Programowanie obiektowe i C++

Janusz Jabłonowski

Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski
http://mst.mimuw.edu.pl

18 marca 2012

Wprowadzenie

Ankieta - co umiemy

- Programowałam/em w C++.
- Programowałam/em w języku obiektowym.
- Programowałam/em w C.
- Programowałam/em w innym języku.
- Nigdy nie programowałam/em.

Sprawy organizacyjne

- Rejestracja.
- Godziny wykładu.
- Wydziałowy Moodle.
- Co umiemy.
- Co będzie na wykładach.
- Zasady zaliczania.

Co będzie na wykładach

- Podstawy obiektowości [1].
- Elementy C w C++ [2-4].
- Programowanie obiektowe w C++ (klasy, dziedziczenie, metody wirtualne, szablony, ...) [pozostałe wykłady].
- Być może: Smalltalk, Delphi, Java, C# [tylko jeden z tych tematów i tylko jeśli starczy czasu].

Zasady zaliczania

Zaliczenie ćwiczeń:

- obecności,
- klasówka,
- program zaliczeniowy.

Egzamin na koniec wykładu:

- trzeba będzie napisać fragment programu w C++,
- 1,5 - 2 godz.

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 bibliotek i rozszerzeń języka C++.

Wstęp do programowania obiektowego

Programowanie obiektowe - wstęp

- 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 - wstęp cd.

- 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

- 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.

Polimorfizm

- 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*).

Polimorfizm cd.

- 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 - ponowne wykorzystywanie

- 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 - etap ewolucji

- Programowanie obiektowe stanowi następny etap ewolucji mechanizmów abstrakcji w programowaniu:
  - procedury,
  - bloki (w sensie Smalltalka),
  - 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.

Programowanie obiektowe - podsumowanie

- 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 czymś). Bardzo często ta relacja jest mylona z relacją has-a (ma coś) dotyczącą składania.

Dziedziczenie cd.

- Czasami chcemy wyrazić w hierarchii klas wyjątki (pingwin), można to uzyskać dzięki *przedefiniowywaniu* (*podmienianiu*) 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 (tak nie należy projektować - tu można zastosować składanie, 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 to 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).

Zalety programowania obiektowego cd.

- 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
- Ukrywanie informacji. Użytkownika klasy interesuje tylko interfejs należących do niej obiektów (komunikaty i ich znaczenie), a nie zawartość tych obiektów (metody i dane). Ten mechanizm nazywamy *kapsułkowaniem* (lub hermetyzacją).

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).

Zalety dziedziczenia cd.

- Ponowne wykorzystywanie pozwala szybciej tworzyć nowe systemy (budować je z klocków).
- Zgodność interfejsów (gdy wiele klas dziedziczy po wspólnym przodku).

Wady: 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).

Narzuty związane z programowaniem obiektowym cd.

- Efekt jo-jo (nadużywanie dziedziczenia może uczynić czytanie programu bardzo żmudnym procesem).
- Modyfikacje kodu w nadklasach mają wpływ na podklasy i vice-versa (wirtualność metod).

Podstawy C++: instrukcje

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).
- Standard C++ ISO/IEC 14882:1998, znany jako C++98 (1998).
- Nowy standard C++0x (rok publikacji nadal nieznany).

Elementy C w C++

- Uwaga: to nie jest opis języka C!
- C++ jest kompilowanym 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).
- Poszczególne wiersze odpowiadają poszczególnym wersjom omawianej konstrukcji składniowej.

Instrukcje języka C++

instrukcja:

- instrukcja_etykietowana
- instrukcja_wyrażeniowa
- blok
- instrukcja_warunkowa
- instrukcja_wyboru
- instrukcja_pętli
- instrukcja_deklaracyjna
- instrukcja_próbuj
- instrukcja_skoku

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 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 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.

Przykład przesłaniania

```
struct ff{int k;} f;
//...
f.k++; // poprawne
int f=f.k; // niedozwolone
f++; // poprawne
```

Instrukcja warunkowa

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

- Wyrażenie musi być typu logicznego, arytmetycznego lub wskaźnikowego.
- Wartość warunku jest niejawnie przekształcana na typ **bool**.
- Jeśli wartość wyrażenia jest liczbą lub wskaźnikiem, to wartość różna od zera jest interpretowana jak **true**, wpp. za **false**.
- **else** dotyczy ostatnio spotkanego **if** bez **else**.
- Warunek może być także deklaracją (z pewnymi ograniczeniami) mającą część inicjującą, jej zasięgiem jest cała instrukcja warunkowa.
- Instrukcja składowa może być deklaracją (jej zasięgiem zawsze jest tylko ta instrukcja składowa).

Instrukcja wyboru

instrukcja_wyboru:
switch (wyrażenie) instrukcja

- Powoduje przekazanie sterowania do jednej z podinstrukcji występujących w instrukcji, o etykiecie wyznaczonej przez wartość wyrażenia.
- Wyrażenie 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 wyrażenia, 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

Instrukcje złożone

Pętle while i do

instrukcja_pętli:

while (warunek) instrukcja

do instrukcja **while** (warunek);

pętla_for

warunek:

wyrażenie

Pętle while i do cd

Instrukcja **while**

- Dopóki warunek jest spełniony, wykonuje podaną *instrukcję*.
- *Warunek* wylicza się przed każdym wykonaniem *instrukcji*.
- Może nie wykonać ani jednego obrotu.

Instrukcja **do**

- Powtarza wykonywanie *instrukcji* aż *warunek* przestanie być spełniony.
- *Warunek* wylicza się po każdym wykonaniu *instrukcji*.
- Zawsze wykonuje co najmniej jeden obrót.

Pętle while i do cd

- Warunek musi być typu logicznego, arytmetycznego lub wskaźnikowego.
- Jeśli wartość warunku jest liczbą lub wskaźnikiem, to wartość różną od zera uważa się za warunek prawdziwy, a wartość równą zeru za warunek fałszywy.
- Wartość warunku typu innego niż logiczny jest niejawnie przekształcana na typ **bool**.
- Warunek w pętli **while** może być także deklaracją (z pewnymi ograniczeniami), której zasięgiem jest ta pętla,
- Ta deklaracja musi zawierać część inicjującą.
- Zauważmy, że w pętli **do** warunek nie może być deklaracją.
- *Instrukcja* może być deklaracją (jej zasięgiem zawsze jest tylko ta instrukcja). Przy każdym obrocie pętli sterowanie wchodzi i opuszcza ten lokalny zasięg, z wszelkimi tego konsekwencjami.

Instrukcja pętli - for

pętla_for:

for (inst_ini_for *warunek*$_{opc}$ **;** *wyraenie*$_{opc}$ **)** instrukcja

inst_ini_for:

instrukcja_wyrażeniowa

prosta_deklaracja

- *Warunek* jak w poprzednich pętlach,
- Pominięcie *warunku* jest traktowane jako wpisanie **true**,
- Jeśli instrukcja *instr_inic* jest deklaracją, to zasięg zadeklarowanych nazw sięga do końca pętli,
- Zasięg nazw zadeklarowanych w warunku jest taki sam, jak zasięg nazw zadeklarowanych w inst_ini_for,
- Instrukcja może być deklaracją (jej zasięgiem zawsze jest tylko ta instrukcja). Przy każdym obrocie pętli sterowanie wchodzi i opuszcza ten lokalny zasięg.

Semantyka pętli for

Instrukcja **for** jest (praktycznie) równoważna instrukcji:

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

Różnica: jeśli w instrukcji wystąpi **continue**, to wyrażenie w pętli **for** będzie obliczone przed obliczeniem warunku. W pętli **while** nie można pominąć warunku.

Instrukcje skoku

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

W C++ zawsze przy wychodzeniu z zasięgu widoczności następuje niszczenie obiektów automatycznych zadeklarowanych w tym zasięgu, w kolejności odwrotnej do ich deklaracji.

Instrukcja break

- Może się pojawić jedynie wewnątrz pętli lub instrukcji wyboru i powoduje przerwanie wykonywania 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.
- Każda funkcja o typie wyniku innym niż **void** musi zawierać co najmniej jedną taką instrukcję.
- Jeśli typem wyniku funkcji jest **void**, to funkcja może nie zawierać żadnej instrukcji **return**, wówczas koniec działania funkcji następuje po dotarciu sterowania do końca treści funkcji.

Instrukcja goto

- Nie używamy tej instrukcji.

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.

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 jawnie (bo jet przyjmowany domyślnie),
  - **register** tyle co auto, z dodatkowym wskazaniem dla kompilatora, że deklarowana zmienna będzie często używana,
  - **static** to słowo ma kilka różnych znaczeń w C++, tu oznacza, że identyfikator będzie zachowywał swoją wartość pomiędzy kolejnymi wejściami do bloku, w którym jest zadeklarowany,
  - **extern** oznacza, że identyfikator pochodzi z innego pliku, czyli w tym miejscu jest tylko jego deklaracja (żeby kompilator znał np. jego typ, a definicja (czyli miejsce gdzie została przydzielona pamięć) jest gdzie indziej.

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.

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

Literały zmiennopozycyjne

- Mają typ double (o ile nie zmienia tego przyrostek)
- 1.23, 12.223e3, -35E-11,
- Przyrostek f, F (float), l, L (long double),

Literały 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!

Literały 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.

Literały logiczne (typu bool)

- **true**,
- **false**.

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.

Typy

Co można zrobić z typami?

- Typ określa rozmiar pamięci, dozwolone operacje i ich znaczenie.
- Z każdą nazwą w C++ związany jest typ, mamy tu statyczną kontrolę typów.
- 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).

Typy podstawowe cd.

Liczby całkowite bez znaku:

- unsigned char,
- unsigned short int,
- unsigned int,
- unsigned long int.

(część int można opuścić)

Typy podstawowe cd.

Liczby rzeczywiste:

- float,
- double,
- long double.

Typy podstawowe cd.

- W C++ **sizeof**(**char**) wynosi 1 (z definicji),
- Typ wartości logicznych **bool**,
- **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(dest, src);
// Błędny fragment programu (ale kompilujący się bez
// ostrzeżeń) zwn złe położenie prawego, okrągłego 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] (nie można zapisać tego w jednej parze nawiasów kwadratowych).
- W C++ nazwy tablicy można używać jako wskaźnika. Oznacza ona (stały) wskaźnik do pierwszego elementu tablicy. Przekazywanie tablicy jako parametru oznacza przekazanie adresu pierwszego elementu.
- Nie ma operacji przypisania tablic (przypisanie kopiuje tylko wskaźniki).
- 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

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:

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 }.

Kwalifikator const

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

- Można używać **const** przy deklarowaniu wskaźników:
  - **char** $*$p $=$ "ala"; wskaźnik do znaku (napis),
  - **char const** $*$p $=$ "ala"; wskaźnik do stałych znaków (stały napis),
  - **char** $*$ **const** p $=$ "ala"; stały wskaźnik do znaku (napis),
  - **char const** $*$ **const** p $=$ "ala"; stały wskaźnik do stałych znaków (stały napis).

Inicjowanie

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 (jest procedurą).
- 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
```

Wartości domyślne parametrów cd

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();
```

Wartości domyślne parametrów cd

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)) !
}
}
```

Wartości domyślne parametrów cd

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].
- Gdy nie uda się przydzielić pamięci zgłasza wyjątek (bad_alloc).
- Operator **delete**
  - **delete** wskaźnik lub
  - **delete**[] wskaźnik.

Zarządzanie pamięcią cd

- Operator **sizeof**
  - **sizeof** wyr
    - podanego 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

Najlepiej unikać

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 w C++

Klasy - podstawowe pojęcia

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

  specyfikator_klasy:
  nagłówek_klasy { *specyfikacja_skadowych*$_{opt}$ }
  nagłówek_klasy:
  słowo_kluczowe_klasy *identyfikator*$_{opc}$ *klauzula_klas_bazowych*
  słowo_kluczowe_klasy specyfikator_zagnieżdżonej_nazwy *ident*

Przykład klasy - liczby zespolone

Bez klas byłoby tak:

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

Przykład klasy - liczby zespolone cd

Dokładnie to samo można wyrazić używając klas:

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

Przykład klasy - liczby zespolone cd

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 tę wadę, że trudno się zorientować, czym tak naprawdę jest typ Zespolona (trzeba przeczytać cały program, żeby znaleźć wszystkie definicje dotyczące liczb zespolonych).

Operacje w klasie

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();
};
```

Operacje w klasie cd

Zauważmy, że:

- Zwiększyła się czytelność programu: od razu widać wszystkie operacje dostępne na naszym typie danych.
- Zmieniła się liczba parametrów operacji.
- Nie podaliśmy (jeszcze) ani treści operacji, ani nazw parametrów.

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

Implementacja operacji z klasy

```
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 powinny 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 !!!
```

Po co jest potrzebna ochrona danych

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 w przykładzie z mod) 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).

Składowe prywatne i publiczne

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. Jest to zresztą (poza domyślnym trybem dziedziczenia i oczywiście słowem kluczowym) jedyna różnica pomiędzy klasami a strukturami w C++.

Klasa Zespolona z ochroną danych

Zatem teraz mamy następującą deklarację:

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

Klasa Zespolona z ochroną danych - konsekwencje

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 niezainicjowane obiekty?

Czy chcemy mieć niezainicjowane obiekty? Oczywiście nie:

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

Jak temu zaradzić? Można dodać metodę ini(), która będzie inicjować liczbę, ale ... to nic nie daje. Dalej nie ma możliwości zagwarantowania, że zmienna typu Zespolona będzie zainicjowana przed pierwszym jej użyciem.

Konstruktory

Na szczęście w C++ możemy temu skutecznie zaradzić. Rozwiązaniem są konstruktory. Konstruktor jest specjalną metodą klasy. Ma taką samą nazwę jak klasa. Nie można podać 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 (i zwykle tak się robi) 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),
- inicjację obiektu (to nasze zadanie, realizuje je treść konstruktora).

Rodzaje konstruktorów

Wyróżnia się 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).
  - ten konstruktor nie inicjuje składowych typów prostych,
  - dla składowych będących klasami lub strukturami wywołuje ich konstruktory bezargumentowe,
  - jeśli składowa będąca klasą lub strukturą nie ma konstruktora bezargumetowego bądź jest on niedostępny, generowanie konstruktora domyślnego kończy się błędem kompilacji.
- 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!!!

Klasa Zespolona z konstruktorem

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
```

Konsekwencje zdefiniowania konstruktora

Jakie są konsekwencje zdefiniowania konstruktora?

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

Konstruktory a obiekty tymczasowe

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

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

Można tę funkcję wywołać tak:

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

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

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

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

Konstruktor kopiujący w klasie Zespolona

Dla klasy Zespolona nie ma potrzeby definiowania 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(const Zespolona& z){
re = z.re;
```

Ułatwianie sobie życia

Jest zupełnie naturalne, by chcieć używać liczb zespolonych, które są tak naprawdę 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;
}
```

Ułatwianie sobie życia cd

Przedstawione rozwiązanie jest 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;
```

Ułatwianie sobie życia cd

Zdefiniowanie konstruktora 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 (omawiając operatory).

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.

Destruktor w klasie Zespolona

W klasie Zespolona destruktor nie jest potrzebny, ale można 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
}
```

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.

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[1].

[1] Dokładna semantyka **protected** jest nieco bardziej skomplikowana, ale w praktyce nie ma to znaczenia.

Przykładowa podklasa

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

Przykłady użycia

```
 void B::m(){
  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
```

Podsumowanie

- 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.

Uwaga: obiekty podklas, mimo że nie mają bezpośredniego dostępu do prywatnych odziedziczonych składowych, mają także i te odziedziczone składowe.

Przesłanianie nazw

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

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

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

Przesłanianie nazw cd

```
void C::m(){
 a = 1; // Składowa klasy C
}

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

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

To samo dotyczy metod.

Operator zasięgu

- Do odwoływania się do składowych z nadklas służy operator zasięgu.
- Ma on postać ::.
- Przykład użycia:

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

Dostęp do zmiennej globalnej za pomocą operatora zasięgu

```
int i;
void m(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 ?
```

Zgodność typów parametrów

```
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->m();
```

jaka funkcja powinna się wywołać? cwsk pokazuje na obiekt klasy D. Ale kompilator o tym nie wie i wygeneruje kod wywołujący 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**).

Przykłady ilustrujące rodzaje dziedziczenie

Oto przykłady ilustrujące przedstawione reguły:

```
class A{
 public:
  int i;
  // ...
};
class B1: public A{};
class B2: protected A{};
class B3: private A{
void m(B1*, B2*, B3*);
};

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

Odwołania z funkcji

```
void m(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
}
```

Odwołania z podklasy klasy dziedziczącej w sposób chroniony

```
void C2::m(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
}
```

Odwołania z klasy dziedziczącej prywatnie

```
void B3::m(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
}
```

Podsumowanie

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.

Przykład klasy Figura

Rozważmy poniższy (uproszczony) przykład:[2]

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

[2] W tym i w pozostałych przykładach pozwalamy sobie zapisywać identyfikatory z polskimi znakami. Jest to niezgodne ze składnią C++ (taki program się nie skompiluje), ale zwiększa czytelność przykładów, zaś doprowadzenie do zgodności ze składnią C++ jest czysto mechaniczne i nie powinno Czytelnikowi sprawić kłopotu.

Przykład klasy Figura cd

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

Figura::ustaw(int n_x, int n_y){
 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(){
```

Przykład klasy Figura cd

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

Okrąg::Okrąg(int x, int y, int r): Figura(x,y), promień(r){}
```

Przykład klasy Figura cd

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

Metody wirtualne

- Czemu nie działa przykład z Okręgiem?
- Czy instrukcja warunkowa lub wyboru jest tu rozwiązaniem?
- Różnica między metodami wirtualnymi a funkcjami.
- Składnia deklaracji metod wirtualnych.
- W podklasie klasy z metoda wirtualną można tę metodę:
  - zdefiniować na nowo (zachowując sygnaturę),
  - nie definiować.

Przykład z trójpoziomową hierarchią klas cd

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

class B: public A{
};

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

Przykład z trójpoziomową hierarchią klas cd

```
int main(){
  A *p;
  p = new A;
  p->m(3); // A::m()
  p = new B;
  p->m(3); // A::m()
  p = new C;
  p->m(3); // C::m(), bo *p jest obiektem klasy C
```

Przykład z trójpoziomową hierarchią klas cd

```
// Ale:
  // ...
  A a;
  C c;
  a = c;
  a.m(3); // A::m(), bo a jest obiektem klasy A
}
```

Implementacja metod wirtualnych

Implementacja:

- Jest efektywna.
- Np. 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{
 // ...
 virtual void pokaż() = 0;
 virtual 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 zainicjowanie wszystkich warstw. Ponadto klasy mogą mieć składowe również będące obiektami klas - je też trzeba zainicjować w konstruktorze. Na szczęście w podklasie musimy zadbać o inicjowanie jedynie:

- bezpośredniej nadklasy,
- własnych składowych.

tzn. my nie musimy już (i nie możemy) inicjować dalszych nadklas oraz składowych z nadklas. Powód jest oczywisty: to robi konstruktor nadklasy. My wywołując go (w celu zainicjowania bezpośredniej klasy bazowej) spowodujemy (pośrednio) inicjację wszystkich warstw pochodzących z dalszych klas bazowych. Nie musimy inicjować nadklasy, jeśli ta posiada konstruktor domyślny (i wystarczy nam taka inicjacja). Nie musimy inicjować składowych, które mają konstruktor domyślny (i wystarcza nam taka inicjacja).

Inicjacja w hierarchiach klas

Składnia inicjacji: po nagłówku konstruktora umieszczamy nazwę nadklasy (składowej), a po niej w nawiasach parametr(y) konstruktora.
Kolejność inicjacji:

- najpierw inicjuje się klasę bazową,
- następnie inicjuje 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 inicjowane.)

Inicjacja składowych

Czemu ważna jest możliwość inicjowania składowych:

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

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

Inicjacja składowych cd

Rozważmy następujące wersje konstruktora dla B:

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

i

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

Niszczenie obiektu

Kolejność wywoływania destruktorów:

- destruktor w klasie,
- destruktory (niestatycznych) obiektów składowych,
- destruktory klas bazowych.

Ważne uwagi:

- W korzeniu hierarchii klas musi być destruktor wirtualny.
- Uwaga na metody wirtualne w konstruktorach i destruktorach.

Operatory

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.
- Definiowanie operatorów wymaga szczególnej ostrożności.

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**.

Opis cd

- 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.

Opis cd

- 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 kopiującego operatora przypisania, który jest bardziej złożonym przypadkiem).
- 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).

Operatory jednoargumentowe - przykład

```
class X{
 public:
  X operator++(); // przedrostkowy ++a
  X operator++(int); // przyrostkowy 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?

- Najlepiej jako metodę.
- Nie zawsze można:
  - gdy operator dotyczy dwa klas,
  - gdy istotne jest równe traktowanie obu argumentów operatora.

Operator jako metoda - niesymetryczne traktowanie argumentów

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

Operator jako metoda - niesymetryczne traktowanie argumentów

Przy przedstawionych 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.

Kopiujący operator przypisania

- Kopiujący operator przypisania jest czymś innym niż konstruktor kopiujący!
- O ile nie zostanie zdefiniowany przez użytkownika, to będzie zdefiniowany przez kompilator, 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,

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.

Konwersje typów

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

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 >>

Operatory << i >> służą (m.in.) do wczytywania i wypisywania obiektów definiowanej klasy. Zostaną omówione wraz ze strumieniami.

Szablony

Czemu potrzeba w C++ szablonów

- Chcemy mieć ATD,
- W C czy Pascalu nie jest to możliwe,
- Dziedziczenie zdaje się oferować rozwiązanie,
- Przykład - stos elementów dowolnego typu.

Stos: klasa EltStosu

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

Stos: klasa Stos

```
// ------------------
// 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 = 1024);
  ~Stos();

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

Stos: implementacja klasy Stos

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

Stos::Stos(unsigned s){
 size = s;
 top = 0;
 tab = new EltStosu*[size];
}

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);
  top++;
```

Stos: klasa EltStosu

Muszę zdefiniować podklasę reprezentującą wkładane elementy:

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

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

Stos: klasa EltStosu

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;
}
```

Błędy w przedstawionym rozwiązaniu

Niestety tak zapisany program zawiera sporo błędów:

- Nagłówek push wymaga zmiany na

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

Błędy w przedstawionym rozwiązaniu cd

- Uniknięcie kopiowania w implementacji push wymaga zmian także w klasach EltStosu i Liczba na

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

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

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

Błędy w przedstawionym rozwiązaniu - eliminujemy przekazywanie przez wartość

- 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.

Błędy w przedstawionym rozwiązaniu - eliminujemy konstruktor kopiujący

- 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 metodę czysto wirtualną kopia, dającą wskaźnik do kopii oryginalnego obiektu:

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

Trzeba jeszcze tę metodę przedefiniować w klasie Liczba:

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

Błędy w przedstawionym rozwiązaniu - zmieniona treść push

- Treść push jest teraz taka:

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

Błędy w przedstawionym rozwiązaniu - zły typ wyniku pop

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

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

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

Błędy w przedstawionym rozwiązaniu - zmiana treści programu

- 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;
 }
}
```

Błędy w przedstawionym rozwiązaniu - podsumowanie

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).

Deklaracje szablonów

- Szablon klasy - matryca klas
- Składnia (**template**, **typename**, **class**)
- Parametry (typy, wartości proste)

Przykład deklaracji szablonu

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();
};
```

Przykład definicji metod dla szablonu klasy

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];
}

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){
```

Przykład definicji metod dla szablonu klasy

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

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

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

Używanie szablonów

- Składnia użycia szablonu klasy
- Przykład

```
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;
 }
}
```

Używanie szablonów - dalsze przykłady

- 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).

- Parametr szablonu może także być zwykłym parametrem typu całkowitego, wyliczeniowego lub wskaźnikowego

```
template<class T, int i> class A {};
```

- Może też być szablonem.

```
template<template<class T> class U> class A { };
```

Szablony funkcji

- 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 */ }
```

Szablony funkcji

- 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.

Szablony funkcji - sort

- W bibliotece standardowej jest uniwersalna funkcja sortująca sort.
- Pełne jej omówienie wymaga znajomości STLa (nasze końcowe wykłady).
- Uproszczona wersja opisu: argumenty są parą wskaźników.
- Uwaga: zakres sortowany *nie* obejmuje elementu zadanego drugim wskaźnikiem.
- Jako trzeci parametr można podać funkcję porównującą elementy, wpp. użyty będzie operator $<$.
- To sortowanie nie jest stabilne (ale jest też stable_sort).
- Oczekiwany czas to $O(n * lgn)$, pesymistyczny nie gorszy niż $O(n^2)$, zależnie od implementacji.

Sortowanie tablicy liczb zadanej wskaźnikami

```
const int n = 10;
int tab[n];
// ...
ini(tab, tab+n); // Treść w wykładzie
pokaż(tab, tab+n); // Treść w wykładzie
sort(tab, tab+n);
pokaż(tab, tab+n);
// ...
```

Sortowanie tablicy wskaźników do liczb

```
const int n = 10;
int* tab[n];

bool por_wsk(int* p1, int* p2){
  return *p1 < *p2; // Nie <= !!!
}
// ...
ini_wsk(tab, tab+n); // Treść w wykładzie
pokaż_wsk(tab, tab+n); // Treść w wykładzie
sort(tab, tab+n, por_wsk);
pokaż_wsk(tab, tab+n);
// ...
```

Obsługa wyjątków

Wprowadzenie do obsługi wyjątków

Sposoby reagowania na błędne sytuacje:

- komunikat i przerwanie działania programu,
- kod błędu jako wynik,
- globalna zmienna z kodem błędu,
- parametr - funkcja obsługi błędów.

Idea obsługi wyjątków w C++

- Celem jest przezwyciężenie problemów z wcześniejszych rozwiązań.
- Wyjątek rozumiany jako błąd.
- Funkcja zgłasza wyjątek.
- Kod obsługi wyjątku może być w zupełnie innym miejscu programu.
- Szukanie obsługi wyjątku z "paleniem mostów".
- Nie ma mechanizmu powrotu z obsługi wyjątku do miejsca jego zgłoszenia.

Wyjątki - składnia

Składnia instrukcji związanych z obsługą wyjątków:

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

Zgłoszenie wyjątku:

```
throw <wyrażenie>;
```

Wyjątki - semantyka

- Zgłoszenie wyjątku rozpoczyna wyszukiwanie obsługi
- Klauzule **catch** są przeglądane w kolejności ich deklaracji
- Trzy możliwości
  - instrukcje nie zgłosiły wyjątku
  - klauzula wyjątku znaleziona - być może w innym bloku - i wykonana
  - brak pasującej klauzuli

Wyjątki - przykład

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){
```

Wyjątki - przykład cd

```
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
}
```

Wyjątki - semantyka

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

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

Wyjątki

- Obsługa wyjątku może zgłosić kolejny.
- Wyjątek jest uważany za obsłużony z momentem wejścia do klauzuli obsługującej go.
- Można zagnieżdżać wyjątki.

Przekazywanie informacji wraz z wyjątkiem

- Wyjątek jest obiektem.
- Obiekt może mieć swój stan.

Przekazywanie informacji z wyjątkiem - przykład

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

Przekazywanie informacji z wyjątkiem - przykład

```
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

- Klasy wyjątków można łączyć w hierarchie.
- Pozwala to specjalizować obsługę wyjątków.

Hierarchie wyjątków - przykład

```
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. */ }
```

Dodatkowe własności wyjątków

- Wznawianie wyjątku **throw**
- Obsługa dowolnego wyjątku

Wyjątki - przykłady

```
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).

- Problem - zwalnianie zasobów.
- Pomysł:
  - opakowanie zasobów obiektami
  - wykorzystanie mechanizmu wyjątków.

Rozwiązanie 1 (tradycyjne - niewygodne):

Rozwiązanie 1 (tradycyjne - 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):

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
};

void używanie_pliku(const char* np){
 Wsk_do_pliku p(np, "w");
 /* coś z plikiem p */
}
```

Zdobywanie zasobów jest inicjacją

- Zdobywanie zasobów jest inicjacją (RAII - Resource Acquisition Is Initialization)
- Bardzo skuteczna technika w C++
- W innych językach:
  - Java: finilize
  - C#: instrukcja using

Obsługa wyjątków w konstruktorach

```
class X{
 Wsk_do_pliku p;
 Wsk_do_pamięci<int> pam;
 // ...
 X(const char*x, int y): p(x, "w"), pam(r) { /* inicjacja */ }
 // ...
};

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){
```

Specyfikowanie wyjątków w interfejsie funkcji

- Wyjątki są istotną cechą specyfikacji funkcji.
- C++ nie wymusza ich specyfikowania.
- Jeśli są wyspecyfikowane ich sprawdzenie odbywa się podczas wykonania programu.
- Specyfikowanie braku wyjątków i ich niezgłaszania.

Specyfikowanie wyjątków w interfejsie funkcji - przykład

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

Jest to równoważne napisaniu:

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