Matematyka stosowana

Programowanie obiektowe i C++

Janusz Jabłonowski
janusz@mimuw.edu.pl
http://janusz.mimuw.edu.pl

Uniwersytet Warszawski, 2012

Streszczenie. Wykład przedstawia podstawy programowania obiektowego oraz programowania w języku C++.

Wersja internetowa wykładu:
http://mst.mimuw.edu.pl/lecture.php?lecture=poc

(może zawierać dodatkowe materiały)

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego.

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Skład w systemie LaTeX, z wykorzystaniem m.in. pakietów beamer oraz listings. Szablony podręcznika i prezentacji: Piotr Krzyżanowski; koncept: Robert Dąbrowski.

Spis treści

Wprowadzenie

1. Wprowadzenie

Dodatkowe materiały do tych zajęć są udostępniane uczestnikom za pomocą platformy Moodle.

http://moodle.mimuw.edu.pl

W szczególności wszystkie informacje organizacyjne są sukcesywnie zamieszczane na wydziałowym Moodle'u.

1.1. Założenia wstępne

Materiały do niniejszego wykładu składają się z dwu części:
— podręcznika (plik wykład.pdf),
— slajdów (plik slajdy.pdf).
Podręcznik jest zbudowany na bazie slajdów, co oznacza, że treść slajdów jest także zawarta w podręczniku, a poszczególne opisy z podręcznika komentują, uzupełniają i rozszerzają materiał przedstawiony w slajdach.

Wykład został podzielony na części tematyczne, które nie zawsze odpowiadają 90-minutowym jednostkom zajęć. Doświadczenia przy prowadzeniu zajęć według tego wykładu pozwalają zaproponować następujący podział materiału na poszczególne wykłady:
1. Wstęp do programowania obiektowego.
2. Wstęp do C++: historia, założenia, instrukcje proste.
3. Wstęp do C++: instrukcje złożone.
4. Wstęp do C++: typy.
5. Wstęp do C++: funkcje, zarządzanie pamięcią, struktura programu.
6. Klasy.
7. Dziedziczenie: wstęp.
8. Dziedziczenie: hierarchie klas, konstruktory i destruktory w hierarchiach, polimorfizm.
9. Operatory.
10. Klasówka.
11. Wyjątki.
12. Szablony.
13. STL: wstęp.
14. STL: dalsze omówienie.
15. Strumienie.

Gdyby udało się wygospodarować dodatkowy wykład, to warto by było go poświęcić na pokazanie przykładowego innego języka programowania wspierającego paradygmat obiektowy (więcej na ten temat dalej w tym rozdziale).

Na początku zajęć można przeprowadzić ankietę, pozwalającą lepiej się zorientować w umiejętnościach studentów. Oto przykład takiej ankiety (chodzi o policzenie osób należących do poszczególnych kategorii, niektóre kategorie nie są rozłączne):

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

Ten wykład zakłada, że jego uczestnicy (zgodnie z programem zajęć na wydziale MIM UW) przeszli podstawowy kurs programowania w jakimś języku wysokiego poziomu (np. w Pascalu). Nie jest natomiast wymagana znajomość języka C. Nie zakłada się tu również wcześniejszej znajomości jakiegoś języka obiektowego. Gdyby na pytanie o wcześniejsze programowanie część uczestników odpowiedziała negatywnie, przeprowadzenie niniejszych zajęć w 60 godzin byłoby wysoce niewskazane. Nauczenie się programowania w małej skali (stosowania pętli, rekurencji, wskaźników, tablic itp.) wymaga zdecydowanie więcej czasu niż pięć spotkań proponowanych w niniejszym wykładzie.

1.2. Cele wykładu

Niniejszy wykład służy osiągnięciu dwu celów.

Pierwszym celem jest nauczenie programowania obiektowego. Cel ten jest realizowany przez przedstawienie ogólnych założeń i zasad tego paradygmatu programowania, a następnie pokazanie ich realizacji w przykładowym języku programowania.

Drugim celem jest nauczenie programowania w języku C++. Wykład omawia wszystkie istotne elementy tego złożonego języka, z szablonami włącznie. Z zasady nie zawiera natomiast informacji o bibliotekach do języka C++ (choć niektóre z nich, np. Boost, są niezwykle interesujące). Jedynym wyjątkiem jest STL, ale włączenie go do wykładu wynika z tego, że STL stał się częścią biblioteki standardowej C++. Pominięte są te konstrukcje, które są typowe tylko dla C lub mało istotne (jak np. `printf` czy unie) lub są powszechnie uważane za niebezpieczne i przestarzałe (takie jak instrukcja skoku).

1.3. Sposób prowadzenia zajęć

Sprawy organizacyjne.
— Rejestracja.
— Godziny wykładu.
— http://moodle.mimuw.edu.pl.
— Co umiemy.
— Co będzie na wykładach.
— Zasady zaliczania.

Zajęcia z programowania są z natury rzeczy zajęciami praktycznymi, więc nie da się ich przeprowadzić bez laboratorium komputerowego. Na wydziale MIM UW te zajęcia są prowadzone w formie semestralnego wykładu (15 tygodni). W każdym tygodniu odbywają się dwie godziny zajęć wykładowych i dwie laboratoryjno-ćwiczeniowych. Ta liczba godzin zajęć jest wystarczająca (choć mogłaby być większa), natomiast można też zorganizować te zajęcia z dwiema godzinami ćwiczeń tablicowych i dwiema godzinami laboratoriów w każdym tygodniu. Klasówkę lepiej zorganizować na wykładzie, przede wszystkim dlatego, że praktyka pokazuje,

że do opanowania przedstawionego tu materiału potrzeba dużo ćwiczeń. Gdyby można było zwiększyć liczbę godzin przewidzianych na realizację niniejszych zajęć, to na pewno należałoby te godziny przede wszystkim przeznaczyć na dodatkowe ćwiczenia laboratoryjne.

1.4. Plan wykładu

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].

Wykład zaczyna się od wprowadzenia ogólnych zasad programowania obiektowego. Następnie zaczynają się zajęcia z C++. Ponieważ nie zakłada się znajomości C najpierw przedstawiona jest imperatywna część C++, czyli te elementy tego języka, które nie używają obiektowości. W pewnym uproszczeniu można to nazwać elementami C w C++, ale ten wykład w ogóle nie uwzględnia tych konstrukcji języka C, które w C++ są zbędne lub nieużywane, jak na przykład funkcji *printf*, zamiast której od początku używany jest operator << (na początku pomijając bardziej zaawansowane zagadnienia z nim związane, jak np. sposób definiowania go).

Ponieważ pierwszym tematem tego wykładu jest programowanie obiektowe bardzo cenne byłoby pokazanie oprócz języka C++ jakiegoś innego języka obiektowego. Jest to niestety niezwykle trudne ze względu na czas - tak programowanie obiektowe jak i język C++ są bardzo obszernymi tematami, więc zwykle nie udaje się wygospodarować dodatkowego wolnego wykładu. Gdyby jednak się udało, to warto by było pokazać jeden z wymienionych języków ze względu na podane jego cechy:
— Smalltalk: czysta obiektowość i kompletnie odmienne podejście autorów do tworzonego języka niż w C++ (inna realizacja kapsułkowania, automatyczne zarządzanie pamięcią, integracja ze środowiskiem wykonania).
— Delphi: niezwykła łatwość tworzenia programów z interfejsami graficznymi, wykorzystanie znajomości Pascala wśród studentów, istotnie odmienna składnia niż w C++.
— Java: inna realizacja obiektowości, automatyczne zarządzanie pamięcią, bezpłatny dostęp i powszechność implementacji i narzędzi.
— C#: połączenie łatwości tworzenia interfejsów graficznych z Delphi i modelu obiektowości z Javy.

Oczywiście powyższa lista nie jest kompletna, kolejnymi ciekawymi kandydatami są takie języki jak Python czy Ruby.

1.5. Zaliczanie

Zaliczenie ćwiczeń:
— obecności,
— klasówka,
— program zaliczeniowy.

Egzamin na koniec wykładu:
— trzeba będzie napisać fragment programu w C++,
— 1,5 - 2 godz.

Ponieważ część praktyczna jest niezwykle istotna na tym wykładzie, konieczne jest uwzględnienie jej w końcowej ocenie. Na wydziale MIM UW zwykle zaliczenie polega na napisaniu klasówki w okolicy połowy semestru, programu zaliczeniowego (pisanego poza zajęciami) i egzaminu. Przy czym zaliczenie programu jest konieczne dla otrzymania pozytywnej oceny końcowej. Oczywiście zamiast jednego dużego programu można wymagać wielu małych, ale kluczowe jest, by do zaliczenia tego przedmiotu było wymagane zdobycie praktycznej umiejętności programowania. Oczywiście nie jest istotne, by studenci nauczyli się na pamięć składni C++, dlatego tak na klasówce jak i na egzaminie powinni mieć dostęp do wszelkich materiałów drukowanych (notatki z zajęć, książki).

1.6. Wybór języka programowania

Czemu C++?

— Bardzo popularny.
— Dostępny w wielu implementacjach.
— Łatwo dostępna literatura.

Wykład z programowania musi być oparty na jakimś narzędziu umożliwiającym praktyczne przećwiczenia nabytej wiedzy. W przypadku programowania obiektowego naturalnym wyborem jest jakiś obiektowy język programowania. W tym wykładzie wybrano jednak C++, czyli język hybrydowy. Powodem jest to, że jest to bardzo popularny język, którego implementacje istnieją dla praktycznie wszystkich platform. Istnieje wiele środowisk (IDE) do tworzenia programów w C++, a literatura, tak drukowana jak i internetowa, jest przebogata. Bardzo ważnym powodem wyboru C++ było również to, że znajomość tego języka jest bardzo pożądana na innych zajęciach prowadzonych na MIM UW.

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.

Należy jednak też wskazać wady języka C++ jako narzędzia do nauki programowania obiektowego. Po pierwsze, pamiętajmy o tym, że istnieje wiele różnych języków obiektowych, nie jest więc tak, że tylko C++ stosuje się w programowaniu obiektowym. Nie jest to też najlepszy język obiektowy. Z dwóch powodów. Po pierwsze nie jest językiem czysto obiektowym (takim jak np. Smalltalk, czy praktycznie czysto obiektowym, tak jak Java czy C#), jest językiem hybrydowym. Można w nim programować tylko imperatywnie, można tylko obiektowo. Dla niektórych ta właśnie cecha będzie zresztą największą zaletą C++. Po drugie, określenie "najlepszy język" jest z natury rzeczy mylące, gdyż fałszywie sugeruje istnienie bezwzględnej gradacji języków programowania (liniowego porządku mówiąc po ludzku), która byłaby powszechnie akceptowana. Innymi słowy - po prostu nie ma najlepszego języka programowania obiektowego, aczkolwiek zapewne wielu programistów ma swój ulubiony język programowania i uważa go za najlepszy dla siebie.

Warto też na samym początku zauważyć, że C++ nie jest językiem łatwym. Wynika to przede wszystkim ze sposobu jaki powstał - jako rozszerzenie innego, dużego i wcale nie tak prostego, języka programowania - C. C++ zawiera bardzo wiele ciekawych (i zwykle niebanalnych) mechanizmów, a konieczność pogodzenia zgodności z imperatywnym (i dość niskopoziomowym) językiem C mechanizmów programowania obiektowego spowodowała, że niektóre konstrukcje C++ nie są tak zgrabne, jak by mogły być w języku tworzonym od nowa.

Czego nie będzie na wykładach

— Opisu poszczególnych implementacji C++.

— Opisu poszczególnych bibliotek i rozszerzeń języka C++.

Powstało tak wiele implementacji języka C++, że nie sposób ich wszystkich wymienić na wykładzie. Co więcej, często te implementacje dostarczają pewnych specyficznych dla siebie udogodnień. A ponieważ sam język podlega rozwojowi (w chwili pisania tego tekstu ciągle oczekiwana jest nowa wersja standardu C++), często jest tak, że poszczególne implementacje różnią się między sobą stopniem zaimplementowania nowego (lub przyszłego) standardu języka. Z oczywistych powodów na tym wykładzie nie omawiamy tych zagadnień. Koncentrujemy się wyłącznie na obowiązującym standardzie C++. Oczywiście laboratorium wymaga użycia jakiejś konkretnej implementacji, ale na wykładzie nie czynimy żadnych założeń na temat tego, jaka implementacja została wybrana. Również nie omawiamy tu żadnej z licznych bibliotek stworzonych dla języka C++. Sam język C++ ze standardową biblioteką jest bardzo obszernym tematem, wymagającym poświęcenia mu całego semestru zajęć. Jednocześnie jest to bardzo bogate narzędzie, pozwalające na pisanie bardzo zaawansowanego, profesjonalnego oprogramowania.

2. Wstęp do programowania obiektowego

Wstęp do programowania obiektowego

2.1. 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 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.

2.2. Pojęcia programowania obiektowego

— 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 (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.

— 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.

2.3. 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.

— 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).

2.4. Podsumowanie zalet i wad

— 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).

— 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).

— 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).

— 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).

— 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).

3. Podstawy C++: instrukcje

Podstawy C++: instrukcje

3.1. 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).

Historia C++ sięga odległych czasów - Algolu 60, języka który stał się pierwowzorem dla większości współczesnych języków. Jednym z języków powstałych na bazie Algolu-60 była Simula-67, czyli pierwszy język obiektowy. Wydaje się, że Simula-67 wyprzedziła swoje czasy - idea obiektowości pod koniec lat 60-tych nie podbiła informatycznego świata. Tym nie mniej, idee rodzącego się paradygmatu programowania zaczęły przenikać do świata akademickiego, Simula-67 była używana do kształcenia studentów w niektórych krajach. Tak się złożyło, że jednym ze studentów informatyki, którzy mieli okazję poznać ten język był Duńczyk Bjarne Stroustrup. Gdy po studiach przeniósł się do Stanów Zjednoczonych i tam pracował programując w C, zauważył, że bardzo mu brakuje w pracy narzędzi znanych mu z Simuli. Postanowił dodać do C makropolecenia, które by pozwalały programować w C używając pojęć programowania obiektowego. Tak powstał język C z klasami. To rozwiązanie okazało się na tyle dobre, że szybko zdobyło popularność najpierw wśród znajomych Bjarne'a Stroustrupa, później krąg użytkowników znacznie się powiększył. Popularność tego rozwiązania zaowocowała stworzeniem pełnoprawnego (a więc nie będącego tylko zestawem makropoleceń do kompilatora innego języka) języka programowania C++. Nazwa pochodzi od operatora ++ występującego w C, oznaczającego zwiększanie wartości zmiennej. W ten sposób podkreślono, że ten nowy język jest rozszerzeniem języka C. Język C++ cały czas jest rozwijany. Obecna wersja standardu została zatwierdzona w 1998 roku. Na tej wersji oparty jest niniejszy wykład. Obecnie trwają prace nad nową wersją standardu języka, gdzie rozważa się wiele interesujących rozszerzeń, ale zakończenie tych prac opóźnia się i nie jest jasne, które z proponowanych rozszerzeń zostaną ostatecznie zaakceptowane.

3.2. 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).

W tej części wykładu zajmujemy się nieobiektową częścią C++. Nie oznacza to, że ta część jest poświęcona opisowi języka C. Wprawdzie C++ powstało jako rozszerzenie języka C i zapewnia niemal pełną zgodność z tym językiem, to jednak wiele rzeczy zapisuje się w C++ inaczej niż w C (na przykład czytanie i wypisywanie). W naszym wykładzie zajmujemy się tylko językiem C++, zatem ignorujemy wszelkie konstrukcje C nieużywane w C++.

Zanim zaczniemy dokładnie opisywać poszczególne konstrukcje języka C++ podamy kilka podstawowych informacji o tym języku. C++ jest językiem kompilowanym, co oznacza, że każdy program, zanim zostanie uruchomiony, musi być przetłumaczony na język maszynowy za pomocą kompilatora. Takie rozwiązanie ma liczne zalety. Po pierwsze, kompilator jest w stanie podczas kompilacji wykryć (proste) błędy w programie - dzięki temu zwiększa się szansa na napisanie poprawnego programu. Po drugie, takie podejście pozwala na znacznie efektywniejsze wykonywanie programów.

C++ jest też językiem ze statyczną kontrolą typów. Oznacza to, że każda zmienna (parametr, funkcja itp.) musi być zadeklarowana przed użyciem, a jej deklaracja musi określać jej typ, czyli zbiór wartości, które może przyjmować. Takie podejście jest typowe dla języków kompilowanych i pozwala kompilatorowi wykrywać znacznie więcej błędów na etapie kompilacji (jak np. próba dodania liczby do napisu) oraz generować efektywniejszy kod.

Programy pisane na ćwiczeniach zwykle mieszczą się w jednym pliku. Oczywiście nie jest to typowe dla większych programów. Duże programy należy dzielić na mniejsze części, tak by móc łatwiej nimi zarządzać. Na przykład możemy podzielić pracę tak, by różne pliki były tworzone przez różnych programistów (to zresztą ma miejsce nawet w pozornie jedno-plikowych programach z zajęć, korzystają one przecież ze standardowych bibliotek). W C++ nie ma wprawdzie pojęcia modułu czy pakietu jak w wielu innych językach, ale można dowolnie dzielić tworzony program na pliki.

3.3. 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.

Opisując poszczególne instrukcje C++ będziemy posługiwać się specjalną notacją, dość typową. Pozwala ona rozróżniać elementy języka od metaelementów oraz opisywać opcjonalność lub wielokrotne powtarzanie konstrukcji.

3.4. Instrukcje języka C++

Instrukcje języka programowania stanowią o tym, co da się w tym języku wyrazić. Zestaw instrukcji C++ jest dość typowy. Zamieszczamy tu informacyjnie listę wszystkich instrukcji, w dalszej części wykładu będziemy omawiać je szczegółowo.

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

3.4.1. 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.

Jest to jedna z najważniejszych instrukcji w C++. Spełnia rolę trzech różnych instrukcji z Pascala (pustej, przypisania i wywołania procedury)! A jednocześnie łatwo ją zapisać omyłkowo (aczkolwiek w pełni poprawnie składniowo) tak, by nic nie robiła. Przyjrzyjmy się jej więc dokładniej.

Najprostsza jej postać to sam średnik (przy pominiętym wyrażeniu). W tej postaci pełni rolę instrukcji pustej. Wbrew pozorom czasem instrukcja pusta bywa przydatna, na przykład przy zapisywaniu prostych pętli (pętle omówimy wkrótce).

```
// Wyszukanie pierwszego zera w tablicy t
int i;
for(i=0; i<n && t[i]!=0; i++)
;
```

Ważne żeby zapamiętać, że w C++ instrukcja pusta nie jest pusta (składa się ze średnika).

Typowe zastosowanie instrukcji wyrażeniowej polega na zapisaniu w niej wyrażenia przypisania i zakończeniu go znakiem średnika. Wyliczenie takiej instrukcji polega na wyliczeniu wartości wyrażenia i ... zignorowaniu tej wartości. Kiedy to ma sens? Wtedy, gdy wyliczenie wyrażenia ma efekty uboczne. Podstawowym wyrażeniem, którego wyliczenie ma efekt uboczny jest wyrażenie przypisujące. Na przykład i=0 jest takim wyrażeniem. Zrobienie z niego instrukcji wyrażeniowej jest bardzo proste - wystarczy dodać średnik.

```
i = 0;
```

W C++ nie ma instrukcji przypisania, jest wyrażenie przypisujące. Nie jest to wielka różnica (wszak instrukcja wyrażeniowa zamienia dowolne wyrażenie na instrukcję), ale często wygodnie jest móc zapisać przypisanie jako wyrażenie. Wartością wyrażenia przypisującego jest przypisywana wartość. Zatem chcąc przypisać tę samą wartość do kilku zmiennych, można to zrobić w C++ jedną instrukcją:

```
i = j = k = 0;
```

Inną sytuacją, gdy traktowanie przypisania jako wyrażenia jest wygodne, jest zapamiętywanie wartości użytej w warunku logicznym. Załóżmy, że operacja getc() daje kolejny znak z wejścia, oraz że chcemy pominąć kolejne znaki odstępu i zapamiętać pierwszy, różny od odstępu znak. Możemy to zapisać tak:

```
while((c = getc()) != '␣'); // Instrukcja pusta jako treść pętli
```

Trzecia ważna forma instrukcji wyrażeniowej służy wywołaniu funkcji, które chcemy traktować jako procedury. W C++ są tylko funkcje, nie ma procedur, ale nie ma też obowiązku odczytywania wartości wyniku funkcji, można więc wywołać funkcję jako procedurę. Co więcej można zadeklarować funkcję, która nie daje wyniku - czyli funkcję pełniącą rolę procedury. Załóżmy, że mamy taką bezparametrową funkcję-procedurę o nazwie wypisz. Jej wywołanie można zapisać następująco:

```
wypisz();
```

Efektem tej instrukcji wyrażeniowej będą efekty działania funkcji wypisz.

Oczywiście nie każde wyrażenie warto zamieniać na instrukcję wyrażeniową. Oto całkowicie poprawny i całkowicie bezużyteczny przykład instrukcji wyrażeniowej - wyliczenie podanego tu wyrażenia nie ma żadnych efektów ubocznych, więc równie dobrze można by tu wstawić instrukcję pustą lub po prostu usunąć tę instrukcję.

```
13;
```

Powszechnym błędem jest zapominanie o podaniu pustych nawiasów po nazwie wywoływanej funkcji bezargumentowej. Sama nazwa funkcji jest poprawnym wyrażeniem w C++ (jego wartością jest wskaźnik do funkcji), ale wyliczenie takiego wyrażenia nie daje żadnych efektów ubocznych, więc jest całkowicie bezużyteczne w instrukcji wyrażeniowej.

```
wypisz; // powinno być wypisz();
```

Na szczęście większość kompilatorów generuje w takiej sytuacji ostrzeżenie.

Powiedziawszy o zaletach traktowania przypisania jako wyrażenia koniecznie musimy jednocześnie ostrzec o niebezpieczeństwach związanych ze stosowaniem efektów ubocznych. Choć sam język tego nie zabrania pamiętajmy, żeby nigdy nie nadużywać efektów ubocznych. Zilustrujemy to ostrzeżenie przykładami:

```
i = 1;
t[i] = i = 0; // przypisanie 0 do t[1] czy t[0]?
f(i=1,i=2); // jaką wartość ma i po wywołaniu funkcji f?
```

3.4.2. 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 etykietowana to po prostu dowolna instrukcja poprzedzona etykietą. Charakterystyczne dla C++ jest to, że owa etykieta może przyjąć jedną z kilku form.

Po pierwsze może być po prostu identyfikatorem. W tej postaci instrukcja etykietowana służy jako wskazanie miejsca, do którego należy skoczyć w instrukcji **goto** (instrukcji skoku). Ponieważ instrukcja skoku jest powszechnie uważana za szkodliwą i od dawna nie występuje w nowych językach programowania pomijamy ją (i tę postać instrukcji etykietowanej) w naszym wykładzie.

Druga i trzecia postać instrukcji etykietowanej dotyczy instrukcji wyboru (instrukcji oznaczonej w C++ słowem kluczowym **switch**). Dlatego omówienie tych instrukcji znajduje się w omówieniu instrukcji wyboru. Tu zaznaczmy tylko, że obie te formy instrukcji etykietowanej mogą występować wyłącznie wewnątrz instrukcji wyboru, a wyrażenie stałe występujące po słowie kluczowym **case**

3.4.3. Instrukcja złożona (blok)

blok:
$\quad$ 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 złożona (zwana czasem blokiem) służy do grupowania wielu instrukcji w jedną oraz do wyznaczania zasięgu deklaracji zmiennych. Grupowanie instrukcji jest często potrzebne ze względu na składnię języka, która w wielu miejscach (np. jako treść pętli) wymaga pojedynczej instrukcji. Jeśli chcemy w takim miejscu umieścić kilka (lub więcej) instrukcji, to musimy użyć instrukcji złożonej. Ta instrukcja jest pomocna także wówczas, gdy chcemy zadeklarować zmienną, która ma być widoczna tylko w najbliższym otoczeniu deklaracji.

Warto zwrócić uwagę, że składnia C++ nie wymaga żadnych separatorów pomiędzy poszczególnymi instrukcjami - każda instrukcja C++ kończy się średnikiem lub prawym nawiasem klamrowym, nie ma więc potrzeby stosowania dodatkowych separatorów. W instrukcji złożonej może występować dowolna liczba instrukcji składowych (w szczególności może ich być 0, ale nie jest to użyteczny przypadek).

Wykonanie instrukcji złożonej polega na wykonaniu po kolei instrukcji składowych (zostaną wykonane wszystkie, o ile w czasie ich wykonywania nie pojawi się instrukcja zmieniająca przepływ sterowania w programie - taka jak np. **return**;).

Ważne jest zauważenie, że w C++ deklaracja zmiennej (można również deklarować klasy, struktury, wyliczenia lub nazywać typy za pomocą **typedef** ale te deklaracje zwykle są globalne) też jest instrukcją. Czyli można zadeklarować zmienną lokalnie - wewnątrz bloku. Taka zmienna jest widoczna od miejsca deklaracji do końca bloku. Jeśli przy deklaracji podano inicjalizator, to przy zmiennych automatycznych będzie on wykonywany za każdym razem, gdy sterowanie dojdzie do tej deklaracji, zaś dla zmiennych statycznych deklaracja zostanie wykonana tylko raz, przed wejściem do bloku. Lokalnie zadeklarowana zmienna automatyczna jest niszczona, gdy

sterowanie opuszcza blok. Z powyższych rozważań wynika, że zmienne lokalne w bloku mogą być deklarowane w dowolnym jego miejscu (nie koniecznie na początku, choć zwykle tak się dzieje).

Oczywiście nie można deklarować w jednym bloku dwu zmiennych o tej samej nazwie. Jeśli na zewnątrz bloku jest zadeklarowany identyfikator użyty w deklaracji lokalnej, to jest on przesłonięty od miejsca deklaracji do końca bloku. Oto przykład:

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

3.4.4. 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 warunkowa służy warunkowemu wykonaniu instrukcji. Wykonanie instrukcji warunkowej zaczyna się od obliczenia wartości wyrażenia i niejawnego przekształcenia jej do typu **bool**. Jeśli tak otrzymana wartością jest **true**, to wykonywana jest pierwsza podinstrukcja, jeśli zaś tą wartością jest **false** to, o ile jest część **else**, wykonywana jest druga podinstrukcja.

```
if (i>=0)
 cout << "nieujemne";
else
 cout << ujemne";
```

Ciekawostką (odziedziczoną po C) jest to, że typem wyrażenia pełniącego rolę warunku może być typ liczbowy lub nawet wskaźnikowy. Jeśli wartość takiego wyrażenia jest różna od zera (dla wskaźników oznacza to wskaźnik o wartości różnej od NULL), to jest ono traktowane jako warunek prawdziwy (a w przeciwnym przypadku jako fałszywy).

```
if (i)
 i = 0;
```

```
if (p)
 i = *p;
else
 i = 0;
```

Jeśli jedna instrukcja warunkowa jest zagnieżdżona w drugiej i nie każda z nich ma część **else**, to powstaje problem składniowy, z którą instrukcją warunkową powiązać tę część **else**.

```
if (i>=0)
 if (i>0)
   cout << "dodatnie";
 else // do której instrukcji if?
  cout << "??";
```

W C++ przyjęto, tak jak prawdopodobnie w każdym języku programowania, że **else** jest doklejany do bliższej mu poprzedzającej instrukcji **if**. Zatem ostatni przykład można uzupełnić następująco.

```
if (i>=0)
 if (i>0)
   cout << "dodatnie";
 else // (i>=0) && !(i>0)
  cout << "równe_zero";
```

Oczywiście gdyby chodziło o odwrotne dowiązanie części **else** łatwo to osiągnąć za pomocą instrukcji złożonej.

```
if (i>=0)
 {if (i>0)
   cout << "dodatnie";}
else // !(i>=0)
 cout << "ujemne";
```

Subtelności związane z deklarowaniem zmiennych w warunku lub w przypadku zapisania instrukcji deklaracji jako podinstrukcji instrukcji warunkowej pomijamy.

3.4.5. 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, 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**!.
— Wyrażenie 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.

Czasem wybór jednej z dwu operacji nie wystarcza, wtedy często można zastosować instrukcję wyboru **switch**. Nie jest ona niestety tak wygodna jak np. w Pascalu. Jej wykonanie polega na wyliczeniu wartości wyrażenia (musi ono być typu całkowitego lub wyliczeniowego), a następnie na wyszukaniu w instrukcji składowej (która praktycznie zawsze jest instrukcją złożoną) instrukcji z etykietą **case** i wyrażeniem o szukanej wartości. Jeśli taka instrukcja się znajdzie (może być co najwyżej jedna taka instrukcja), to sterowanie przechodzi bezpośrednio za tę etykietę. Jeśli takiej etykiety nie ma, a jest instrukcja z etykietą **default**, to sterowanie przechodzi do niej, w przeciwnym przypadku sterowanie przechodzi za instrukcję **switch**.

Przy wyborze podinstrukcji pasującej do wyrażenia kolejność instrukcji składowych nie ma znaczenia. W szczególności instrukcja z etykietą **default** nie musi być ostatnia (choć zwykle, dla czytelności, umieszcza się ją na końcu).

Jeśli dla kilku wartości wyrażenia ma być wykonany ten sam zestaw czynności, to te instrukcje poprzedzamy ciągiem etykiet **case**. Na przykład chcąc wykonać tę samą akcję dla wartości zmiennej i tak -1 jak i 1 można zapisać stosowną instrukcję wyboru następująco:

```
switch (i){
  case −1: case 1: i =− i;
}
```

Nie ma niestety możliwości ani pominięcia powtarzających się słów kluczowych **case**, ani podania przedziału wartości, dla których ma się wykonać operacja. To ostatnie powoduje, że jeśli chcemy wybrać wykonywane operacje dla przedziałów wartości np. 0..99, 100..199, 200..299, to instrukcja wyboru staje się bezużyteczna (trzeba wtedy wybrać instrukcję warunkową). Jest to dość rozczarowujące w porównaniu np. z Pascalem.

Tym co zapewne powoduje najwięcej kłopotów przy używaniu instrukcji **switch** jest jej nieintuicyjna semantyka (oparta na semantyce instrukcji skoku). Otóż po wybraniu odpowiedniej etykiety i wykonaniu instrukcji nią opatrzonej sterowanie *nie* przechodzi za instrukcję **switch**, lecz przechodzi do kolejnych instrukcji znajdujących się po niej w instrukcji **switch**!. W kolejnym przykładzie dla znaku ch mającego wartość 'C' wykonają się oba przypisania, czyli wynik końcowy będzie taki sam, jak dla ch == 'b'.

```
switch (ch){
  case 'C': cout << "stopnie␣Celsjusza";
  case 'K': cout << "stopnie␣Kelvina";
}
```

Jeśli chcemy, żeby ta instrukcja przypisywała różne wartości dla znaków 'a' i 'b' musimy zastosować instrukcję **break**;, tak jak to pokazano w następnym przykładzie.

```
switch (ch){
  case 'a': i = 1; break;
  case 'b': i = 2; break;
}
```

Pisanie **break**; na końcu ostatniej podinstrukcji w **switch** nie jest konieczne, ale jest dobrym zwyczajem, przydającym się, gdy za jakiś czas dopisuje się kolejną podinstrukcję na końcu instrukcji **switch**.

4. Instrukcje złożone

Instrukcje złożone

4.1. Instrukcje pętli

4.1.1. Pętle while i do

instrukcja_pętli:
while (warunek) instrukcja
do instrukcja **while** (warunek);
pętla_for
warunek:
wyrażenie

Podstawowym narzędziem do opisywania powtarzania wykonywania operacji jest iteracja. W C++ do jej zapisu służą aż trzy instrukcje pętli. Są one takie same jak w C, dwie pierwsze z nich są podobne do pętli (odpowiednio) **while** i **repeat** z Pascala.

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.

Jak widać semantyka pętli **while** i **do** w C++ jest typowa. Znającym Pascala warto zwrócić uwagę, że dozór w pętli **do** oznacza warunek kontynuowania pętli (a nie kończenia jak w **repeat** z Pascala).

Oto przykłady użycia tych pętli.

Obliczanie największego wspólnego dzielnika dwu liczb naturalnych większych od zera za pomocą odejmowania.

```
while (m!=n)
 if(m>n)
  m=m-n;
else
  n=n-m;
```

Wypisywanie (od tyłu) cyfr liczby naturalnej nieujemnej.

```
do{
 cout << n%10;
 n=n/10;
}
```

```
while (n>0);
```

— 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.

Zaskakującą cechą pętli w C++ (odziedziczoną po C) jest to, że warunek nie musi mieć typu logicznego (w pierwszych wersjach języków C i C++ w ogóle nie było takiego typu). Składnia języka C była tak tworzona, by łatwo zapisywało się w niej typowe programy, natomiast twórcy C nie przykładali dużej wagi do czytelności programów w tym języku. Język był przeznaczony dla bardzo zaawansowanych programistów, np. tworzących systemy operacyjne, w związku z tym twórcy języka uznali, że nie warto udawać przed użytkownikami tego języka, że w pamięci komputera są inne rzeczy niż liczby i adresy — rzeczy takie jak na przykład wartości logiczne. Stąd reguła, że każda wartość (liczba lub wskaźnik) różna od zera będzie oznaczała prawdę, a wartość zero fałsz. W wielu przypadkach taka składnia okazywała się poręczna, na przykład przeglądanie listy za pomocą wskaźnika p można zapisać używając dozoru pętli o postaci po prostu p, czyli:

```
while (p)
 p=p->next;
```

(Wskaźniki będą omówione w kolejnych wykładach). Chcąc użyć wartości logicznych należałoby zapisać tę pętlę tak:

```
while (p != NULL)
 p=p->next;
```

Niestety za tę kuszącą zwięzłość zapisu płaci się zmniejszoną czytelnością i większym ryzykiem popełniania błędów. Na przykład pominięcie części warunku przy przeglądaniu listy cyklicznej:

```
while (p) // pominięta część != start
 p=p->next;
```

zamiast

```
while (p != start)
 p=p->next;
```

nie spowoduje żadnego komunikatu ani ostrzeżenia ze strony kompilatora. Niestety nie wszystko to co daje się łatwo zapisać, daje się ławo odczytać.

Deklarowanie zmiennych w dozorach pętli **while** nie ma większego praktycznego znaczenia.

Należy pamiętać, że jeśli w treści pętli będzie zadeklarowana (niestatyczna) zmienna, to przy każdym obrocie pętli będzie ona od nowa tworzona (i usuwana na koniec obrotu pętli). Zatem na przykład nie można w kolejnym obrocie pętli odwołać się do wartości nadanej tej zmiennej w poprzednich obrotach pętli. Na przykład program:

```
i=0;
while(i<n){
 int j = 0;
 j++;
 cout << "* " << j << endl;
 i++;
}
```

Wypisze liczby: 1 1 1 1 1 Ten sam program po dodaniu słowa **static** przy deklaracji zmiennej j:

```
i=0;
while(i<n){
 static int j = 0;
 j++;
 cout << "* " << j << endl;
 i++;
}
```

Wypisze liczby: 1 2 3 4 5.

4.1.2. 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.

4.1.3. 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.

4.2. Dalsze instrukcje zmieniające przepływ sterowania

4.2.1. 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.

4.2.2. 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.

4.2.3. Instrukcja continue

— Może się pojawić jedynie wewnątrz instrukcji pętli i powoduje zakończenie bieżącego obrotu (najciaśniej otaczającej) pętli.

4.2.4. 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.

4.2.5. Instrukcja goto

— Nie używamy tej instrukcji.

4.3. Pozostałe konstrukcje

4.3.1. 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.

4.3.2. 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.

4.3.3. 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.

4.4. Literały

4.4.1. 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,

4.4.2. 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),

4.4.3. 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!

4.4.4. 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.

4.4.5. Literały logiczne (typu bool)

— **true**,
— **false**.

4.4.6. 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.

5. Typy

Typy

5.1. Typy

5.1.1. 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.

5.2. Typy

5.2.1. 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**,
— **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.

5.3. Typy pochodne

5.3.1. 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.
```

Zamieszczony przykład pokazuje niezwykle nieprzyjemny i trudny do zlokalizowania błąd związany ze wskaźnikami. Funkcje strlen i strcpy służą, odpowiednio, do policzenia długości napisu (nie licząc znaku o kodzie 0, oznaczającego koniec napisu) oraz do skopiowania napisu (wraz ze znakiem o kodzie 0).

Prawy okrągły nawias został przez pomyłkę ustawiony za +1 zamiast przed. Prawidłowa postać tego fragmentu programu powinna być taka:

```
char *dest = new char[strlen(src)+1];
// ...
```

Czyli przydzielamy pamięć wystarczającą do zapamiętania wszystkich znaków napisu src oraz znaku oznaczającego koniec napisu (znaku o kodzie 0), stąd to dodanie jedynki. Po przestawieniu nawiasu liczymy długość napisu zaczynającego się od drugiego znaku napisu src, co (o ile src nie było pustym napisem) jest dobrze zdefiniowaną operacją i da wynik o jeden mniejszy niż długość src. Czyli łącznie wyrażenie strlen(src+1) da wynik za mały o 2 (poza przypadkiem pustego src, kiedy to w ogóle nie jesteśmy w stanie przewidzieć jaki będzie wynik). Zatem pamięć przydzielona na napis dest może być za krótka. Może, bo ze względu na specyfikę algorytmów przydzielania pamięci, czasami przydzielają one nieco więcej bajtów, niż było żądane (np. przydzielają pamięć w blokach po 8 bajtów), więc dla niektórych długości napisu src mogą przydzielić wystarczająco dużo pamięci. Jeśli pamięci będzie za mało, to skopiowanie operacją strcpy może zamazać dwa bajty pamięci, należące np. do innej zmiennej.

Zauważmy, że:
— Błąd jest trudy do zauważenia w tekście programu.
— Wystąpienie błędu jest niezwykle trudne podczas testowania - ten błąd może się ujawnić bądź nie w zależności od dość przypadkowych czynników (użyty algorytm przydziału pamięci, długość napisu src, to czy, a jeśli tak to jaka, zmienna zostanie zamazana w wyniku kopiowania.

Zatem nawet dowolnie wiele razy przeprowadzane testy mogą nie wykryć wystąpienia tego błędu. Taki błąd jest idealnym tematem do najgorszych sennych koszmarów programisty!

5.3.2. 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 **int** p[100];

5.3.3. 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.

5.3.4. 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).

5.3.5. 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.

5.3.6. Wyliczenia

— Można definiować wyliczenia np.: **enum** kolor{ czerwony, zielony }.

5.3.7. 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).

5.3.8. 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}
}
```

5.3.9. 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 &.

5.3.10. 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** = 2.0)).

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.

5.3.11. 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.

— 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.

5.3.12. Jawna konwersja typu

Najlepiej unikać

5.3.13. Operatory

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

5.3.14. Preprocesor

— **#include** <...>,
— **#include** "...".

5.3.15. 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[]){ /*... */}.

6. Klasy w C++

Klasy w C++

6.1. Klasy jako struktury

6.1.1. 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 $identyfikator_{opc}$ $klauzula\_klas\_bazowych_{opc}$
słowo_kluczowe_klasy specyfikator_zagnieżdżonej_nazwy $identyfikator_{opc}$ $klauzula\_klas\_bazow$

6.1.2. Przykład klasy - liczby zespolone

W tym rozdziale przyjrzymy się definiowaniu klas na przykładzie klasy Zespolona, której obiekty reprezentują (oczywiście) 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 tę wadę, że trudno się zorientować, czym tak naprawdę jest typ Zespolona (trzeba przeczytać cały program, żeby znaleźć wszystkie definicje dotyczące liczb zespolonych).

6.2. Klasy jako struktury z operacjami

6.2.1. 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();
};
```

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).

6.2.2. 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);
}
```

6.3. Kapsułkowanie

6.3.1. 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 !!!
```

6.3.2. 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).

6.3.3. 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++.

6.3.4. 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();
};
```

6.3.5. 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();
```

6.4. Konstruktory i destruktory

6.4.1. 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.

6.4.2. 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).

6.4.3. 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!!!
  — jest wywoływany niejawnie przy przekazywaniu parametrów do funkcji i przy przekazywaniu wyników funkcji!!!

6.4.4. 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
```

6.4.5. 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.
```

Zatem nie można teraz utworzyć niezainicjowanego obiektu klasy Zespolona!

6.4.6. 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.

6.4.7. 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;
  im = z.im;
}
```

6.4.8. 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;
}
```

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;
  im = i;
}

// .....
```

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

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).

6.4.9. 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.

6.4.10. 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
}
```

6.5. Uwaga o szablonie complex

W tym rozdziale przedstawiliśmy definiowanie klasy na przykładzie liczb zespolonych. Taki przykład wybrano, gdyż reprezentuje dobrze znane pojęcie, jest prosty, a jednocześnie pozwala na pokazanie wielu interesujących własności klas. Warto jednak zaznaczyć, że standardowa biblioteka C++ zawiera własną definicję liczb zespolonych w postaci szablonu complex (szablony omawiamy w osobnym rozdziale). Oto fragment programu używającego szablonu complex.

```
#include <iostream>
#include <complex>

using namespace std;

int main(){
  complex<double> c1;
  complex<double> c2(7.0, 3.5);
  cout << "c1 = " << c1 << ", c2 = " << c2;
  cout << ", c1 + c2 = " << c1+c2 << endl;
}
```

Wynikiem działania tego programu jest wypisanie

```
c1 = (0,0), c2 = (7,3.5), c1 + c2 = (7,3.5)
```

7. Dziedziczenie i hierarchie klas

Dziedziczenie i hierarchie klas

7.1. Dziedziczenie

7.1.1. 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).

7.1.2. 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.

7.1.3. Przykładowa podklasa

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

7.1.4. 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
 i = b.skl6; // OK
}
```

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 **protected**.

7.1.5. Podsummowanie

— 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.

Można powiedzieć, że obiekt klasy pochodnej przypomina kanapkę (czy tort), zawierający warstwy pochodzące ze wszystkich nadklas. W szczególności obiekt b z przykładu 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).

7.1.6. 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();
};
```

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

```
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. Można przesłaniać zwykłe metody (co jest mało użyteczne) oraz można przedefiniowywać metody wirtualne (co ma olbrzymie zastosowanie praktyczne zwn. polimorfizm).

7.1.7. Operator zasięgu

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

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

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::m():

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

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

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

7.1.8. 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
```

7.1.9. 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.

7.1.10. 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**).

7.1.11. 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*);
};
```

```
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
}
```

```
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
}
```

```
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
}
```

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.

7.2. Metody wirtualne

7.2.1. 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);
};
```

Mamy zatem klasę reprezentująca figury geometryczne na ekranie. Obiekty tej klasy znają swoje położenie na ekranie (atrybuty x i y), oraz potrafią:
— zapamiętać nowe położenie, czyli nowe wartości współrzędnych x i y (ustaw),
— narysować się na ekranie w bieżącym położeniu (pokaż),
— wymazać się z ekranu w bieżącym położeniu (schowaj),
— przesunąć się na ekranie z bieżącego do wskazanego parametrami położenia (przesuń).
Ponadto jest zadeklarowany konstruktor. Spróbujmy zapisać implementację tych metod.

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

[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.

```
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(){
 // j.w.
}
```

Udaje się to tylko częściowo. Łatwo można zapisać treść ustaw. Przesuń też wydaje się proste. Natomiast nie wiedząc z jaką figurą mamy do czynienia, nie potrafimy jej ani narysować ani schować. Spróbujmy zatem zdefiniować jakąś konkretną figurę geometryczną, np. okrąg:

```
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){}
```

Oczywiście okrąg oprócz współrzędnych musi też znać swój promień, definiujemy więc stosowny atrybut. W klasie Okrąg wiemy już co należy wyrysować na ekranie, więc stosując operacje z dostępnej biblioteki graficznej pokaż implementujemy np. jako rysowanie okręgu kolorem czarnym, zaś schowaj jako rysowanie kolorem tła. Naturalnie można by zastosować bardziej subtelne algorytmy (szczególnie jeśli chodzi o chowanie), ale dla naszych rozważań nie ma to znaczenia.

Można teraz zacząć działać z opisanymi wcześniej Okręgami. Przykładowy program mógłby wyglądać następująco:

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

Niestety tak pieczołowicie przygotowany zestaw klas zdaje się nie działać. Ale czemu? Każda z operacji czytana osobno wydaje się być poprawna. Problem polega oczywiście na tym, że w treści metody przesuń wywołały się metody Figura::pokaż i Figura::ukryj zamiast Okrąg::pokaż i Okrąg::ukryj. To bardzo nienaturalne, przecież używamy obiektów klasy Okrąg, która ma te operacje prawidłowo zdefiniowane.

7.2.2. Znaczenie metod wirtualnych

— 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.

Problem wynika stąd, że kompilator kompilując treść metody Figura::przesuń nie wie o tym, że kiedyś zostanie zdefiniowana klasa Okrąg. Zatem wywołanie metody pokaż traktuje jako wywołanie metody Figura::pokaż. Wydaje się zresztą, że jest to jedyna możliwość, bo przecież moglibyśmy zdefiniować zaraz klasę Trójkąt, i wtedy chcielibyśmy, żeby w przesuń wywołało się nie Okrąg::pokaż tylko Trójkąt::pokaż.

Jak zaradzić temu problemowi? 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 i wszystko wiedzącą klasę. Dodanie nowej figury wymagałoby zmian w większości metod tego giganta, byłoby więc bardzo trudne i łatwo mogłoby powodować błędy. Ponieważ w programowaniu obiektowym chcemy, 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 językach obiektowych 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 wersja tej metody 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 tę samą liczbę i typy parametrów oraz wyniku. 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.

Przyjrzyjmy się teraz trochę bardziej abstrakcyjnemu przykładowi, ale za to z głębszą hierarchią klas. Tym razem trójpoziomową. Przekonajmy się, czy te same reguły dotyczą dalszych klas pochodnych.

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

class B: public A{
};

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

Mamy trzy klasy A, B, C dziedziczące jedna po drugiej. W klasie B nie ma deklaracji żadnych składowych, zatem obiekty tej klasy mają wirtualną metodę m, odziedziczoną z klasy A. Dziedziczy się zatem także to, czy metoda jest wirtualna. W klasie C metoda m została zdefiniowana ponownie (podmieniona). Zwróćmy uwagę na to, że mimo braku słowa **virtual** w tej klasie i w klasie B metoda m jest wirtualna.

```
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
```

Przedstawiony fragment programu ilustruje niezwykle istotną technikę programowania obiektowego w C++. Wskaźnik p jest zadeklarowany jako wskaźnik do klasy, po której dziedziczy cała hierarchia klas. Następnie na ten wskaźnik przypisywane są adresy obiektów klas dziedziczących (pośrednio lub bezpośrednio) po użytej w deklaracji wskaźnika klasie. Ponieważ obiekty podklas są też obiektami nadklas, tak napisany program nie tylko będzie się kompilował, ale ponadto będzie działał zgodnie z naszymi oczekiwaniami. Tak zapisany fragment programu ilustruje wykorzystanie polimorfizmu.

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

Pamiętajmy jednak, że polimorfizm jest możliwy tylko wtedy, gdy używamy wskaźników bądź referencji. Wynika to stąd, że choć obiekty podklas logicznie są obiektami nadklas, to fizycznie zwykle mają większy rozmiar. Zatem zmienna typu nadklasy nie może przechowywać obiektów podklas. W przedstawionym przykładzie przypisanie a = c; skopiuje jedynie pola zadeklarowane w klasie A i nic więcej. Zmienna a nadal będzie zmienną typu A, czyli będzie przechowywać obiekt typu A.

7.2.3. Implementacja metod wirtualnych

Implementacja:
— Jest efektywna.
— Np. tablica metod wirtualnych.

Mechanizmy kompilacji języków programowania nie wchodzą w zakres tego wykładu, zatem poprzestaniemy tylko na dwóch stwierdzeniach.

Po pierwsze, metody wirtualne są implementowane efektywnie. Zatem nie ma powodów, by unikać ich w swoich programach. Dodatkowy narzut związany z wywołaniem metody wirtualnej odpowiada pojedynczemu sięgnięciu do pamięci. Ponadto (przy typowej implementacji) każdy obiekt przechowuje jeden dodatkowy wskaźnik.

Po drugie, typowa implementacja metod wirtualnych wykorzystuje tablice metod wirtualnych (ang. vtables).

7.2.4. 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;
};
```

7.2.5. 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).

7.2.6. 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.)

Czemu ważna jest możliwość inicjowania 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 inicjacja konstruktorem domyślnym,
— przypisanie.
W drugim tylko jedna:
— tworzenie i inicjowanie konstruktorem kopiującym.
Tak więc druga wersja konstruktora jest lepsza.

7.2.7. 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.

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

W korzeniu hierarchii klas musi być destruktor wirtualny.

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.

8. Operatory

Operatory

8.1. Wprowadzenie

8.1.1. 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.

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.

8.1.2. 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));
```

8.1.3. 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.

8.1.4. 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).

```
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);
}
```

8.1.5. 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.

8.1.6. 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.

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 miałby 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
  Zespolona operator+(const Zespolona&);
};
```

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.

8.1.7. 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, bo teraz już w czasie kompilacji otrzymamy komunikaty o próbie użycia tego operatora poza tą klasą).
  — Automatycznie definiowany kopiujący operator przypisania w podklasie wywołuje operator przypisania z nadklasy, jeśli samemu definiujemy ten operator, to musimy sami o to zadbać.
  — Można zdefiniować także inne (niż kopiujący) operatory przypisania.

8.1.8. 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)
```

8.1.9. Operator indeksowania

Wyrażenie:

```
wyrażenie_proste [ wyrażenie ]
```

interpretuje się jako operator dwuargumentowy. Zatem wyrażenie:

```
x[y]
```

interpretuje się jako:

```
x.operator[](y)
```

8.1.10. 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).

8.1.11. 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

8.1.12. 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.

8.1.13. 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.

8.1.14. Operatory czytania i pisania

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

9. Szablony

Szablony

9.1. Szablony - wstęp

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

Projektując abstrakcyjny typ danych chcemy to zrobić tak, by można było do niego wstawiać obiekty dowolnego innego typu. W Pascalu ani w C 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.

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

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

```
// ------------------------
// 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++;
 }
 else
   blad("przepełnienie stosu");
}

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

// Ze względu na konieczność usuwania wykonuje się 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:

— Nagłówek push wymaga zmiany na

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

— 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");
}
```

— 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 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);
}
```

— Treść push jest teraz taka:

```
void Stos::push(EltStosu& elt){
 if (top < size){
  tab[top] = elt.kopia();
  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 EltStosu, 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;
 }
}
```

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.

9.2. Szablony - deklarowanie

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

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).

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];
}

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);
  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 są
 delete tab[top]; // dwie operacje kopiowania w pop().
 return res;
}

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

9.3. Szablony - używanie

— 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;
 }
}
```

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).

— 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 { };
```

Uwaga: Mechanizm szablonów jest realizowany w podobny sposób do rozwijania makropoleceń, to znaczy, że kompilator przy każdym użyciu szablonu z nowym para metrem 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);
 // ...
}
```

9.4. 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 */ }
```

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.

— 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.

Oczywiście potrzeba sortowania tablicy dowolnych elementów jest tak powszechna, że biblioteka standardowa zawiera stosowne narzędzie. Najpopularniejsze z nich to sort. Powyżej wymieniono najważniejsze cechy tej funkcji, a teraz przypatrzmy się przykładom zastosowania.

```
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);
// ...
```

Tu sort zostało wywołane dla fragmentu tablicy tab zadanego indeksami (w tym przypadku tym fragmentem jest cała tablica tab). Zwróćmy uwagę, że elementu o adresie tab+n nie ma w tablicy. Jest to zgodne z podanym wcześniej opisem znaczenia parametrów funkcji sort.

Oto przykładowa implementacja pomocniczych funkcji użytych w przedstawionym przykładzie.

```
void ini(int* start, int* stop){
  for(int* p=start; p<stop; p++)
    *p = rand() % (stop-start);
}

void pokaż(int* start, int* stop){
  for(int* p=start; p<stop; p++)
    cout << (*p) << ",␣";
  cout << endl;
}
```

Przyjrzyjmy się jeszcze na koniec trudniejszemu przypadkowi. Co zrobić wtedy, gdy tablica zawiera elementy nie dające się porównać operatorem <? Na przykład gdy chcemy posortować liczby zespolone według ich modułu. Albo gdy mamy tablicę nie liczb całkowitych, lecz wskaźników do liczb całkowitych. Porównywanie elementów takiej tablicy - czyli wskaźników - jest składniowo dozwolone w C++, ale oczywiście nie ma sensu w tej sytuacji. Chcemy sortować tablicę według wartości liczb, na które wskazują wskaźniki.

Rozwiązanie polega na podaniu trzeciego parametru, funkcji realizującej operację porównywania. Uwaga: ta funkcja koniecznie musi realizować operację "ostro mniejsze" (<). Podanie zamiast tego funkcji realizującej operację "mniejsze bądź równe" (<=) może powodować na przykład błędy wykonania programu.

```
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);
// ...
```

Zwróćmy uwagę, że trzeci parametr sort nie ma postaci por_wsk() lecz por_wsk. Przekazujemy funkcję, a nie jej wartość.

Oto przykładowa implementacja pomocniczych funkcji użytych w przedstawionym przykładzie.

```
void ini_wsk(int** start, int** stop){
  for(int** p=start; p<stop; p++)
    *p = new int(rand() % (stop-start));
}

void pokaż_wsk(int** start, int** stop){
  for(int** p=start; p<stop; p++)
    cout << (**p) << ", ";
  cout << endl;
}
```

10. Obsługa wyjątków

Obsługa wyjątków

10.1. Obsługa wyjątków - wstęp

10.1.1. 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.

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:
— Wypisać komunikat i przerwać działanie całego programu.
Bardzo brutalne.
— 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.
— 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.
— Wywołać funkcję dostarczoną przez użytkownika (np. jako parametr), obsługującą błędne sytuacje.
Najlepsze z dotąd przedstawionych rozwiązanie. Jego wadą jest to, że każde wywołanie funkcji trzeba obciążyć dodatkowym parametrem.

— 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.

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>;
```

— 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

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. Klauzule **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) { /* ... */ }
 }
```

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

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).

10.1.2. Przekazywanie informacji wraz z wyjątkiem

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

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";
   // ...
  }
 }
```

10.1.3. Hierarchie wyjątków

— Klasy wyjątków można łączyć w hierarchie.
— Pozwala to specjalizować obsługę 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. */ }
```

10.1.4. Dodatkowe własności wyjątków

— Wznawianie wyjątku **throw**
— Obsługa dowolnego 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
}
```

10.1.5. 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):

```
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
};

void używanie_pliku(const char* np){
 Wsk_do_pliku p(np, "w");
```

```
  /* coś z plikiem p */
 }
```

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

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

Tę technikę nazywa się zwykle "zdobywanie zasobów jest inicjacją".

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) { /* 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){
  p = new T[t];
 }
```

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

10.1.6. 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.

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() */ }
```

Jest to 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();
```

Literatura