znaków zaczynającymi się od \. Takich sekwencji jest 13, oto niektóre z nich:
    - \n (nowy wiersz),
    - \\ (lewy ukośnik),
    - \’ (apostrof),
    - \ooo (znak o ósemkowym kodzie ooo, można podać od jednej do trzech cyfr ósemkowych),
    - \xhhh (znak o szesnastkowym kodzie hhh, można podać jedną lub więcej cyfr szesnastkowych),
  - Każda z tych sekwencji opisuje pojedynczy znak!
- stałe napisowe (typu const char[n])
  - ciąg znaków ujęty w cudzysłów (”ala\n”),
  - zakończony znakiem ‘\0’,
  - musi się zawierać w jednym wierszu, ale
  - sąsiednie stałe napisowe (nawet z różnych wierszy) są łączone,
- stałe logiczne (typu bool):
  - true,
  - false.

- Stała 0 jest typu int, ale można jej używać jako stałej (oznaczającej pusty wskaźnik) dowolnego typu wskaźnikowego,

Identyfikatory

- Identyfikator (nazwa) to ciąg liter i cyfr zaczynający się od litery (_ traktujemy jako literę),
- rozróżnia się duże i małe litery,
- długość nazwy nie jest ograniczona przez C++ (może być ograniczona przez implementację),
- słowo kluczowe C++ nie może być nazwą,
- nazw zaczynających się od _ i dużej litery, bądź zawierających __ (podwójne podkreślenie) nie należy definiować samemu (są zarezerwowane dla implementacji i standardowych bibliotek),
- nazwa to maksymalny ciąg liter i cyfr.

Deklaracje

- Każdy identyfikator musi być najpierw zadeklarowany,
- Deklaracja określa typ, może też określać wartość początkową,
- Zwykle deklaracja jest też definicją (przydziela pamięć zmiennej, definiuje treść funkcji),
- Deklarując nazwę w C++ można podać specyfikator klasy pamięci:

- **auto**
  prawie nigdy nie stosowany
- **register**
  tyle co auto, z dodatkowym wskazaniem. dla kompilatora, że deklarowana zmienna będzie często używana,
- **static**
- **extern**

Typy

- Typ określa rozmiar pamięci, dozwolone operacje i ich znaczenie,
- z każdą nazwą w C++ związany jest typ,
- typy można nazywać,
- operacje dozwolone na nazwach typów:
  - podawanie typu innych nazw,
  - sizeof
  - new
  - specyfikowanie jawnych konwersji

Typy podstawowe

- Liczby całkowite:
  - char
  - signed char

- short int (signed short int)
- int (signed int)
- long int (signed long int)

- Liczby całkowite bez znaku:
  - unsigned char
  - unsigned short int
  - unsigned int
  - unsigned long int

  (część int można opuścić)

- liczby rzeczywiste:
  - float
  - double
  - long double
- w C++ sizeof(char) wynosi 1 (z definicji),
- typ wartości logicznych bool (niedawno wprowadzony)

- char może być typem ze znakiem lub bez znaku,
- C++ gwarantuje, że
  - 1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
  - sizeof(float) <= sizeof(double) <= sizeof(long double)
  - sizeof(T) = sizeof(signed T) = sizeof(unsigned T), dla T = char, short, int lub long

- char ma co najmniej 8, short 16, a long 32 bity

Typy pochodne - wskaźniki

- Wskaźniki są bardzo często używane w C++,
- wskaźnik do typu T deklarujemy (zwykle) jako T*,
- zwn. składnię C++ (wziętą z C) wskaźniki do funkcji i tablic definiuje się mniej wygodnie,
- Operacje na wskaźnikach:
  - przypisanie
  - stała NULL
  - * (operator wyłuskania)
  - p++, p+wyr, p-wyr, p1-p2 gdzie p, p1, p2 to wskaźniki, a wyr to wyrażenie całkowitoliczbowe
- Uwaga na wskaźniki - tu bardzo łatwo o błędy, np.:

```
char *dest = new char[strlen(src+1)];
strcpy(src, dest);
// Błędny fragment programu (ale kompilujący się bez
// ostrzeżeń) zwn złe położenie czerwonego nawiasu.
```

Typy pochodne - tablice

- T[rozmiar] jest tablicą *rozmiar* elementów typu T, indeksowaną od 0 do *rozmiar*-1,
- odwołanie do elementu tablicy wymaga użycia operatora []
- tablice wielowymiarowe deklaruje się wypisując kilka razy po sobie [rozmiar_i] (nie można zapisać tego w jednej parze nawiasów kwadratowych)
- w C++ nazwy tablicy można używać jako wskaźnika. Oznacza ona wskaźnik do pierwszego elementu tablicy. Przekazywanie tablicy jako parametru oznacza przekazanie adresu pierwszego elementu.
- nie można przypisywać tablic,
- właściwie nie ma tablic:

  a[i] oznacza *(a+i) co z kolei oznacza i[a].

  Ale uwaga na różnicę:

  int *p;

  oznacza coś zupełnie innego niż

  int p[100];

Typy pochodne - struktury

- Struktura to zestaw elementów dowolnych typów (przynajmniej w tej części wykładu),
- struktury zapisujemy następująco:

struct <nazwa> {
typ_1 pole_1;
typ_2 pole_2;
…
typ_k pole_k;
};

- do pól struktury odwołujemy się za pomocą
  - . jeśli mamy strukturę,
  - -> jeśli mamy wskaźnik do struktury,
- struktura może być wynikiem funkcji, parametrem funkcji i można na nią przypisywać,
- nie jest natomiast zdefiniowane porównywanie struktur (== i !=)
- można definiować struktury wzajemnie odwołujące się do siebie. Używa się do tego deklaracji:

  struct <nazwa>;

  Tak wstępnie zadeklarowanej struktury można używać tyko tam, gdzie nie jest wymagana znajomość rozmiaru struktury,

Typy pochodne - referencje

- Referencja (alias) to inna nazwa już istniejącego obiektu,

- typ referencyjny zapisujemy jako T&, gdzie T jest jakimś typem (T nie może być typem referencyjnym),
- referencja **musi** być zainicjalizowana i nie można jej zmienić,
- **wszelkie** operacje na referencji (poza inicjalizacją) dotyczą obiektu na który wskazuje referencja, a nie samej referencji!
- referencje są szczególnie przydatne dla parametrów funkcji (przekazywanie przez zmienną),

Definiowanie nazwy typu

- Deklaracja **typedef** służy do nazywania typu. Składniowo ma ona postać zwykłej deklaracji poprzedzonej słowem kluczowym **typedef,**
- dwa niezależnie zadeklarowane typy są różne, nawet jeśli mają identyczną strukturę, typedef pozwala ominąć tę niedogodność,
- typedef służy do zadeklarowania identyfikatora, którego można potem używać tak jak gdyby był nazwą typu.

Wyliczenia

- można definiować wyliczenia np.:
  enum kolor{ czerwony, zielony }

Stałe nazwane

Do deklaracji dowolnego obiektu można dodać słowo kluczowe **const**, dzięki czemu uzyskujemy deklarację stałej a nie zmiennej (oczywiście taka deklaracja musi zawierać inicjalizację),

- można używać **const** przy deklarowaniu wskaźników:
  - const char *p = „ala”;
    wskaźnik do stałego napisu
  - char const*p = „ala”;
    stały wskaźnik do napisu
  - const char const *p = „ala”;
    stały wskaźnik do stałego napisu

Inicjalizowanie

- Deklarując zmienne można im nadawać wartości początkowe:
  - struct S {int a; char* b;};
    S s = {1, „Urszula”};
  - int x[] = {1, 2, 3};
  - float y[4] [3] = {
    { 1, 3, 5},
    { 2, 4, 6},
    {3, 5, 7}
    }

Funkcje

- deklaracja funkcji ma następującą postać (w pewnym uproszczeniu):

  *typ_wyniku nazwa **(** lista par. **)***

  instrukcja_złożona

  - jako typ wyniku można podać void, co oznacza, że funkcja nie przekazuje wyniku,
  - lista parametrów to ciąg (oddzielonych przecinkami) deklaracji parametrów, postaci (w uproszczeniu):

    typ nazwa
- parametry są zawsze przekazywane przez wartość (ale mogą być referencjami, lub wskaźnikami)
- jeśli parametrem jest wskaźnik, to jako argument można przekazać adres obiektu, używając operatora &

Wartości domyślne parametrów

Deklarując parametr funkcji (lub metody), można po jego deklaracji dopisać znak = i wyrażenie. Deklaruje się w ten sposób domyślną wartość argumentu odpowiadającego temu parametrowi. Pozwala to wywoływać tak zadeklarowaną funkcję zarówno z tym argumentem jak i bez niego. W tym

drugim przypadku, przy każdym wywołaniu podane wyrażenie będzie wyliczane, a uzyskana w ten sposób wartość będzie traktowana jako brakujący argument:

```
char* DajTablicę(unsigned rozmiar = 10)
{
   return new char[rozmiar];
}

char* p = DajTablicę(100);  // Tablica 100-elementowa
char* q = DajTablicę();        // Tablica 10-elementowa
```

Można w jednej funkcji zadeklarować kilka parametrów o wartościach domyślnych, ale muszą to być ostatnie parametry. Oznacza to, że jeśli zadeklarujemy wartość domyślną dla jednego parametru, to wówczas dla wszystkich następnych parametrów również należy określić domyślne wartości (w tej lub jednej z poprzednich deklaracji funkcji):

```
void f(int, float, int = 3);
void f(int, float=2, int);
void f(int a=1, float b, int c)
// Oczywiście można było od razu napisać:
// void f(int a=1, float b=2, int c=3)
{
   cout << endl << a << " " << b << " " << c;
```

```
}
// Wszystkie poniższe wywołania odnoszą się do tej
samej funkcji f.
f(-1,-2,-3);
f(-1,-2);
f(-1);
f();
```

Nie można ponownie zdefiniować argumentu domyślnego w dalszej deklaracji (nawet z tą samą wartością).

Przykład zastosowania dodefiniowywania wartości domyślnych poza definicją funkcji: funkcja z innego modułu używana w danym module z domyślną wartością (np. sqrt(double = 2).

Uwagi techniczne:

Wiązanie nazw i kontrola typów wyrażenia określającego wartość domyślną odbywa się w punkcie deklaracji, zaś wartościowanie w każdym punkcie wywołania:

```
// Przykład z książki Stroustrupa
int a = 1;
int f(int);
int g(int x = f(a));  // argument domyślny: f(::a)

void h() {
a = 2;
{
  int a = 3;
  g();              // g(f::a), czyli g(f(2)) !
```

```
    }
}
```

Wyrażenia określające wartości domyślne:

- nie mogą zawierać zmiennych lokalnych (to naturalne, chcemy w prosty sposób zapewnić, że na pewno w każdym wywołaniu da się obliczyć domyślną wartość argumentu),
- nie mogą używać parametrów formalnych funkcji (bo te wyrażenia wylicza się przed wejściem do funkcji, a porządek wartościowania argumentów funkcji nie jest ustalony (zależy od implementacji)). Wcześniej zadeklarowane parametry formalne są w zasięgu i mogą przesłonić np. nazwy globalne.
- Argument domyślny nie stanowi części specyfikacji typu funkcji, zatem funkcja z jednym parametrem, który jednocześnie ma podaną wartość domyślną, może być wywołana z jednym argumentem lub bez argumentu, ale jej typem jest (tylko) funkcja jednoargumentowa (bezargumentowa już nie).
- Operator przeciążony nie może mieć argumentów domyślnych.

Zarządzanie pamięcią

- operator new

  new *nazwa_typu*

  lub

  new *nazwa_typu **[** wyrażenie **]***

  - daje 0, gdy nie uda się przydzielić pamięci,
  - uwaga: w nowej wersji języka przewidziano możliwość zgłaszania wyjątku w przypadku niemożności przydzielenia pamięci.

- operator delete

  delete *wskaźnik*

  lub

  delete[] *wskaźnik*

- operator sizeof

  sizeof wyr

  tego wyrażenia się nie wylicza, wartością jest rozmiar wartości wyr.

  sizeof (typ)

  rozmiar typu typ,

  sizeof(char) = 1,

  wynik jest stałą typu size_t zależnego od implementacji,

Jawna konwersja typu

Operatory

- *, /, %
- +, -
- <<, >>
- <, >, <=, >=,
- ==, !=
- &&
- ||
- ? :
- =, +=, /=. %=, +=. -=

Preprocesor

- #include <...>
- #include ”...”

Program

- Program składa się z jednostek translacji. Jednostka translacji to pojedynczy plik źródłowy (po uwzględnieniu dyrektyw preprocesora: #include oraz tych dotyczących warunkowej kompilacji).

- Jednostki translacji składające się na jeden program nie muszą być kompilowane w tym samym czasie.
- program składa się z
  - deklaracji globalnych (zmienne, stałe, typy)
  - definicji funkcji

  Wśród funkcji musi się znajdować funkcja main(). Jej typem wyniku jest int. Obie poniższe definicje funkcji main są dopuszczalne (i żadne inne):
  - int main(){ /* ... */ },
  - int main(int argc, char* argv[]){ /* ... */ }.

Klasy jako struktury

Klasy jako struktury

- Klasa jest nowym typem danych zdefiniowanym przez użytkownika.
- Wartości tak zdefiniowanego typu nazywamy obiektami.
- Najprostsza klasa jest po prostu strukturą (rekordem), czyli paczką kilku różnych zmiennych.
- Składnia deklaracji klasy:

specyfikator_klasy:
nagłówek_klasy { specyfikacja_składowych $_{opt}$ }
nagłówek klasy:
słowo_kluczowe_klasy identyfikator$_{opt}$
klauzula_klas_bazowych$_{opt}$
słowo_kluczowe_klasy
specyfikator_zagnieżdżonej_nazwy identyfikator$_{opt}$
klauzula_klas_bazowych$_{opt}$

Przykład: liczby zespolone
Bez klas byłoby tak:

```
struct Zespolona
{
double re, im;
};
```

Dokładnie to samo można wyrazić używając klas:

```
  class Zespolona
{
    public:
    double re, im;
};
```

Ale taka definicja nie wystarcza, potrzebujemy operacji na tym typie danych. Możemy je zdefiniować tak:

```
Zespolona dodaj(Zespolona z1, Zespolona z2)
{
  Zespolona wyn;
  wyn.re = z1.re + z2.re;
  wyn.im = z1.im + z2.im;
  return wyn;
}
```

Ma to jednak wadę, trudno się zorientować czym tak na prawdę jest typ Zespolona (trzeba przeczytać cały program, żeby znaleźć wszystkie definicje dotyczące liczb zespolonych).

Klasy jako struktury z operacjami

W C++ możemy powiązać definicję typu danych z dozwolonymi na tym typie operacjami:

```
class Zespolona
{
  public:
  double re, im;

  Zespolona dodaj(Zespolona);
  Zespolona odejmij(Zespolona);
  double modul();
};
```

Zauważmy, że:

- zwiększyła się czytelność programu: od razu widać wszystkie operacje dostępne na naszym typie danych,
- zmieniłaj się liczba parametrów operacji,
- nie podaliśmy ani treści operacji, ani nazw parametrów.

To co podaliśmy powyżej jest **specyfikacją interfejsu** typu Zespolona. Oczywiście trzeba też określić implementację (gdzieś dalej w programie):

```
Zespolona Zespolona::dodaj(Zespolona z)
{
  Zespolona wyn;
  wyn.re = re + z.re;
  wyn.im = im + z.im;
  return wyn;
}
Zespolona Zespolona::odejmij(Zespolona z)
{
  Zespolona wyn;
  wyn.re = re - z.re;
  wyn.im = im - z.im;
  return wyn;
}
double Zespolona::modul()
{return sqrt(re*re + im*im);}
```

Klasy potrafią chronić swoje dane

Przy poprzedniej definicji klasy Zespolona, można było pisać następujące instrukcje:

```
Zespolona z;
double mod;
.....
mod = sqrt(z.re*z.im+z.im*z.im);  // Błąd !!!
```

Nie znamy na razie metody zmuszającej użytkownika do korzystania tylko z dostarczonych przez nas operacji. To bardzo źle, bo:

- upada poprzednio postawiona teza, że wszystkie operacje na typie danych są zdefiniowane tylko w jednym miejscu,
- użytkownik pisząc swoje operacje może (tak jak powyżej) napisać je źle,
- projektując klasę, zwykle nie chcemy, żeby użytkownik mógł bez naszej wiedzy modyfikować jej zawartość (przykład ułamek: nie chcielibyśmy, żeby ktoś wpisał nam nagle mianownik równy zero),
- program użytkownika odwołujący się do wewnętrznej reprezentacji klasy niepotrzebnie

się od niej uzależnia (np. pola re nie możemy teraz nazwać czesc_rzeczywista).

Na szczęście w C++ możemy temu bardzo łatwo zaradzić. Każda składowa klasy (zmienna lub metoda) może być:

- prywatna (private:),
- publiczna, czyli ogólnodostępna (public:).

Domyślnie wszystkie składowe klasy są prywatne, zaś wszystkie składowe struktury publiczne.

Zatem teraz mamy następującą deklarację:

```
class Zespolona
{
    private:  // tu można pominąć private:
    double re, im;
  public:
    Zespolona dodaj(Zespolona);
    Zespolona odejmij(Zespolona);
    double modul();
};
```

Teraz zapis:

```
mod = sqrt(z.re*z.re+z.im*z.im);  // Błąd składniowy (choć wzór poprawny)
```

jest niepoprawny. Użytkownik może natomiast napisać:

```
mod = z.modul();
```

Czy chcemy mieć niezainicjalizowane obiekty?

Oczywiście nie:

```
{
  Zespolona z1;
  cout << z1.modul();  // Wypisze się coś bez sensu
}
```

Jak temu zaradzić? Można dodać metodę ini(), która będzie inicjalizować liczbę, ale ... to nic nie daje. Dalej nie ma możliwości zagwarantowania, że zmienna typu Zespolona będzie zainicjalizowana przed pierwszym jej użyciem. Na szczęście w C++ możemy temu skutecznie zaradzić. Rozwiązaniem są konstruktory. Konstruktor jest specjalną metodą klasy. Ma taką samą nazwę jak klasa. Nie można mu określić typu wyniku konstruktora. Nie można przekazać z niego wyniku instrukcją return. Można w nim wywoływać funkcje składowe klasy. Można go wywołać jedynie przy tworzeniu nowego obiektu danej klasy. W klasie można zdefiniować wiele konstruktorów. Konstruktor może mieć (nie musi)

parametry. Konstruktor jest odpowiedzialny za dwie rzeczy:

- zapewnienie, że obiekt będzie miał przydzieloną pamięć (to jest sprawa kompilatora),
- inicjalizację obiektu (to nasze zadanie, realizuje je treść konstruktora).

Wyróżniamy kilka rodzajów konstruktorów:

- konstruktor bezargumentowy:
  - można go wywołać bez argumentów,
  - jest konieczny, jeśli chcemy mieć tablice obiektów tej klasy.
- konstruktor domyślny:
  - jeśli nie zdefiniujemy żadnego konstruktora, to kompilator sam wygeneruje konstruktor domyślny (bezargumentowy), niczego nie inicjalizujący, poza tymi składowymi, które mają konstruktory domyślne.
- konstruktor kopiujący:
  - można go wywołać z jednym argumentem tej samej klasy, przekazywanym przez referencję,
  - jeśli żadnego takiego konstruktora nie zdefiniujemy, to kompilator wygeneruje go automatycznie. Uwaga: automatycznie wygenerowany konstruktor kopiujący kopiuje obiekt składowa po składowej, więc zwykle

się nie nadaje dla obiektów zawierających wskaźniki !!!

- jest wywoływany niejawnie przy przekazywaniu parametrów do funkcji i przy przekazywaniu wyników funkcji !!!

Teraz deklaracja naszej klasy wygląda następująco:

```
class Zespolona
{
    private:  // tu można pominąć private:
    double re, im;
  public:
    // konstruktory
    Zespolona(double, double);
    // operacje
    Zespolona dodaj(Zespolona);
    Zespolona odejmij(Zespolona);
    double modul();
};

Zespolona::Zespolona(double r, double i)
{
  re = r;
  im = i;
}

// ... reszta definicji
```

Jakie są tego konsekwencje?

```
Zespolona z;      // Błąd! Nie ma konstruktora domyślnego
Zespolona z(3,2);      // OK, taki konstruktor jest
zdefiniowany.
```

Zatem nie można teraz utworzyć niezainicjalizowanego obiektu klasy Zespolona!

Każde użycie konstruktora powoduje powstanie nowego obiektu. Można w ten sposób tworzyć obiekty tymczasowe:

```
double policz_cos(Zespolona z)
 {
  // .....
 }
```

Mogę tę funkcję wywołać tak:

```
Zespolona z(3,4);
policz_cos(z);
```

ale jeśli zmienna z nie jest mi potrzebna, to mogę wywołać tę funkcję także tak:

```
policz_cos( Zespolona(3,4) );
```

Utworzony w ten sposób obiekt tymczasowy będzie istniał tylko podczas wykonywania tej jednej instrukcji.

Dla klasy Zespolona nie musimy definiować konstruktora kopiującego (ten wygenerowany automatycznie przez kompilator zupełnie nam w tym przypadku wystarczy). Gdybyśmy jednak chcieli, to musielibyśmy zrobić to następująco:

```
class Zespolona
{
  private:  // tu można pominąć private:
     double re, im;
  public:
     // konstruktory
     Zespolona(double, double);
     Zespolona(Zespolona&);
     // operacje
     Zespolona dodaj(Zespolona);
     Zespolona odejmij(Zespolona);
     double modul();
};

Zespolona::Zespolona(Zespolona& z)
{
  re = z.re;
  im = z.im;
}
```

Ułatwianie sobie życia

Jest zupełnie naturalne, by chcieć definiować liczby zespolone, które są tak na prawdę liczbami rzeczywistymi. Możemy to teraz robić następująco:

```
Zespolona z(8,0);
```

Gdybyśmy mieli często używać takich liczb, to wygodniej by było mieć konstruktor, który sam dopisuje zero:

```
class Zespolona
{
// ...
  public:
     // konstruktory
     Zespolona(double);
  // ...
};

Zespolona::Zespolona(double r)
{
  re = r;
  im = 0;
}
```

Jest to zupełnie poprawne. Można definiować wiele konstruktorów, kompilator C++ na podstawie listy argumentów zdecyduje, którego należy użyć. Możemy to jednak zapisać prościej korzystając z parametrów domyślnych:

```
class Zespolona
{
    private:  // tu można pominąć private:
    double re, im;
  public:
    // konstruktory
    Zespolona(double, double = 0);
    Zespolona(Zespolona&);
    // operacje
    Zespolona dodaj(Zespolona);
    Zespolona odejmij(Zespolona);
    double modul();
};

Zespolona::Zespolona(double r, double i)
{
  re = r;
  im = i;
}

// .....
```

Uwaga: nie można deklarować wartości domyślnej i w nagłówku funkcji i w jej implementacji.

To że mamy konstruktor liczb zespolonych z jednym argumentem (liczbą typu double) ma dalsze konsekwencje. Poniższe wywołanie jest teraz poprawne:

```
policz_cos( 6 );
```

Innymi słowy zdefiniowanie w klasie K konstruktora, którego można wywołać z jednym parametrem typu T, oznacza zdefiniowanie konwersji z typu T do typu K. O tym jak definiować konwersje w drugą stronę powiemy później.

Zwalnianie zasobów

Gdy obiekt kończy swoje istnienie automatycznie zwalnia się zajmowana przez niego pamięć. Nie dotyczy to jednak zasobów, które obiekt sam sobie przydzielił w czasie swego istnienia. Rozwiązaniem tego problemu są destruktory.
Destruktor to metoda klasy. Klasa może mieć co najwyżej jeden destruktor. Destruktor nie ma parametrów. Nie można specyfikować typu wyniku destruktora. Nie można w nim używać instrukcji return z parametrem. Nazwa destruktora jest taka sama jak nazwa klasy, tyle że poprzedzona tyldą. Destruktor jest odpowiedzialny za dwie rzeczy:

- zwolnienie pamięci zajmowanej przez obiekt (to sprawa kompilatora),
- zwolnienie zasobów (to nasze zadanie, zwalnianie zasobów zapisujemy jako treść destruktora).

Zasobami, które obiekty przydzielają sobie najczęściej są fragmenty pamięci.

W klasie Zespolona destruktor nie jest potrzebny, ale możemy go zdefiniować:

```
class Zespolona
{
    private:  // tu można pominąć private:
    double re, im;
  public:
    // konstruktory i destruktory
    Zespolona(double, double = 0);
    Zespolona(Zespolona&);
    ~Zespolona();
    // operacje
    Zespolona dodaj(Zespolona);
    Zespolona odejmij(Zespolona);
    double modul();
};

Zespolona::~Zespolona()
{
  // W tej klasie nie mamy żadnych zasobów do zwolnienia,
więc pusty
}
```

Przykład definicji klasy:

Zadeklaruj i zaimplementuj klasę implementującą zbiór liczb całkowitych.

Dziedziczenie i hierarchie klas

Wprowadzenie

- Dziedziczenie jest jednym z najistotniejszych elementów obiektowości.
- Dziedziczenie umożliwia pogodzenie dwóch sprzecznych dążeń:
  - raz napisany, uruchomiony i przetestowany program powinien zostać w niezmienionej postaci.
  - programy wymagają stałego dostosowywania do zmieniających się wymagań użytkownika, sprzętowych itp.
- Dziedziczenie umożliwia tworzenie hierarchii klas.
- Klasy odpowiadają pojęciom występującym w świecie modelowanym przez program. Hierarchie klas pozwalają tworzyć hierarchie pojęć, wyrażając w ten sposób zależności między pojęciami.
- Klasa pochodna (podklasa) dziedziczy po klasie bazowej (nadklasie). Klasę pochodną tworzymy wówczas, gdy chcemy opisać bardziej wyspecjalizowane obiekty klasy bazowej.

Oznacza to, że każdy obiekt klasy pochodnej jest obiektem klasy bazowej.

- Zalety dziedziczenia:
  - Jawne wyrażanie zależności między klasami (pojęciami). Np. możemy jawnie zapisać, że każdy kwadrat jest prostokątem, zamiast tworzyć dwa opisy różnych klas.
  - Unikanie ponownego pisania tych samych fragmentów programu (ang. reuse)

Jak definiujemy podklasy

Przykładowa klasa bazowa:

```
class A {
   private:
      int skl1;
   protected:
      int skl2;
   public:
      int skl3;
};
```

Słowo **protected**, występujące w przykładzie, oznacza że składowe klasy po nim wymienione, są widoczne w podklasach (bezpośrednich i dalszych), nie są natomiast widoczne z zewnątrz.

Przykładowa podklasa:

```
class B: public A {
  private:
    int skl4;
  protected:
    int skl5;
  public:
    int skl6;
void f();
};
```

Przykłady użycia:

```
void B::f() {
  skl1 = 1; // Błąd, składowa niewidoczna
  skl2 = 2; // OK!
  skl3 = 3; // OK
  skl4 = skl5 = skl6 = 4; // OK
}

int main() {
  A a;
  B b;
  int i;
  i = a.skl1; // Błąd, składowa niewidoczna
  i = a.skl2; // Błąd, składowa niewidoczna
  i = a.skl3; // OK
  i = a.skl4; // Błąd, nie ma takiej składowej (to samo dla
skl5 i skl6)
  i = b.skl1; // Błąd, składowa niewidoczna
  i = b.skl2; // Błąd, składowa niewidoczna
  i = b.skl3; // OK!
```

```
    i = b.skl4; // Błąd, składowa niewidoczna
    i = b.skl5; // Błąd, składowa niewidoczna
    i = b.skl6; // OK

    return 0;
}
```

Jak wynika z tego przykładu, w języku C++ nie ma żadnego składniowego wyróżnika klas bazowych - można je definiować tak jak zwykłe klasy. Jednak jeśli chcemy żeby projektowana przez nas klasa była kiedyś klasą bazową, to już w momencie jej deklarowania należy myśleć o dziedziczeniu, odpowiednio ustalając, które składowe mają mieć atrybut public, a które protected.

Podsumowując:

- składowe prywatne (private) są widoczne jedynie w klasie z której pochodzą i w funkcjach/klasach z nią zaprzyjaźnionych,
- składowe chronione (protected) są widoczne w klasie z której pochodzą i w funkcjach/klasach z nią zaprzyjaźnionych oraz w jej klasach pochodnych i funkcjach/klasach z nimi zaprzyjaźnionych,
- składowe publiczne (public) są widoczne wszędzie tam, gdzie jest widoczna sama klasa.

Można powiedzieć, że obiekt klasy pochodnej przypomina kanapkę (czy tort), zawierający warstwy pochodzące ze wszystkich nadklas. W szczególności obiekt b zawiera składową skl1 (choć metody z warstwy B nie mają do tej składowej dostępu). Jest tak dlatego, że każdy obiekt klasy pochodnej (tu B) jest obiektem klasy bazowej (tu A).

W podklasach można deklarować składowe o takiej samej nazwie jak w nadklasach:

```
class C {
 public:
  int a;
  void f();
};

class D: public C {
  public:
   int a;
   void f();
};
```

Kompilator zawsze będzie w stanie rozróżnić, o którą składową chodzi:

```
void C::f() {
```

```
 a = 1;  // Składowa klasy C
}

void D::f() {
 a = 2;  // Składowa klasy D
}

int main () {
 C a;
 D b;
 a.a = 2;          // Składowa klasy C
 b.a = 2;          // Składowa klasy D
 a.f();            // Składowa klasy C
 b.f();            // Składowa klasy D
 return 0;
}
```

Operator zasięgu

Oczywiście stosowanie tych samych nazw w nadklasach i podklasach dla zmiennych obiektowych nie ma sensu (dla metod już ma, p. metody wirtualne). Ale co zrobić, gdy już tak się stanie i chcemy w podklasie odwołać się do składowej z nadklasy? Należy użyć operatora zasięgu (::). Oto inna definicja D::f():

```
void D::f() { a = C::a; }
```

W podobny sposób można w funkcji odwoływać się do przesłoniętych zmiennych globalnych:

```
int i;
void f(int i)
 {
   i   = 3;  // Parametr
   ::i = 5;  // Zmienna globalna
 }
```

Zgodność typów

Obiekt klasy pochodnej jest obiektem klasy bazowej, chcielibyśmy więc, żeby można było wykorzystywać go wszędzie tam, gdzie można używać obiektów z klasy bazowej. Niestety, nie zawsze to jest możliwe (i nie zawsze ma sens):

```
A a, &ar=a, *aw;
B b, &br=b, *bw;

a = b;              // OK, A::operator=
b = a;              // Błąd, co miałoby być wartościami
    // zmiennych obiektowych występujących w B a w A
    // nie?
    // Byłoby poprawne po zdefiniowaniu:
    //   B& B::operator=(A&);
ar = br;            // OK
br = ar;            // Błąd, tak samo jak b = a;
aw = bw;            // OK
bw = aw;            // Błąd, co by miało znaczyć bw->skl6 ?
```

```
fA(a);          // OK, A::A(&A)
fA(b);          // OK, A::A(&A) automatyczny
konstruktor zadziała
fAref(a);       // OK
fAref(b);       // OK
fAwsk(&a);      // OK
fAwsk(&b);      // OK
fB(a);          // Błąd, A ma za mało składowych
fB(b);          // OK
fBref(a);       // Błąd, A ma za mało składowych
fBref(b);       // OK
fBwsk(&a);      // Błąd, A ma za mało składowych
fBwsk(&b);      // OK
```

Na co wskazują wskaźniki?

Zwróćmy uwagę na interesującą konsekwencję reguł zgodności przedstawionych powyżej:

```
D d;
C *cwsk=&d;

cwsk->f();
```

jaka funkcja powinna się wywołać? cwsk pokazuje na obiekt klasy D. Ale kompilator o tym nie wie i wywoła funkcję z klasy C. Powrócimy do tego tematu przy okazji funkcji wirtualnych.

Dziedziczenie public, protected i private

Klasa może dziedziczyć po nadklasie na trzy różne sposoby. Określa to słowo wpisane w deklaracji podklasy przed nazwą klasy bazowej. Decyduje ono o tym kto wie, że klasa pochodna dziedziczy po klasie bazowej. Tym słowem może być:

- public: wszyscy wiedzą
- protected: wie tylko klasa pochodna, funkcje/klasy zaprzyjaźnione z nią oraz jej klasy pochodne i funkcje/klasy zaprzyjaźnione z nimi,
- private: wie tylko klasa pochodna i funkcje/klasy z nią zaprzyjaźnione.
- Co daje ta wiedza? Dwie rzeczy:
- pozwala dokonywać niejawnych konwersji ze wskaźników do podklas na wskaźniki do nadklas,
- pozwala dostawać się (zgodnie z omówionymi poprzednio regułami dostępu) do składowych klasy bazowej.

Jeśli pominiemy to słowo, to domyślnie zostanie przyjęte private (dla struktur public).

Oto przykłady ilustrujące przedstawione reguły:

```
class A {
  public:
    int i;
    // ...
};

class B1: public A {};

class B2: protected A {};

class B3: private A {
  void f(B1*, B2*, B3*);
};

class C2: public B2 {
  void f(B1*, B2*, B3*);
};

void f(B1* pb1, B2* pb2, B3* pb3)
{
  A* pa = pb1;       // OK
  pb1->a = 1;  // OK
  pa = pb2;     // Błąd (f nie wie, że B2 jest podklasą A)
  pb2->a = 1;  // Błąd
  pa = pb3;     // Błąd
  pb3->a = 1;  // Błąd
}
```

```
void C2::f(B1* pb1, B2* pb2, B3* pb3)
{
  A* pa = pb1;       // OK
  pb1->a = 1;  // OK
  pa = pb2;     // OK
  pb2->a = 1;  // OK
  pa = pb3;     // Błąd (C2::f nie wie, że B3 jest podklasą A
  pb3->a = 1;  // Błąd
}
void B3::f(B1* pb1, B2* pb2, B3* pb3)
{
  A* pa = pb1;       // OK
  pb1->a = 1;  // OK
  pa = pb2;     // Błąd (B3::f nie wie, że B2 jest
                // podklasą A
  pb2->a = 1;  // Błąd
  pa = pb3;     // OK
  pb3->a = 1;  // OK
}
```

Zatem jeśli nazwa klasy bazowej jest poprzedzona słowem protected, to składowe publiczne tej klasy zachowują się w klasie pochodnej jak chronione, zaś jeśli nazwa klasy bazowej jest poprzedzona słowem private, to jej składowe publiczne i chronione zachowują się w klasie pochodnej jak prywatne.

Przedstawione tu mechanizmy określania dostępu do klasy podstawowej mają zdecydowanie

mniejsze znaczenie, niż omówione poprzednio mechanizmy ochrony dostępu do składowych.

Metody wirtualne

Rozważmy poniższy (uproszczony) przykład:

```
class Figura {
   // ...
   protected:
      int x,y;      // położenie na ekranie
   public:
      void ustaw(int, int);
      void pokaż();
      void schowaj();
      void przesuń(int, int);
};

Figura::ustaw(int n_x, int n_y, int )
{
   x = n_x;
   y = n_y;
}

Figura::przesuń(int n_x, int n_y)
{
   schowaj();
   ustaw(n_x, n_y);
   pokaż();
}
```

```
Figura::pokaż()
{
                    // Nie umiemy narysować dowolnej figury
}

Figura::schowaj()
{
                    // j.w.
}
```

Teraz definiujemy figurę okrąg:

```
class Okrąg: public Figura {
  protected:
    int promień,
  public:
    Okrag(int, int, int);
    pokaż();
    schowaj();
  //
};
```

i jakoś definiujemy dla niej metody pokaż i schowaj. Możemy teraz napisać:

```
Okrąg o(20, 30, 10);          // Okrąg o zadanym promieniu
o.pokaż();                    // Rysuje okrąg
o.przesuń(100, 200);          // Nie przesuwa !
```

Problem polega oczywiście na tym, że w treści metody przesuń wywołały się metody Figura::pokaż i Figura::ukryj zamiast Okrąg::pokaż i Okrąg::ukryj. Jak temu zaradzić? W tradycyjnym języku programowania jedynym rozwiązaniem byłoby wpisanie do metod pokaż i ukryj w klasie Figura długich ciągów instrukcji warunkowych (lub wyboru) sprawdzających w jakim obiekcie te metody zostały wywołane i wywoływanie na tej podstawie odpowiednich funkcji rysujących. Takie rozwiązanie jest bardzo niedobre, bo stosując je dostajemy jedną gigantyczną o wszystko wiedzącą klasę. Dodanie nowej figury wymaga zmian w większości metod tego giganta, jest więc bardzo trudne i łatwo może powodować błędy. Ponieważ w programowaniu obiektowym chcielibyśmy, żeby każdy obiekt reprezentował jakąś konkretną rzecz z modelowanego przez nas świata i żeby jego wiedza była w jak największym stopniu lokalna musimy mieć w języku mechanizm rozwiązujący przedstawiony problem. Tym mechanizmem są metody wirtualne.

Metoda wirtualna tym różni się od metody zwykłej, że dopiero w czasie wykonywania programu podejmuje się decyzję o tym, która metoda zostanie wywołana. Deklarację metody wirtualnej poprzedzamy słowem virtual. Wirtualne mogą być tylko metody (a nie np. funkcje globalne). Deklarowanie metod wirtualnych ma sens tylko w hierarchiach klas. Jeśli w klasie bazowej zadeklarowano jakąś metodę jako wirtualną, to w klasie pochodnej można:

- zdefiniować jeszcze raz tę metodę (z inną treścią). Można wówczas użyć słowa virtual, ale jest to nadmiarowe. Ta metoda musi mieć dokładnie te same typy argumentów. Wówczas w tej klasie obowiązuje zmieniona definicja tej metody.
- nie definiować jej ponownie. Wówczas w tej klasie obowiązuje ta sama definicja metody co w klasie bazowej.

To samo dotyczy dalszych klas pochodnych.

Popatrzmy na przykład:

```
class A {
  public:
    void virtual f(int);
};

class B: public A {
};

class C: public B {
  public:
    void f(int);// Ta metoda jest wirtualna!
};

int main()
{
  A *p;
  p = new A;
  p->f(3);       //  A::f()
  p = new B;
  p->f(3);       //  A::f()
  p = new C;
  p->f(3);       //  C::f(), bo *p jest obiektem klasy C
    // Ale:
  A a;
  C c;
  a = c;
  a.f(3);        // A::f(), bo a jest obiektem klasy A
  return 0;
}
```

Implementacja: tablica metod wirtualnych.

Klasy abstrakcyjne

Często tworząc hierarchię klas na jej szczycie umieszcza się jedną (lub więcej) klas, o których wiemy, że nie będziemy tworzyć obiektów tych klas. Możemy łatwo zagwarantować, że tak rzeczywiście będzie, deklarując jedną (lub więcej) metod w tej klasie jako czyste funkcje wirtualne. Składniowo oznacza to tyle, że po ich nagłówku (ale jeszcze przed średnikiem) umieszcza się =0 i oczywiście nie podaje się ich implementacji O ile w klasie pochodnej nie przedefiniujemy wszystkich takich funkcji, klasa pochodna też będzie abstrakcyjna. W podanym poprzednio przykładzie z figurami, powinniśmy więc napisać:

```
class Figura {
  //
     void pokaż() = 0;
     void schowaj() = 0;
};
```

Konstruktory i destruktory w hierarchiach klas

Jak pamiętamy obiekt klasy dziedziczącej po innej klasie przypomina kanapkę, tzn. składa się z wielu warstw, każda odpowiadająca jednej z nadklas w hierarchii dziedziczenia. Tworząc taki obiekt musimy zadbać o zainicjalizowanie wszystkich warstw. Ponadto klasy mogą mieć składowe również będące obiektami klas - je też trzeba zainicjalizować w konstruktorze. Na szczęście w podklasie musimy zadbać o inicjalizowanie jedynie:

- bezpośredniej nadklasy,
- własnych składowych.

tzn. my nie musimy już (i nie możemy) inicjalizować dalszych nadklas oraz składowych z nadklas. Powód jest oczywisty: to robi konstruktor nadklasy. My wywołując go (w celu zainicjalizowania bezpośredniej klasy bazowej) spowodujemy (pośrednio) inicjalizację wszystkich warstw pochodzących z dalszych klas bazowych.

Nie musimy inicjalizować nadklasy, jeśli ta posiada konstruktor domyślny (i wystarczy nam taka inicjalizacja). Nie musimy inicjalizować składowych, które mają konstruktor domyślny (i wystarcza nam taka inicjalizacja).
Składnia inicjalizacji: po nagłówku konstruktora umieszczamy nazwę nadklasy (składowej), a po niej w nawiasach parametr(y) konstruktora.

Kolejność inicjalizacji:

- najpierw inicjalizuje się klasę bazową,
- następnie inicjalizuje się składowe (w kolejności deklaracji, niezależnie od kolejności inicjatorów).

(Uniezależnienie od kolejności inicjatorów służy zagwarantowaniu tego, że podobiekty i składowe zostaną zniszczone w odwrotnej kolejności niż były inicjalizowane.)

Czemu ważna jest możliwość inicjalizowania składowych:

```
class A {
    /* ... */
  public:
    A(int);
    A();
};

class B {
    A a;
  public:
    B(A&);
};
```

Rozważmy następujące wersje konstruktora dla B:

```
B::B(A& a2) { a = a2; }
```

i

```
B::B(A& a2): a(a2) {};
```

W pierwszej na obiektach klasy A wykonują się dwie operacje:

- tworzenie i inicjalizacja konstruktorem domyślnym,
- przypisanie.

W drugim tylko jedna:

- tworzenie i inicjalizowanie konstruktorem kopiującym.

Tak więc druga wersja konstruktora jest lepsza.

Niszczenie obiektu:

Treść destruktora wykonuje się przed destruktorami dla obiektów składowych. Destruktory dla (niestatycznych) obiektów składowych wykonuje się przed destruktorem (-rami) klas bazowych.

W konstruktorach i destruktorach można wywoływać metody klasy, w tym także wirtualne. Ale uwaga: wywołana funkcja będzie tą, zdefiniowaną w klasie konstruktora/destruktora lub jednej z jej klas bazowych, a nie tą, która ją później unieważnia w klasie pochodnej.

Kilka ostatnio wydanych książek o C++

- **"C++. 50 efektywnych sposobów na udoskonalenie Twoich programów"; Scott Meyers; Helion 11/2003; ("Effective C++: 50 Specific Ways to Improve Your Programs and Design"; Addison-Wesley; 2nd ed. Sep. 1997; 1st ed. Dec. 30, 1991); 248 str.**
- **"C++. Biblioteka standardowa. Podręcznik programisty"; Nicolai M. Josuttis; Helion 09/2003; ("The C++ Standard Library: A Tutorial and Reference"; Addison-Wesley; 1st ed., Aug. 12, 1999;); 728 str.**
- **C++. Inżynieria programowania; Victor Shtern; Helion 12/2003; (Core C++: A Software Engineering Approach; Prentice Hall; 1st ed. January, 2000) 1088 str.**
- **C++. Kruczki i fortele w programowaniu. Stephen C. Dewhurst; Helion 12/2003; (C++ Gotchas: Avoiding Common Problems in Coding and Design; Addison-Wesley; 1st ed. Nov. 26, 2002), 272 str.**
- **"C++. Potęga języka. Od przykładu do przykładu"; Andrew Koenig, Barbara E. Moo; Helion 1/2004; ("Accelerated C++. Practical Programming by Example"; Addison-Wesley; 1st edition Jan. 15, 2000); 432 str.**
- **"C++. Strategie i taktyki. Vademecum profesjonalisty"; Robert B. Murray; Helion 12/2003 (C++ Strategies and Tactics; Addison-Wesley; Mar. 1993); 240 str.**
- **"C++. Styl i technika zaawansowanego programowania"; James O. Coplien; Helion 1/2004; (Advanced C++ Programming Styles and Idioms; Addison-Wesley; Sep. 1991); 480 str.**
- **"C++. Styl programowania"; Tom Cargill; Helion 12/2003; (C++ Programming Style, Addison-Wesley; 1st ed. Aug. 1992); 224 str.**
- **"Istota języka C++. Zwięzły opis"; Stanley B. Lippman; WNT 1/2004; ("Essential C++"; Addison-Wesley; 1st edition Oct. 26, 1999); 326 str.**

==========

- **"C++. Szablony. Vademecum profesjonalisty"; David Vandevoorde, Nicolai M. Josuttis; Helion 7/2003; (C++ Templates The Complete Guide; Addison-Wesley; 1st ed. Oct. 26, 1999); 480 str.**

Operatory

Wprowadzenie

Motywacja

Klasy definiowane przez użytkownika muszą być co najmniej tak samo dobrymi typami jak typy wbudowane. Oznacza to, że:

- muszą dać się efektywnie zaimplementować,
- muszą dać się wygodnie używać.

To drugie wymaga, by twórca klasy mógł definiować operatory.

Ostrzeżenie

Operatory definiujemy przede wszystkim po to, by móc czytelnie i wygodnie zapisywać programy. Jednak bardzo łatwo można nadużyć tego narzędzia (np. definiując operację + na macierzach jako odejmowanie macierzy). Dlatego projektując operatory (symbole z którymi są bardzo silnie związane pewne intuicyjne znaczenia), trzeba zachować szczególną rozwagę.

Opis

Większość operatorów języka C++ można przeciążać, tzn. definiować ich znaczenie w sposób odpowiedni dla własnych klas. Przeciążanie operatora polega na zdefiniowaniu metody (prawie zawsze może to też być funkcja) o nazwie składającej się ze słowa operator i nazwy operatora (np. operator=). Poniższe operatory można przeciążać:

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

Dla poniższych operatorów można przeciążać zarówno ich postać jedno- jak i dwuargumentową:

- +, -, *, &.

Tych operatorów nie można przeciążać:

- ., .*, ::, ?:, sizeof (ani symboli preprocesora # i ##)

Operatory new i delete mają specyficzne znaczenie i nie odnoszą się do nich przedstawione tu reguły.

Tak zdefiniowane metody (funkcje) można wywoływać zarówno w notacji operatorowej:

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

jak i funkcyjnej (tej postaci praktycznie się nie stosuje):

```
a.operator=(b.operator+(c));
```

Uwagi dotyczące definiowania operatorów:

- Definiując operator nie można zmieniać jego priorytetu, łączności ani liczby argumentów. Można natomiast dowolnie (p. nast. punkt) ustalać ich typy, jak również typ wyniku.
- Jeśli definiujemy operator jako funkcję, to musi ona mieć co najmniej jeden argument będący klasą bądź referencją do klasy. Powód: chcemy, żeby 1+3 zawsze znaczyło 4, a nie np. -2.
- Operatory =, (), [] i -> można deklarować jedynie jako (niestatyczne) metody.
- Metody operatorów dziedziczą się (poza wyjątkiem operator=).
- Nie ma obowiązku zachowywania równoważności operatorów występujących w przypadku typów podstawowych (np. ++a nie musi być tym samym co a+=1).
- Operator przeciążony nie może mieć argumentów domyślnych.

Operatory jednoargumentowe

Operator jednoargumentowy (przedrostkowy) @ można zadeklarować jako:

- (niestatyczną) metodę składową bez argumentów:
  typ operator@()
  i wówczas @a jest interpretowane jako:
  a.operator@()
- funkcję przyjmującą jeden argument:
  typ1 operator@(typ2)
  i wówczas @a jest interpretowane jako:
  operator@(a).

Jeśli zadeklarowano obie postacie, to do określenia z której z nich skorzystać używa się standardowego mechanizmu dopasowywania argumentów.

Operatorów ++ oraz -- można używać zarówno w postaci przedrostkowej jak i przyrostkowej. W celu rozróżnienia definicji przedrostkowego i przyrostkowego ++ (--) wprowadza się dla operatorów przyrostkowych dodatkowy parametr typu int (jego wartością w momencie wywołania będzie liczba 0). Oto przykład:

```
class X
{
  public:
     X operator++();          // przedrostkowe ++a
     X operator++(int);       // przyrostkowe a++
};

// Uwaga: ze względu na znaczenie tych operatorów
// pierwszy z nich raczej definiuje się jako:
//       X& operator++();

int main()
{
  X a;
  ++a;                  // to samo co: a.operator++();
  a++;                  // to samo co: a.operator++(0);
}
```

Operatory dwuargumentowe

Operator dwuargumentowy @ można zadeklarować jako:

- (niestatyczną) metodę składową z jednym argumentem:
  typ1 operator@(typ2)
  i wówczas a @ b jest interpretowane jako:
  a.operator@(b)
- funkcję przyjmującą dwa argumenty:
  typ1 operator@(typ2, typ3)
  i wówczas a @ b jest interpretowane jako:
  operator@(a, b).

Jeśli zadeklarowano obie postacie, to do określenia z której z nich skorzystać używa się standardowego mechanizmu dopasowywania argumentów.

Kiedy definiować operator jako funkcję, a kiedy jako metodę?

Bardziej naturalne jest definiowanie operatorów jako metod, gdyż operator jest częścią definicji klasy, zatem także tekstowo powinien znajdować się w tej definicji. Są jednak sytuacje wymuszające odstępstwa od tej reguły:

- Czasem operator pobiera argumenty będące obiektami dwu różnych klas, wówczas nie widać w której z tych klas miał by być zdefiniowany (ze względów składniowych musiałby być zdefiniowany w klasie, z której pochodzi pierwszy argument). Co więcej czasami definiując taki operator mamy możliwość modyfikowania tylko jednej z tych klas, i może to akurat być klasa drugiego argumentu operatora (np. operator<<).
- Czasami zamiast definiować wszystkie możliwe kombinacje typów argumentów operatora, definiujemy tylko jedną jego postać i odpowiednie konwersje.

Oto przykład:

```
class Zespolona {
  //
  public:
    Zespolona(double);  // Konstruktor ale i konwersja
    operator+(Zespolona&, Zespolona&);
};
```

Przy powyższych deklaracjach można napisać:

```
Zespolona z1, z2;
z1 = z2 + 1;    // Niejawne użycie konwersji
```

ale nie można napisać:

```
z1 = 1 + z2;
```

co jest bardzo nienaturalne. Gdybyśmy zdefiniowali operator+ jako funkcję, nie było by tego problemu.

Operator przypisania

- Operator przypisania **jest czymś innym** niż konstruktor kopiujący!
- O ile nie zostanie zdefiniowany przez użytkownika, to będzie zdefiniowany domyślnie, jako przypisanie składowa po składowej (więc nie musi to być przypisywanie bajt po bajcie). Język C++ nie definiuje kolejności tych przypisań.
- Zwykle typ wyniku definiuje się jako X&, gdzie X jest nazwą klasy, dla której definiujemy operator=.
- Uwaga na przypisania x = x, dla nich operator= też musi działać poprawnie!
- Jeśli uważamy, że dla definiowanej klasy operator= nie ma sensu, to nie wystarczy go nie definiować (bo zostanie wygenerowany automatycznie). Musimy zabronić jego stosowania. Można to zrobić na dwa sposoby:

- zdefiniować jego treść jako wypisanie komunikatu i przerwanie działają programu (kiepskie, bo zadziała dopiero w czasie wykonywania programu),
- zdefiniować go (jako pusty) w części private (to jest dobre rozwiązanie, bo teraz już w czasie kompilacji otrzymamy komunikaty o próbie użycia tego operatora poza tą klasą).

Operator wywołania funkcji

Wywołanie:

wyrażenie_proste(lista_wyrażeń)

uważa się za operator dwuargumentowy z wyrażeniem prostym jako pierwszym argumentem i, być może pustą, listą wyrażeń jako drugim. Zatem wywołanie:

x(arg1, arg2, arg3)

interpretuje się jako:

x.operator()(arg1, arg2, arg3)

Operator indeksowania

Wyrażenie:

wyrażenie_proste [wyrażenie]

interpretuje się jako operator dwuargumentowy. Zatem wyrażenie:

x[y]

interpretuje się jako:

x.operator[](y)

Operator dostępu do składowej klasy

Wyrażenie:

wyrażenie_proste -> wyrażenie_proste

uważa się za operator jednoargumentowy. Wyrażenie:

x -> m

interpretuje się jako:

(x.operator->())->m

Zatem operator->() musi dawać wskaźnik do klasy, obiekt klasy albo referencję do klasy. W dwu ostatnich przypadkach, ta klasa musi mieć zdefiniowany operator -> (w końcu musimy uzyskać coś co będzie wskaźnikiem).

Konwersje typów

W C++ możemy specyfikować konwersje typów na dwa sposoby:

- do definiowanej klasy z innego typu (konstruktory),
- z definiowanej klasy do innego typu (operatory konwersji).

Oba te rodzaje konwersji nazywa się konwersjami zdefiniowanymi przez użytkownika. Są one używane niejawnie wraz z konwersjami standardowymi. Konwersje zdefiniowane przez użytkownika stosuje się jedynie wtedy, gdy są jednoznaczne. Przy liczeniu jednej wartości kompilator może użyć niejawnie co najwyżej jednej konwersji zdefiniowanej przez użytkownika. Na przykład:

```
class X { /* ... */  X(int); };
class Y { /* ... */  Y(X); };
Y a = 1;
// Niepoprawne, bo Y(X(1)) zawiera już dwie konwersje użytkownika
```

Uwaga:
ponieważ kompilator stosuje konwersje niejawnie trzeba być bardzo ostrożnym przy ich definiowaniu. Definiujmy je dopiero wtedy, gdy uznamy to za absolutnie konieczne i naturalne

Operatory konwersji

- Są metodami o nazwie:
  operator nazwa_typu
- nie deklarujemy typu wyniku, bo musi być dokładnie taki sam jak w nazwie operatora,
- taka metoda musi instrukcją return przekazywać obiekt odpowiedniego typu.

Operatory new i delete

Jeśli zostaną zdefiniowane, będą używane przez kompilator w momencie wywoływania operacji new i delete do (odpowiednio) przydzielania i zwalniania pamięci. Opis zastosowania tych metod nie mieści się w ramach tego wykładu.

Operatory << i >>

Służą (m.in.) do wczytywania i wypisywania obiektów definiowanej klasy. Zostaną omówione wraz ze strumieniami.

Szablony (wzorce)

Projektując abstrakcyjny typ danych chcemy to zrobić tak, by można było do niego wstawiać obiekty dowolnego innego typu. W Pascalu nie było to możliwe. Spójrzmy jak można by osiągnąć taki efekt korzystając z dziedziczenia.

Załóżmy, że chcemy zdefiniować abstrakcyjny stos. Elementami tego stosu będą mogły być obiekty klas pochodnych po klasie EltStosu.

```
// ------------------------------------------------------------
//   Klasa EltStosu
// ------------------------------------------------------------

class EltStosu{
 // Podklasy tej klasy będą elementami stosu.
     virtual ~EltStosu(){};
};
```

```
// ------------------------------------------------------------
//   Klasa Stos
// ------------------------------------------------------------

class Stos{
 // Możliwie najprostsza implementacja stosu
 private:                 // Nie przewiduje dziedziczenia
  EltStosu** tab;         // Tablica wskaźników do elementów stosu
  unsigned size;          // Rozmiar tablicy *tab
  unsigned top;           // Indeks pierwszego wolnego miejsca na stosie

 public:
  Stos(unsigned = 10);
  ~Stos();

  void push(EltStosu);
  EltStosu pop();
  int empty();
};
```

```
// ------------------------------------------------------------
//   Implementacja klasy Stos
// ------------------------------------------------------------

Stos::Stos(unsigned s)
 {
  size = s;
  top = 0;
  tab = new EltStosu*[size];
  if (!tab)
     blad("Brak pamięci na utworzenie stosu");
 }

Stos::~Stos()
 {
  for (unsigned i=0; i<top; i++)
     delete tab[i];
  delete[] tab;
 }

void Stos::push(EltStosu elt)
{
  if (top < size)
     {
     tab[top] = new EltStosu(elt);
     if (!tab[top])
        blad("brak miejsca na nowy element stosu");
      top++;
     }
  else
     blad("przepełnienie stosu");
}
```

```
EltStosu Stos::pop()
 {
  if (!top)
     blad("brak elementów na stosie");

// Ze względu na konieczność usuwania wykonuję tu
// dwukrotne kopiowanie elementu stosu
  EltStosu res(*tab[--top]);
  delete tab[top];
  return res;
}

int Stos::empty()
 {
  return top == 0;
 }
```

Muszę zdefiniować podklasę reprezentującą wkładane elementy:

```
class Liczba: public EltStosu{
 private:
  int i;
 public:
  Liczba(int);
};

Liczba::Liczba(int k): i(k) {}
```

A oto przykładowy program główny:

```
int main()
 {
  Stos s;
  int i;

  s.push(Liczba(3)); // Nie przejdzie bez jawnej konwersji
  s.push(Liczba(5));
  cout << "\nStan stosu: " << s.empty();
  while (!s.empty())
     {
     i = s.pop();
     cout << "\n:" << i;
     }
  return 0;
}
```

Niestety tak zapisany program zawiera sporo błędów:

- Operacja push ma argument przekazywany przez wartość - jest to wprawdzie poprawne językowo, ale spowoduje że na stosie będą kopie jedynie fragmentów obiektów Liczba i w dodatku tych nieciekawych fragmentów, bo pochodzących z klasy EltStosu. Można temu prosto zaradzić deklarując nagłówek push następująco:

```
void Stos::push(EltStosu& elt)
```

  Zyskujemy przy okazji jedno kopiowanie mniej.
- Nadal operacja push jest zła, bo w niej wywołujemy konstruktor kopiujący z klasy EltStosu (który skopiuje tylko tę nieciekawą część wkładanego obiektu). Żeby temu zapobiec definiujemy w klasie EltStosu czystą funkcję wirtualną kopia, dającą wskaźnik do kopii oryginalnego obiektu:

```
class EltStosu{
   public:
    virtual EltStosu* kopia() = 0;
    virtual ~EltStosu(){};
};
```

Teraz muszę tę metodę przedefiniować w klasie Liczba:

```
EltStosu* Liczba::kopia()
  { return new Liczba(i); }
```

Treść push jest teraz taka:

```
void Stos::push(EltStosu& elt)
 {
  if (top < size)
  {
  tab[top] = elt.kopia();
  if (!tab[top])
     blad("brak miejsca na nowy element stosu");
  top++;
  }
  else
     blad("przepełnienie stosu");
 }
```

- Metoda pop też jest zła: wynik jest typu EltStosu, więc skopiuje się tylko ten nieciekawy fragment. Zmieńmy więc typ jej wyniku na wskaźnik do EltuStosu, wymaga to też zmian w treści:

```
EltStosu* Stos::pop()
  {
   if (!top)
      blad("brak elementów na stosie");

   return tab[--top];
  }
```

- Musimy jeszcze zmienić treść programu głównego.

```
int main()
{
   Stos s;
   int i;
   s.push(Liczba(3)); // Nie przejdzie bez jawnej
konwersji
   s.push(Liczba(5));
   cout << "\nStan stosu: " << s.empty();
   while (!s.empty())
      {
      i = ((Liczba*)s.pop())->i;
      cout << "\n:" << i;
      }
   return 0;
}
```

Podsumujmy wady tego rozwiązania:
Wady:

- wkładając muszę używać konwersji to_co_chcę_włożyć -> podklasa_EltStosu,
- muszę zdefiniować podklasę EltStosu,
- muszę w niej zdefiniować konstruktor i operację kopia.

Poważne wady:

- użytkownik musi pamiętać o usuwaniu pamięci po pobranych obiektach,
- użytkownik musi dokonywać rzutowań typu przy pobieraniu (a to jest nie tylko niewygodne, ale przede wszystkim niebezpieczne).

Zalety:

- można naraz trzymać na stosie elementy różnych typów, byleby były podtypami EltStosu. Trzeba tylko umieć potem je z tego stosu zdjąć (tzn. wiedzieć co się zdjęło).

Można też zdefiniować wyspecjalizowaną podklasę stosu, ale lepsze efekty da zdefiniowanie wyspecjalizowanego stosu zawierającego powyższy stos ogólny jako element. Wymaga to jednak definiowania nowej klasy.

Potrzebujemy zatem innego narzędzia umożliwiającego parametryzowanie struktur danych. Tym narzędziem są szablony.

Szablon klasy specyfikuje jak można konstruować poszczególne klasy (podobnie jak klasa specyfikuje jak można konstruować poszczególne obiekty). Deklarację szablonu poprzedzamy słowem **template**, po którym w nawiasach kątowych podajemy parametry szablonu. Parametry szablonu zwykle są typami. Parametr szablonu będący typem deklarujemy używając słowa **class** (**typename**) a potem nazwy parametru (to nie oznacza, że ten typ musi być klasą). W deklaracji klasy możemy używać tak zadeklarowanych parametrów (tam gdzie potrzebujemy typów). Parametr szablonu może także być zwykłym parametrem typu całkowitego, wyliczeniowego lub wskaźnikowego (a nowy projekt języka dopuszcza jeszcze szablony).

Oto przykład deklaracji szablonu:

```
template <class T>
class Stos{
 // Możliwie najprostsza implementacja stosu
 protected:
  T** tab;          // Tablica wskaźników do elementów stosu
  unsigned size; // Rozmiar tablicy *tab
  unsigned top;  // Indeks pierwszego wolnego miejsca na
stosie

 public:
  Stos(unsigned = 10);
  ~Stos();

  void push(T&);
  T pop();
  int empty();
};
```

Deklarując metody dla tego szablonu musimy je poprzedzać informacją, że są szablonami metod:

```
template <class T>
Stos<T>::Stos(unsigned s)
 {
  size = s;
  top = 0;
  tab = new T*[size];
  if (!tab)
     blad("Brak pamięci na utworzenie stosu");
 }
```

```
template <class T>
Stos<T>::~Stos()
 {
  for (unsigned i=0; i<top; i++)
     delete tab[i];
  delete[] tab;
 }

template <class T>
void Stos<T>::push(T& elt)
 {
  if (top < size)
     {
     tab[top] = new T(elt);
     if (!tab[top])
        blad("brak miejsca na nowy element stosu");
     top++;
     }
  else
     blad("przepełnienie stosu");
 }

template <class T>
T Stos<T>::pop()
 {
  if (!top)
     blad("brak elementów na stosie");

  T res(*tab[--top]);   // Ze względu na konieczność usuwania mam
  delete tab[top];      // dwie operacje kopiowania w pop().
  return res;
 }
```

```
template <class T>
int Stos<T>::empty()
 {
  return top == 0;
 }
```

Użycie szablonu polega na zapisaniu jego nazwy wraz z odpowiednimi parametrami (ujętymi w nawiasy kątowe). Takie użycie szablonu może wystąpić wszędzie tam gdzie może wystąpić typ. Parametrami aktualnymi szablonu odpowiadającymi parametrom formalnym określającym typ mogą być dowolne typy (niekoniecznie klasy). Oto przykład użycia szablonu Stos:

```
int main()
 {
  Stos<int> s;
  int i;
  s.push(3);
  s.push(5);
  cout << "\nStan stosu: " << s.empty();
  while (!s.empty())
     {
     i = s.pop();
     cout << "\n:" << i;
     }
  return 0;
 }
```

Można zadeklarować nowy typ będący ukonkretnionym szablonem:

```
typedef Stos<int> StosInt;
```

Można użyć szablonu jako klasy bazowej:

```
class Stos_specjalny: public Stos<int> { /* ... */ }
```

(podklasa może być zwykłą klasą bądź znów szablonem).

Uwaga: Mechanizm szablonów jest realizowany w podobny sposób do rozwijania makropoleceń, to znaczy, że kompilator przy każdym użyciu szablonu z nowym parametrem generuje dla tego nowego parametru nową klasę. Wynika stąd, że kompilator nie musi przeprowadzać (i nie przeprowadza) pełnej analizy poprawności szablonu w momencie napotkania jego deklaracji, robi to dopiero podczas generowania konkretnych klas. Dzięki temu mechanizm szablonów jest dużo bardziej elastyczny, można zdefiniować szablon sparametryzowany typem T, wymagający by dla typu T był zdefiniowany operator +. Ponieważ nie wszystkie typy mają operator +, taki szablon w ogólności nie jest poprawny. Nic nie stoi jednak na przeszkodzie, by używać tego szablonu dla tych typów, dla których operator + jest zdefiniowany. Szablon można sparametryzować kilkoma

parametrami. Jak już wspominaliśmy, parametry nie muszą być typami. Oto przykład deklaracji szablonu stosu o zadanej podczas kompilacji liczbie elementów, przykład definicji metody i przykład użycia:

```
int main()
 {
  Stos<int,100> s;
  Stos< float, 10> s2;
// ...
  s.push(3);
// ...
return 0;
 }
```

Oprócz szablonów klas można także tworzyć szablony funkcji. Oto deklaracja uniwersalnej funkcji sortującej:

```
template<class T>
void sortuj(T tab[], unsigned n)
{  /* Treść tej funkcji */  }
```

W ten sposób zdefiniowaliśmy nieskończoną rodzinę funkcji sortujących (oczywiście kompilator będzie generował elementy tej rodziny tylko w miarę potrzeby). Teraz można sortować dowolne tablice.

Przy wywoływaniu funkcji zdefiniowanej przez szablon nie podaje się jawnie argumentów szablonu, generuje je kompilator, oto przykład:

```
// ...
int t1[100];
Zespolone t2[20];
sortuj(t1, 100);   // wywołanie z T równym int
sortuj(t2, 20);    // wywołanie z T równym Zespolona
```

Każdy argument szablonu funkcji musi wystąpić jako typ argumentu w szablonie funkcji.

Obsługa wyjątków

Wprowadzenie

Bardzo często zdarza się, że pisząc jakąś operację (funkcję), zauważamy, że ta operacja nie zawsze musi się dać poprawnie wykonać. Nasza funkcja powinna jakoś zareagować w takiej sytuacji, kłopot polega na tym, że nie wiemy jak. Może:

1. Wypisać komunikat i przerwać działanie całego programu.
   Bardzo brutalne.
2. Przekazać wartość oznaczającą błąd.
   Nie zawsze jest wykonalne (może nie być wartości, która nie może być poprawną wartością funkcji). Poza tym zwykle jest bardzo niewygodne, bo wymaga sprawdzania wartości funkcji po każdym jej wywołaniu. Program konsekwentnie wykonujący takie sprawdzenia staje zupełnie nieczytelny, jeśli zaś sprawdzanie nie jest konsekwentnie stosowane to jest warte tyle samo, co gdyby go w ogóle nie było.
3. Przekazać jakąś poprawną wartość, natomiast ustawić jakąś zmienną (zmienne) w programie sygnalizujące zaistnienie błędnej sytuacji.
   To już jest zupełnie złe rozwiązanie: tak samo jak poprzednie wymaga ciągłego sprawdzania czy nie nastąpił błąd, jest też bardzo

prawdopodobne, że używający takiej operacji w ogóle nie będzie świadom tego, że błędy w ogóle są sygnalizowane.

4. Wywołać funkcję dostarczoną przez użytkownika (np. jako parametr), obsługującą błędne sytuacje.
   Najlepsze rozwiązanie. Jego wadą jest to, że każde wywołanie funkcji trzeba obciążyć dodatkowym parametrem.

W celu rozwiązania takich problemów włączono do języka C++ mechanizm obsługi wyjątków. W C++ **wyjątek** oznacza błąd, zaś obsługa wyjątków oznacza reakcję programu na błędy wykryte podczas działania programu. Idea obsługi wyjątków polega na tym, że funkcja, która napotkała problem, z którym nie potrafi sobie poradzić **zgłasza** wyjątek. Wyjątek jest przesyłany do miejsca wywołania funkcji. Tam może być **wyłapany** i **obsłużony** lub może być przesłany dalej (wyżej). Podczas tego przechodzenia, przy wychodzeniu z funkcji i bloków następuje automatyczne usuwanie automatycznych obiektów stworzonych w tych funkcjach i blokach (to bardzo ważne). W C++ nie możliwości powrotu z obsługi wyjątku, do miejsca jego wystąpienia, w celu

ponownego wykonania akcji, która spowodowała błąd.

Uwaga: Mechanizm obsługi wyjątków w innych językach może być zrealizowany zupełnie inaczej (np. wyjątek nie musi być utożsamiany z błędem, może być możliwe wznowienie wykonywania programu w miejscu wystąpienia wyjątku itp.). W szczególności nie ma ogólnej zgody czym powinien być wyjątek.

Składnia instrukcji związanych z obsługą wyjątków:

```
try {
  <instrukcje>
}
catch (<parametr 1>) {
  <obsługa wyjątku 1>
}
// ...
catch (<parametr n>) {
  <obsługa wyjątku n>
}
```

Zgłoszenie wyjątku:

```
throw <wyrażenie>
```

Semantyka:

Jeśli jakaś z <instrukcji> zgłosiła wyjątek, to przerywamy wykonywanie tego ciągu instrukcji i szukamy instrukcji catch z odpowiednim parametrem. Jeśli znajdziemy, to wykonujemy obsługę tego wyjątku. instrukcje catch są przeglądane w kolejności ich deklaracji. Po zakończeniu obsługi wyjątku (o ile obsługa wyjątku jawnie nie spowodowała przejścia do innej części programu) wykonuje się pierwszą instrukcję stojącą po instrukcjach catch. Jeśli zaś nie znaleziono obsługi stosownego wyjątku, to następuje przejście do wywołującej funkcji, połączone z usunięciem obiektów automatycznych i tam znów rozpoczyna się poszukiwanie obsługi wyjątku. Jeśli takie poszukiwanie nie zakończy się sukcesem, to wykonywanie programu zostanie przerwane.

Przykład:

```
class Wektor{
  int *p;
  int rozm;
public:
  class Zakres{};          // Wyjątek: wyjście poza zakres
  class Rozmiar{};         // Wyjątek: zły rozmiar wektora
  Wektor(int r);
  int& operator[](int i);
```

```
  // ...
};

Wektor::Wektor(int r)
{
  if (r<=0)
     throw Rozmiar();
  // ...
}

int& Wektor::operator[](int i)
{
  if (0<=i && i < rozm)
     return p[i];
  else
     throw Zakres();    // Na razie nie przekazujemy wartości
błędnego indeksu
}

void f()
{
  try{
     // używanie wektorów
  }
  catch (Wektor::Zakres) {
     // obsługa błędu przekroczenia zakresu
  }
  catch (Wektor::Rozmiar) {
     // obsługa błędu polegającego na błędnym podaniu
rozmiaru
  }
  // Sterowanie do chodzi tutaj, gdy:
  // a) nie było zgłoszenia wyjątku, lub
```

```
  // b) zgłoszono wyjątek Zakres lub Rozmiar i obsługa
tego
  //      wyjątku nie zawierała instrukcji powodujących
wyjście
  //      z funkcji f
}
```

Obsługa wyjątków może być podzielona między wiele funkcji:

```
void f1()
{
  try{ f2(w); }
  catch (Wektor::Rozmiar) { /* ... */ }
}
void f2(Wektor& w)
{
  try{ /* używanie wektora w */ }
  catch (Wektor::Zakres) { /* ... */ }
}
```

W instrukcjach obsługujących wyjątek może się pojawić instrukcja throw. W szczególności może się też pojawić instrukcja throw zgłaszająca taki sam wyjątek, jak ten właśnie wywoływany. Nie spowoduje to zapętlenia ani nie będzie błędem. Z punktu widzenia języka C++ wyjątek jest obsłużony z chwilą wejścia do procedury obsługi wyjątku, zaś wyjątki zgłaszane w procedurach

obsługi są obsługiwane przez funkcje wywołujące blok try. Można także zagnieżdżać bloki try-catch w instrukcjach catch (nie wydaje się to jednak celowe).

Przekazywanie informacji wraz z wyjątkiem

Instrukcja zgłaszająca wyjątek (throw), zgłasza obiekt. Taki obiekt może posiadać składowe i dzięki nim przenosić informację z miejsca zgłoszenia do miejsca obsługi.

```
class Wektor{ // ...
  public:
     class Zakres{
     public:
        int indeks;
        Zakres(int i): indeks(i) {}
  };
  int& operator[] (int i);
  // ...
};

int& Wektor::operator[](int i)
{
  if (0<=i && i<rozm) return p[i];
  throw Zakres(i);
}
```

```
// ...

void f(Wektor& w)
{
  // ...
  try{ /* używanie w */ }
  catch (Wektor::Zakres z) {
     cerr << ”Zły indeks” << z.indeks << ”\n”;
     // ...
  }
}
```

Hierarchie wyjątków

Ponieważ wyjątki są obiektami klas, możemy tworzyć hierarchie klas wyjątków. Co to daje? Możemy, w zależności od sytuacji, pisać wyspecjalizowane procedury obsługi wyjątków, lub jedną obsługującą wszystkie wyspecjalizowane wyjątki:

```
class BłądMatemat {};
class Nadmiar: public BłądMatemat {};
class Niedomiar: public BłądMatemat {};
class DzielPrzezZero: public BłądMatemat {};
// ...
try { /* ... */ }
catch (Nadmiar) { /* Obsługa nadmiaru */ }
catch (BłądMatemat) { /* Obsługa pozostałych bł. mat. */
}
```

Wznawianie wyjątku

W procedurze obsługi wyjątku można ponownie zgłosić ten sam wyjątek pisząc throw bez argumentu (jak już poprzednio zaznaczyliśmy, nie spowoduje to zapętlenia).

Obsługa dowolnego wyjątku

catch (....) oznacza wyłapywanie dowolnego wyjątku. Można je zastosować razem z throw:

```
void f() {
  try { /* ... */
  catch (...) {
     /* Instrukcje, które muszą się zawsze wykonać na
koniec
          procedury f, jeśli nastąpił błąd. */
     throw; // Ponowne zgłoszenie złapanego wyjątku
}
```

Zdobywanie zasobów

Częstym problemem związanym z obsługą wyjątków, jest zwalnianie zasobów, które funkcja zdążyła już sobie przydzielić zanim nastąpił błąd. Dzięki wyjątkom możemy bardzo prosto rozwiązać ten problem, obudowując zasoby obiektami (pamiętajmy, że procesowi szukania procedury

obsługi błędu towarzyszy usuwanie obiektów lokalnych).

Rozwiązanie 1 (niewygodne):

```
void używanie-pliku(const char* np)
  FILE* p = fopen(np, ”w”);
  try { /* coś z plikiem p */ }
  catch (...) {
     fclose(p);
     throw;
  }
  fclose(p);
}
```

Rozwiązanie 2 (eleganckie i ogólne)

```
class Wsk_do_pliku{
     FILE* p;
  public:
     Wsk_do_pliku(const char* n, const char * a)
        {p = fopen(n,a); }
     ~Wsk_do_pliku() {fclose(p);}
     operator FILE*() {return p;}
     // Jeśli będę potrzebował  wskaźnika do struktury FILE
};
```

Teraz nasza funkcja daje się już ładnie zapisać, nie ma w niej ani jednej dodatkowej instrukcji, nie ma nawet operacji fclose!

```
void używanie-pliku(const char* np)
  Wsk_do_pliku p(np, ”w”);
  /* coś z plikiem p */
  }
```

Tę technikę nazywa się zwykle „zdobywanie zasobów jest inicjalizacją”.

Zastanówmy się teraz nad następującym problemem. Obiekt uważa się za skonstruowany, dopiero po zakończeniu wykonywania jego konstruktora. Dopiero wtedy porządki wykonywane wraz z szukaniem procedury obsługi błędu usuną obiekt z pamięci (i zwolnią zajmowane przez niego zasoby). Pojawia się więc naturalny problem: co ma robić konstruktor, gdy wykryje błąd? Powinien zwrócić te zasoby, które już sobie przydzielił. Stosując powyższą technikę jesteśmy w stanie bardzo łatwo to zagwarantować. Oto przykład, konstruktor przydzielający dwa zasoby: plik i pamięć:

```
class X{
  Wsk_do_pliku p;
  Wsk_do_pamięci<int> pam;
  // ...
```

```
  X(const char*x, int y): p(x, ”w”), pam(r) { /* inicjalizacja
*/ }
  // ...
};

class Za_mało_pamięci{};

template<class T> class Wsk_do_pamięci{
public:
  T* p;
  Wsk_do_pamięci(size_t);
  ~Wsk_do_pamięci() {delete[] p;}
  operator T*() {return p;}
};

template<class T>
Wsk_do_pamięci::Wsk_do_pamięci(size_t r){
  p = new T[t];
  if (!p)
     throw Za_mało_pamięci();
}
```

Teraz to już implementacja dba o to, by wywołać destruktory dla tych obiektów, które zostały skonstruowane (i tylko dla nich).

Specyfikacja interfejsu funkcji

Jeśli mamy wyjątki, to interfejs funkcji staje się o nie bogatszy. Język C++ pozwala (nie zmusza) do ich wyspecyfikowania:

```
void f() throw (x1, x2, x3) {  /* treść f() */  }
```

co jest równoważne napisaniu:

```
void f()
{
  try { /* treść f() */ }
  catch (x1) { throw; }
  catch (x2) { throw; }
  catch (x3) { throw; }
  catch (...) { unexpected(); }
}
```

Czyli f() może zgłosić wyjątki x1, x2 i x3 oraz pochodne. Domyślnym znaczeniem unexpected() jest zakończenie działania programu. Zwróćmy uwagę, że sprawdzanie czy zgłoszony wyjątek jest we specyfikacji następuje dopiero podczas wykonywania programu (a nie podczas kompilacji). Funkcja dla której nie wyspecyfikowano żadnego wyjątku, może zgłosić każdy wyjątek. Jeśli funkcja ma nie zgłaszać żadnych wyjątków, to piszemy:

```
void g() throw();
```

Strumienie w C++

Wprowadzenie

- Wejście-wyjście nie jest elementem języka C++ (tak samo jak w języku C nie ma typu plikowego). Z punktu widzenia użytkownika języka nie ma to większego znaczenia, jest natomiast bardzo istotne dla samego języka:
  - jest mocny, skoro da się w nim wyrazić nowe pojęcia,
  - jest łatwiej przenaszalny,
  - jest mniejszy.
- Istnieje standardowa biblioteka iostream.h zawierająca operacje we-wy (i związane z nimi typy).
- Operacje we-wy wykonuje się na strumieniach. Strumienie są ciągami znaków, operacje we-wy polegają na przekształceniu obiektów różnych typów na ciągi znaków i odwrotnie.
- Zalety strumieni: jednolite traktowanie typów wbudowanych i definiowanych przez użytkownika.

Wyjście

- Standardowe strumienie: cout i cerr.

  Jest jeszcze ich więcej: clog (to samo co cerr ale buforowany), wcout, wcerr, wclog (używają szerokich znaków).

  Strumienie cout, wcout używają tego samego wyjścia co stdout (C), zaś cerr, wcerr, clog i wclog tego samego co stderr (C).

- Operator <<:

  ostream& operator<<(ostream&, T)

  – T może być typem referencyjnym, może też być podany z const.

  – taki typ wyniku umożliwia pisanie takich wyrażeń jak:

  cout << ” i = ” << i << ‘\n’

  co oznacza (operatory >> i << wiążą w lewo):

  ((cout.operator<<(”i= ”)).operator<<(i)).operator<<(‘\n’);

  – w treści operatora << należy jako wynik przekazać strumień, który jest parametrem operatora <<,

  – ponieważ pierwszy argument tego operatora pochodzi z klasy ostream, nie można go zdefiniować jako metody w klasie T, lecz jako funkcję,

  – zwykle (nie zawsze) zapisanie operatora << dla klasy T wymaga, by zadeklarować go jako zaprzyjaźniony z klasą T,

– zamiast operatora << można dla znaków używać funkcji put (np. cout.put(c)) zaś dla napisów funkcji write.

- Przykład:

```
class Zespolone{
  friend ostream& operator<<(ostream& s, Zespolone)
  // ...
};

ostream& operator<<(ostream& s, Zespolone& z)
  {
    cout << z.re << ”+” << z.im << ”*i”;
    return s;
  }
```

- Wypisywanie wartości logicznych (typu bool): domyślnie wypisuje się 0 lub 1, ale można to zmienić:

```
cout << true << boolalpha << true; // 1 true
```

- Klasa ostream (w pewnym uproszczeniu bo ostream jest zdefiniowane jako: typedef basic_ostream<char> ostream;)

```
class ostream: public virtual ios{
  // ...
public:
  ostream& operator<<(const char*);
  ostream& operator<<(char);
  ostream& operator<<(int);
  ostream& operator<<(short); { return *this << int(i); }
  // ...
};
```

- Jak zdefiniować wirtualny operator <<? P. przykład.

Wejście

- strumień standardowy cin;
- operator >>
  istream& operator>>(istream&, T)
  zwykle implementujemy go tak:

```
istream& operator>>(istream& s, T& zmienna)
{
  // pomiń białe spacje
  // jakoś wczytaj T na zmienną
  return s;
}
```

  operator >> zwykle definiujemy jako funkcję zaprzyjaźnioną z klasą (analogicznie do operatora >>)
- zapis:
  if (s >> zm)
  jest poprawny i oznacza sprawdzenie, czy udało się wczytać daną ze strumienia s na zmienną zm. Taka postać zapisu jest możliwa dzięki operatorowi konwersji typu istream na int (zdefiniowanemu w klasie istream). Oto fragment programu wykorzystujący tę własność:

```
main()
{
  int i;
  while (cin >> i) cout << i;
}
```

- zamiast operatora >> można używać funkcji get, np. cin.get(c). Bywa to potrzebne w następujących sytuacjach:
-

```
main()
{
  char buf[100];
  cin >> buf;                  // Tu możemy wyjść poza tablicę
  cin.get(buf, 100, '\n');     // Tu jesteśmy bezpieczni
},
```

  - cin >> c pomija białe spacje, cin >> buf wczyta tylko do białego znaku (przy deklaracji char* buf).
- standardowy plik ctype.h zawiera wiele funkcji, które mogą być przydatne przy czytaniu i sprawdzaniu:
  - int isalpha(char); // czy znak jest literą,
  - int isdigit(char); // czy znak jest cyfrą,
  - ...
- można wykonać operację:

  s.putback(c);

  wstawiającą znak z powrotem do strumienia.

Stany strumienia

Z każdym strumieniem istream, ostream związany jest stan. Można go testować na różne sposoby, np. używając:

```
int ios::eof() const;
```

sprawdzającej, czy napotkano koniec strumienia, tzn. czy ostatnie czytanie się powiodło i dało ostatni element strumienia. Stan można także testować używając funkcji rdstate() i stałych bitowych (p. Stroustrup 369-370).

Przykład:
Ten przykład pokazuje jak można zdefiniować własny operator wejścia. Tym razem zakładamy, że liczby zespolone są zapisane w jednym z następujących formatów:

- d Liczba zespolona o zerowej części urojonej,
- (d) j.w.
- (d,d) Liczba zespolona z częścią urojoną i rzeczywistą.

Jednocześnie pokazujemy tu wykorzystanie operacji na stanach strumienia i korzystamy z putback().

```
istream& operator>>(istream & s, Zespolona& z)
{
  double re = 0, im = 0;
  char c = 0;

  s >> c;
  if (c == '(')
  {
   s >> re >> c;
   if (c == ',')
     s >> im >> c;
   if (c != ')')
    s.clear(ios::badbit);       // Ustawiamy stan strumienia
  }
  else
  {
   s.putback(c);
   s >> re;
  }
  if (s) z = Zespolona(re, im);
  return s;
}
```

Formatowanie

- Klasa ios zawiera operacje służące do określania sposobu wypisywania danych, np.:
  int width(int w);

określa minimalną szerokość pola, na którym będzie wypisywana następna liczba bądź napis. Domyślnie jest 0, co oznacza tyle znaków ile będzie potrzeba. Uwaga: width dotyczy tylko jednej (następnej) operacji numerycznego lub napisowego wypisywania. Znak wypełniający można zdefiniować funkcją fil.(c), domyślnie jest nim spacja.

- W klasie ios jest bardzo dużo znaczników (ustawianych i czytanych funkcjami flags i setf) dotyczących formatowania, określających np. to, czy należy pomijać białe znaki przy czytaniu, do której strony mają być dosuwane wypisywane liczby, czy liczby mają być wypisywane ósemkowo czy szesnastkowo czy dziesiętnie itp. (p. Stroustrup 376-380).
- Manipulatory są pojedynczymi elementami wstawianymi jako argumenty >> ustalającymi sposób formatowania kolejnych argumentów operatora >>. Oto przykład pokazujący możliwą implementację manipulatorów:

```
class Flushtype {};
ostream & operator<<(ostream& s, Flushtype)
{
  return flush(s);
}
  Flushtype FLUSH;
  // ...
  cout << x << FLUSH << y << FLUSH;
```

Można jeszcze prościej:

```
typedef ostream& (manip) (ostream&);
```

czyli manip jest wskaźnikiem do funkcji o argumencie typu ostream& i wyniku ostream&. Ale zauważmy, że na przykład funkcja flush ma taki właśnie typ! Zatem:

```
ostream& operator&(ostream& os, manip f)
{
  return f(os);
}
```

Manipulatory mogą mieć dodatkowo parametry, wtedy implementuje się je jako obiekty.

- Można wiązać ze sobą strumienie (tzn. synchronizować operacje na nich):

```
ostream* ie.(ostream* s);
ostream* ie.();
```

Pierwsza operacje wiąże wejście z wyjściem, jako wynik daje poprzednio powiązany strumień (lub 0), druga daje powiązany strumień. Dlaczego to jest ważne? Popatrzmy na poniższy program:

```
main()
{
```

```
    char* s;
    cout << ”Jak się nazywasz?”;
    cin >> s;
    \\ ...
}
```

Gdyby cin nie było powiązane z cout, to program mógłby najpierw wczytywać dane, a dopiero potem wypisywać zapytanie. Wywołanie s.tie(0) przerywa powiązanie (o ile takie było) strumienia s. cin jest domyślnie powiązane z cout.

Pliki i strumienie

Przykład użycia plików ze strumieniami:

```
#include <fstream.h>
// ...
int main(int argc, char* argv[])
{
  // Sprawdzenie liczby parametrów

  ifstream zpliku(argv[1]);
  if (!zpliku)
     błąd(”Nie można otworzyć pliku wejściowego”,
argv[1]);

  ofstream naplik(argv[2]);
  if (!naplik)
     błąd(”Nie można otworzyć pliku wyjściowego”,
argv[2]);

  char ch;
     while (zpliku.get(ch)) naplik.put(ch);

  if(!zpliku.eof() || naplik.bad())
     błąd(”wystąpił jakiś błąd”);

  return 0;
}
```

Uwagi do przykładu:

- otwarcie pliku:

utworzenie obiektu klasy:
ofstream (plik wyjściowy)
istream (plik wejściowy)
z nazwą pliku jako parametrem.

- po otwarciu pliku trzeba sprawdzić jego stan, żeby się upewnić czy otwarcie się powiodło.
- można podać drugi argument specyfikujący tryb otwarcia, np.

```
fstream plik(”dane.txt”, ios::in || ios::out);
```

otworzy plik dane.txt do czytania i pisania.

- wszystkie operacje dostępne dla typów istream oraz ostream można stosować do typu fstream (fstream jest pochodną klasy iostream, która z kolei jest pochodną klas istream i ostream).
- plik można zamknąć jawnie:

```
f.close();
```

robi to też niejawnie destruktor obiektu. Jawne zamykanie stosujemy wtedy, gdy chcemy zwolnić zasoby, a jeszcze nie opuszczamy bloku zawierającego deklarację strumienia.