Informatyka

Rafał Kawa

Programowanie 1

Slajdy do wykładu

2023/2024

Prezentowane slajdy nie są kolejnym kursem języka C++, jakich w Internecie tysiące. Są wprowadzeniem w ogólną dziedzinę programowania i tylko przy okazji przedstawiają w miarę obszerny kurs języka C++.

Część I

Podstawy podstaw

1 Wprowadzenie

1.1 Elementarne ogólne pojęcia

Programowanie - ciąg działań prowadzących do powstania programu komputerowego.

Program komputerowy - ciąg poleceń realizowanych przez komputer.

Komputer - tymczasowo przyjmujemy pojęcie za znane, więcej na przedmiocie *Wstęp do informatyki*.

1.2 Elementarne pojęcia programowania

- Najbardziej ogólny proces programowania, przebiega według kroków:
  1. Stworzenie kodu źródłowego.
  2. Przetworzenie kod źródłowego do kodu wykonywalnego.
- **Kod źródłowy**
  - Zapis programu komputerowego z użyciem sztucznego języka zwanego ***językiem programowania***.
  - Potocznie i kontekstowo często utożsamiany z powstającym z niego programem.
- **Kod wykonywalny**
  - Tymczasowo przyjmujemy równoważność z pojęciem programu komputerowego.
  - Kod wykonywalny nie jest tożsamy ze znacznie szerszym pojęciem kodu wynikowego.
- **Język programowania** - Zbiór reguł wymaganych przy zapisie programu komputerowego w formie kodu źródłowego i służących automatycznemu przetworzeniu kodu źródłowego do kodu wykonywalnego.
- **Kompilacja** (potocznie, w uproszczeniu i nie do końca poprawnie) - Ciąg czynności prowadzących od kodu źródłowego do kodu wykonywalnego
- **Kompilator** - program do kompilacji (również spore uproszczenie).

1.3 Wybór języka programowania

- Istnieje wiele języków programowania, w części poznawanych w trakcie studiów.
- **Do nauki programowania na przedmiocie Programowanie 1 wybrano język programowania C++ (oraz w pewnym zakresie język C).**
- Język C/C++:
  - Potocznie i kontekstowo często utożsamiany z powstającym z niego prograJest od wielu lat najpopularniejszym językiem programowania.
  - Stanowi inspirację dla większości innych popularnych języków programowania.
  - Nie ma ograniczeń wielu innych popularnych języków programowania.
  - Oferuje większe od innych języków programowanie możliwości stosowania ogólnego programowania.
  - Pozwala korzystać z większej wiedzy ogólnej informatyki.
  - Nie jest najłatwiejszym językiem programowania dla nauki programowania.
-

Ważniejsze od znajomości konkretnego języka programowania są ogólne prawidłowości programowania.

1.4 Środowisko programowania

- Spośród wielu sposobów programowania zastosujemy programowanie z użyciem *środowiska programowania*.
- Środowisko programowania
  - Program dedykowany różnorodnym działaniom w zakresie programowania.
  - Obejmuje najczęściej edytor kodu źródłowego, kompilator, możliwość nadzorowania wykonania, diagnostykę błędów i wiele innych działań.
- Na wykładzie używane będzie środowisko programowania *Dev-C++*.
- Ograniczoność środowiska *Dev-C++* (swoisty prymitywizm) jest jedną z jego zalet, pozwalającą skupić się na programowaniu bez rozwodzenia się nad aspektami samego środowiska.
- Istnieje wiele popularnych środowisk programowania dla języka C/C++ (*Visual Studio*, *Code::Blocs* i inne) bardziej rozbudowanych niż *Dev-C++*. Jednakże ich używanie wymaga dodatkowej wiedzy, niezwiązanej bezpośrednio z samym programowaniem.
- Szersza znajomość środowiska programowania jest przewidziana na zajęciach laboratoryjne.

1.5 Programowanie - rzemiosło, inżynieria czy sztuka?

- Programowanie jest przede wszystkim rzemiosłem i dopiero po opanowaniu rzemiosła można rozważać kwestię traktowania programowania w charakterze czegoś więcej.
- Programowanie staje się działalnością inżynierską (w rozumieniu poszukiwania rozwiązań problemów z koniecznością uwzględnienia obiektywnych uwarunkowań) od pewnego poziomu wiedzy ogólnie programistycznej, a jeszcze szerzej - informatycznej.
- Programowanie rzadko jest sztuką, a ewentualne elementy sztuki dotyczą algorytmów.
- Przedmiot *Programowanie 1* obejmuje zasadniczo naukę podstaw rzemiosła.
- Przedmiot *Programowanie 2* obejmuje zaawansowane aspekty rzemiosła.
- Dopiero połączenie wiedzy programistycznej i informatycznej pozwala traktować programowanie jak działalność inżynierską.
- Dla traktowania programowania w charakterze sztuki konieczne jest bycie informatykiem oraz pewna doza talentu algorytmicznego właściwa niewielkiemu procentowi informatyków.

1.6 Podsumowanie

- Programowanie w wersji przedmiotu Programowanie 1 jest nauką:
  - Podstaw ogólnego programowania …
  - … z wykorzystaniem języka programowania C++.
- Programowania nie można nauczyć się teoretycznie (czytając książki lub materiały w Internecie), dlatego …
-

Dla nauki programowania trzeba praktycznie programować.

- Nauka programowania nie jest kwestią ani inteligencji ani pilności, ale funkcją ilości czasu poświęconego praktycznemu programowaniu.

2 Elementarne podstawy

2.1 Oczywistości?

- Kody źródłowe są pisane w zwykłym pliku tekstowy.
- W języku C/C++ małe i wielkie litery są rozróżniane.
- Jeżeli z ogólnych powodów jest możliwy jeden odstęp lub jedna nowa linia, to jest możliwa dowolna ilość odstępów lub nowych linii.
- W zapisie liczb rzeczywistych część całkowitą od części ułamkowej oddziela znak kropki.
- Arytmetyka
  - Stosują się ogólne oznaczenia działań arytmetycznego dodawania, odejmowania i przeciwności.
  - Znakiem działania mnożenia jest znak gwiazdki *.
  - Znakiem działania dzielenia jest wyłącznie znak ukośnika /.
- Relacje
  - Oznaczenia silnych nierówności są zgodne z powszechnie przyjętymi.
  - Oznaczeniami słabej mniejszości i słabej większości są odpowiednio dwuznakowe sekwencje <= oraz >=.
  - Oznaczeniem różności jest sekwencja znaków wykrzyknika i równości (czyli !=).
  - Oznaczeniem równości jest sekwencja dwóch znaków równości (czyli ==).
- Programowanie słabo toleruje polskie znaki diakrytyczne (ą, ę i tak dalej), bezpieczniej ich nie stosować, a konieczne użycie zacieśnić do napisów (a i to nie zawsze).

2.2 Najprostszy program w języku C/C++

1. Uruchomienie środowiska programowania (w naszym przypadku *Dev-C++*).
2. Korzystając z edytora środowiska programowania napisanie najprostszego kodu źródłowego poprawnego programu w języku C/C++.

```
1 main () {
2     }
```

3. Na dysku komputera powstaje plik o wybranej przez użytkownika nazwie na przykład `first.cpp`.
4. Uruchomienie kompilatora środowiska (w *Dev-C++* między innymi poprzez klawisz F9 lub F11) z następującym komunikatem o poprawności kompilacji.
5. Na dysku powstaje plik `firts.exe` zawierający kod wykonywalny dla kodu źródłowego z pliku `first.cpp`.
6. Program powstaje, możliwe uruchomienie (zadziałanie), brak jakiejkolwiek interakcji z użytkownikiem.

2.3 Wnioski z pierwszego programu w C/C++

- Wszystkie programy tworzone na przedmiocie *Programowanie 1* będą kodami wykonywalnymi dla interfejsu znakowego.
- Przydatne będą elementarne polecenia systemu operacyjnego DOS, czyli fundamentu systemu operacyjnego *Microsoft Windows* objawianego najczęściej czarnym oknem „systemowym” z trybem tekstowym.

2.4 Ogólna struktura programu w języku C/C++

… obejmuje:

1. Konieczne słowo `main` (czyli główny).
2. Nawiasy okrągłe z ewentualną zawartością - tymczasowo pozostawiamy pustą, więcej w dalszej części wykładu.
3. Opis działania programu objęty klamrowymi nawiasami.

```
main () {
    Opis działania programu
    }
```

Zarazem:

- *Opis działań programu* to zasadnicza treść wykładu.
- Przed słowem `main` (a nawet po klamrowym zamykającym nawiasie) mogą znaleźć się dodatkowa treści, ale powyższy fragment jest niezbywalny.
- Opisany fragment w całym kodzie źródłowym może być tylko jeden.

2.5 Najprostsze wyświetlanie

2.5.1 Wprowadzenie

- Poprzedni program (wbrew pozorom) wykonuje sporo ważnych czynności związanych z uruchomieniem i zakończeniem pracy każdego programu (nie bez powodu zajmuje ponad 15kB rozkazów).
- Użyteczne programy powinny objawiać rezultat działania, dlatego wyświetlanie informacji na ekranie jest jednym z najbardziej elementarnych poleceń.
- Polecenia wypisywania (ogólnie, instrukcje wyjścia) są jednymi z najbardziej różnorodnych w niemal wszystkich językach programowania.
- W każdym języku programowania polecenie wypisania na ekran obejmuje dwie składowe:
  - Nazwę polecenia wypisującego.
  - Treść do wypisania.

2.5.2 Próby, próby, próby …

2.5.2.1. Wstępna wiedza

- Dokumentacja, przykłady kodów źródłowych i inna wiedza wiążą z wypisywaniem w języku C++ słowo `cout` (od *console output*), zatem być może tak nazywa się polecenie wypisujące?
- Przyjmijmy do wypisania treść:

```
tekst dla wyswietlenia
```

- Pytanie:
  Jak oddzielić polecenie wyświetlania od słów komunikatu? Czyli skąd kompilator ma wiedzieć, że słowa komunikatu nie są kolejnymi poleceniami?
- Odpowiedź:
  Różne konwencje oddzielania komunikatów od innych treści, najczęściej poprzez objęcie znakami tak zwanego amerykańskiego cudzysłowu, popularnie (niezbyt fachowo) zwanego podwójnym apostrofem.

2.5.2.2. Pierwsza wersja

- Pierwsza propozycja kodu wyświetlającego komunikat:

```
1 main () {
2     cout "tekst dla wyswietlenia"
3     }
```

- Błędy:
  - *`cout' undeclared (first use this function)*
    W tłumaczeniu *„brak deklaracji słowa cout”*, a naprawdę *„nie wiem czym jest cout”*
  - *expected `;' before string constant*
    W tłumaczeniu *„oczekiwany `;' przed stałą znakową”*.
- Tymczasowo zaniedbujemy komunikat o nieznajomości cout, dokładamy średnik zgodnie z komunikatem.

2.5.2.3. Druga wersja

- Druga propozycja kodu wyświetlającego komunikat:

```
1 main () {
2     cout ; "tekst dla wyswietlenia"
3     }
```

- Zmiana komunikatu błędu:

expected `;' before string constant

na komunikat:

expected `;' before '}' token

w tłumaczeniu *„oczekiwany `;' przed znakiem '}'”*.

- Zmiana wersji

```
1 main () {
2     cout ; "tekst dla wyswietlenia";
3     }
```

znika błąd braku średnika, pozostaje błąd o nieznajomości `cout`.

- Poprawne umiejscowienie średników: tylko jeden średnik na końcu.

2.5.2.4. Trzecia wersja

- Aktualna wersja:

```
1 main () {
2     cout "tekst dla wyswietlenia";
3     }
```

pozostaje komunikat o nieznajomości `cout`.

- Przyjmujemy do wiadomości, iż pewna ilość bytów programistycznych jest zdefiniowana (a nawet niezmienna), zatem konieczne jest dowiedzenie się jak sprawić by określenia owych bytów były dostępne w naszym kodzie źródłowym.
- Wymagana wiedza obejmuje:
  - Polecenia „zauważenia” przez kod źródłowy zdefiniowanych bytów.
  - Nazwę bytu obejmującego definicję `cout`.

2.5.2.5. Czwarta wersja

- Pozostaje tymczasowo przyjąć do wiadomości, iż:
  - Jedną z wielu możliwości udostępnienia bytów zdefiniowanych przez środowisko jest sekwencja `#include` (więcej w dalszej części wykładu).
  - Informacje z predefiniowanymi bytami są zawarte w różnych plikach dostarczanych wraz ze środowiskiem programowania i definicję `cout` zawiera plik o nazwie `iostream`.
  - Sekwencja „zauważenia” przez kod źródłowy zdefiniowanych bytów ma strukturę:

    `#include <nazwa_pliku>`
  - Powyższa sekwencja musi być umieszczona przed użyciem zawierających bytów i dobrym miejscem jest między innymi poprzedzanie słowa `main`.
- Podsumowanie powyższej wiedzy w kodzie źródłowym:

```
1 #include <iostream>
2
3 main () {
4     cout "tekst dla wyswietlenia";
5     }
```

- Błąd pozostaje!!!
- Istnieją języki programowania z poprawnością odpowiednika powyższego kodu.

2.5.2.6. Piąta wersja

- Okazuje się, że `cout` (wbrew wielu dostępnym informacjom) to zbyt mało dla wyświetlenia, ponieważ odpowiednia (pełna) nazwa bytu służącego wyświetlaniu to `std::cout`.
- Kolejna wersja:

```
1 #include <iostream>
2
3 main () {
4     std::cout "tekst dla wyswietlenia";
5     }
```

- Znika błąd niewiedzy dotyczącej `cout`!!!
- Powraca błąd:

expected `;' before string constant

2.5.2.7. Szósta wersja

- Okazuje się, że `cout` (a nawet `std::cout`) nie jest instrukcją wypisująca na ekran.
- Napis `std::cout` jest określeniem miejsca (urządzenia) pozwalającego odebrać informacje.
- Wirtualnych miejsc w komputerze pozwalających odebrać informacje jest więcej niż urządzeń fizycznych zdolnych zrealizować obiór.
- Konsola w programowania to byt wirtualny, wejściowo/wyjściowy, a nie tylko klawiatura.
- Domyślnie konsola (wirtualna) jest fizycznie sprzężona z klawiaturą i monitorem, ale wcale tak być nie musi.
- Jednym z wielu możliwości wysłania treści na ekran jest użycie sekwencji znaków << zgodnie ze schematem:

  urządzenie << *treść*;

  i właśnie sekwencja << jest instrukcją wypisującą.
- Zatem kolejna wersja:

```
1 #include <iostream>
2
3 main () {
4     std::cout << "tekst dla wyswietlania";
5     }
```

działa poprawnie!!!

2.5.2.8. Wersja ostateczna

- Ostatni kod źródłowy jest poprawnie kompilowany w *Dev*-C++, ale nie musi być poprawnie kompilowany w innych środowiskach.
- Z punktu widzenia ścisłego standardu języka C/C++ jest nawet błędny, ale akceptowalny w dążenia do dydaktycznego uproszczania.
- Przyjmiemy wersję bardziej poprawną, …

```
1 #include <iostream>
2
3 int main () {
4     std::cout << "tekst dla wyswietlenia";
5
6     return 0;
7     }
```

… odkładając wyjaśnienie dodania słowa **`int`** przed słowem `main` oraz sekwencję **`return 0`** przed kończącym nawiasem klamrowym na niezbyt odległą przyszłość.

2.5.3 Wstępne wnioski

2.5.3.1. Komunikaty o błędach

- Komunikaty o błędach kompilacji mogą niewiele mówić lub nawet być kłamliwe.
- Różne kompilatory ten sam błąd opisują różnymi komunikatami.
- Nie jest możliwe opisanie nawet kilkoma zdaniami wszystkich możliwych przyczyn nawet pojedynczego błędu.
- Komunikat o błędzie jest raczej wskazówką niż precyzyjną diagnozą.
- Błędy kompilacji są najprostszą w obsłudze kategorią błędów programistycznych.
- Więcej o błędach kompilacji w dalszej części wykładu.

2.5.3.2. Dyrektywa `#include`

- Sekwencja `#include` jest jedną z możliwych tak zwanych dyrektyw preprocesora (jak jednoznacznie wskazuje znak # będący pierwszym znakiem w linii).
- Wbrew wielu błędnym opiniom, poprzez sekwencję `#include` nie włączamy żadnych bibliotek, bowiem biblioteki (czymkolwiek są) w procesie tworzenia finalnego programu biorą udział po przetworzeniu kodu źródłowego.
- Jedyną funkcją sekwencji `#include` jest wstawienie w kod źródłowy zwartości pliku o nazwie podanej w ostrych nawiasach w zastępstwie jawnego umieszczenie pliku wprost w kodzie.
- Praktycznie, kompilator napotykając sekwencję `#include` przełącza czytanie aktualnego kodu źródłowego na kod zawarty w pliku o nazwie podanej w ostrokątnych nawiasach, by po skończeniu wrócić do czytania pliku zawierającego sekwencję `#include`.
- Dyrektywy preprocesora nie będąc zagadnieniem szczególnie skomplikowanym pozostają zagadnieniem stosunkowo obszernym, dlatego kompleksowo będą omawiane na przedmiocie *Programowanie 2*, co nie przeszkodzi korzystać z innych wybranych elementów zagadnienia na przedmiocie *Programowanie 1*.

2.5.3.3. Inne wnioski

- Plik `iostream` obejmuje opisy różnorodnych treści związane z jednym z wielu mechanizmów wejścia/wyjścia (stąd litery i-*input* oraz o-*output*) zwanym strumieniowym (od słowa *stream*), w sumie składając się na nazwę `iostream`.
- Zawartość pliku `iostream` pozwala używać ogólnego bytu o nazwie `std` (standardowe wejście/wyjście), którego częścią jest obsługa wyjścia na konsolę (*console output*), czyli `cout`.
- Szerszy byt `std` jest zacieśniony do węższego bytu `cout` poprzez rozdzielenie parą znaków dwukropka (`::`) stanowiąc użycie tak zwanego operatora zakresu, o czym więcej w dalszej części wykładu.
- Średnik na końcu sekwencji wypisywania nie jest integralnym elementem konkretnej instrukcji wypisującej, ale objawem ogólnej reguły języka C/C++ mówiącej:

Po każdym poleceniu musi następować znak średnika

2.5.4 Więcej o wypisywaniu

2.5.4.1. Wypisywanie wielu linii

- *A może wyświetlić kilka linii tekstu?*

```
#include <iostream>

int main () {
    std::cout << "tekst dla wyswietlenia";
    std::cout << "kolejny tekst w drugiej linii";
    std::cout << "i jeszcze jeden w trzeciej linii";

    return 0;
    }
```

- *Uruchamiamy i zaskoczenie? Jedna długa linia!*
- *Dlaczego miałoby być inaczej? Brak rozkazu przejścia do nowej linii! Jak zmusić komputer do wykonania czegoś, co jest funkcją edytora?*
- Funkcji uzyskania nowej linii nie może pełnić użycie klawisza `Enter`, ponieważ służy przejściu do nowej linii w edytorze kodu źródłowego. Zatem muszą istnieć specjalne sposoby.

- Istnieje wiele sposobów nakazania przejścia do nowej linii przy wyświetlaniu informacji na ekranie.
- Jednym z elegantszych (w zapisie) sposobów uzyskania na ekranie przejścia do nowej linii jest wypisanie specjalnego słowa `endl`.
- Kod źródłowy:

```cpp
#include <iostream>

int main () {
    std::cout << "tekst dla wyswietlenia";
    std::cout << std::endl;
    std::cout << "kolejny tekst w drugiej linii";
    std::cout << std::endl;
    std::cout << "i jeszcze jeden w trzeciej linii";
    std::cout << std::endl;

    return 0;
    }
```

- Słowo `endl` jest składową ogólnego bytu o nazwie `std`, możliwą do uzyskania podobnie jak `cout` poprzez operator zacieśnienia `::`.

2.5.4.2. Łączenie poleceń wypisywania

- *Skoro miejsc do wyświetlania jest jedno (czyli `std::cout`), to może ilość użyć `std::cout` można ograniczyć?*
- Łączenie poleceń wyświetlania w rozumieniu sekwencji

  `<< treść`

  jest możliwe przy początkowym określeniu miejsca wyświetlania.

```
1 #include <iostream>
2
3 int main () {
4     std::cout << "tekst dla wyswietlenia" << std::endl;
5     std::cout << "kolejny tekst w drugiej linii" << std::endl;
6     std::cout << "i nastepny w trzeciej linii" << std::endl;
7
8     return 0;
9     }
```

- Ogólna postać wyświetlania w dalszej części wykładu.

2.5.4.3. Pominięcie słowa `std` - przestrzeń nazw

- *Wiele stosowań słowa `std`, mimo że nic nie wnosi - może jednak istnieje możliwość opuszczenia?*
- Spróbujmy następująco:

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      cout << "tekst dla wyswietlenia" << endl;
6      cout << "kolejny tekst w drugiej linii" << endl;
7      cout << "i nastepny w trzeciej linii" << endl;
8
9      return 0;
10     }
```

- Więcej w dodatkowych wnioskach.

2.5.5 Dodatkowe wnioski

2.5.5.1. Ogólna postać wypisywania

Ogólna postać polecenia wypisującego ma strukturę:

wyjście wypisywania;

gdzie:

- *wyjście* - określenie urządzenia (wirtualnego lub fizycznego) przekazującego dane z komputera na zewnątrz - w naszym przypadku `std::cout`.
- *wypisywania* - dowolna ilość sekwencji wypisujących postaci:

  << *treść*

  przy czym *treść* oznacza między innymi napisy, ale i wiele innych możliwości. Więcej w dalszej części wykładu.

2.5.5.2. Przestrzeń nazw

- Sekwencja:

```
using namespace std;
```

w swobodnym tłumaczeniu oznacza „*używaj przestrzeni nazw* `std`”, czyli poszukiwaną możliwość pomijania słowa `std`.

- W rzeczywistości oznacza dużo więcej, między innymi poinformowanie kompilatora o stałym wykorzystywaniu pewnego programistycznego mechanizmu nazywanego **przestrzenią nazw** (skąd **`namepace`**) pozwalającego ujednoznacznić dostęp do bytu `std`.
- Przestrzenie nazw:
  - Nie są konieczne dla programowania w C++.
  - Często zmniejszają rozumienie kodu źródłowego.
  - Zostaną omówione w dalszej części wykładu.

3 Podstawy zmiennych

3.1 Wprowadzenie

- Trudno działać jedynie na obiektach zapisanych na stałe w kodzie źródłowym i dlatego naturalną jest chęć działania na komputerze podobnie do działań na wzorach znanych ze szkolnej matematyki.
- Niezbędnym do działania na komputerze jak na wzorach jest:
  1. Przekazanie komputerowi (kompilatorowi poprzez kod źródłowy) nazwy pojęcia opisanego wzorem.
  2. Znajomość reguł wprowadzania własnych nazw w danym języku programowania.
- Przedmiot działań (odpowiednik liter ze szkolnych wzorów) nazywany jest potocznie ***zmienną***.

3.2 Powszechne nieprawdy

- Nie jest prawdą, że zmienna jest ZAWSZE miejscem w pamięci komputera, ponieważ …
- … zmienne istnieją bez pamięci komputera, istnieją bez komputera, a nawet (w pewnym sensie) istnieją nawet bez informatyki.
- Zmienne bywają fragmentami pamięci komputera, ale tylko w jednym ze sposobów realizacji ich użytkowania.
- Nie jest prawdą, że zmienna MUSI posiadać typ. Nawet w praktycznym programowaniu.

3.3 Pojęcie zmiennej

- Brak powszechnie uznawanej ścisłej definicji ***zmiennej***, dlatego …
- … ***zmienną*** „definiuje się” poprzez opis posiadanych cech.
-

W ogólnej informatyce, **zmienna** jest bytem charakteryzującym się:
  - Możliwością przechowywania informacji (pamiętania).
  - Możliwością objawienia przechowywanej informacji (odczytu).
  - Możliwością zmiany przechowywanej informacji na inną (edycji).
  - Unikalną nazwą umożliwiającą realizację powyższych możliwości (identyfikacją).

- Ponadto:
  - Każda z wymienionych cech zmiennej jest niezbywalna.
  - W ogólnej informatyce żadna dodatkowa cecha zmiennej nie jest konieczna.
  - W szczególności nie jest konieczne określanie typu zmiennej (nawet, jeżeli większość języków programowania wymaga typu zmiennej przy kreacji).
-

Zmienne mogą stanowić składową wyrażeń i użyte w wyrażeniu odpowiadają użyciu aktualnie pamiętanej wartości.

3.4 Użytkowanie zmiennych

3.4.1 Deklaracja istnienia zmiennej

- Większość języków programowania dla korzystania ze zmiennych wymaga specjalnego zapisu kreacji zmiennej, nazywanego **deklaracją zmiennej**.
- W języku C/C++ deklaracja zmiennej składa się z trzech części:
  1. Nazwy typu zmiennej (więcej o typach zmiennej w dalszej części wykładu).
  2. Nazwy zmiennej (wymyślanej przez programistę zgodnie z opisanymi dalej warunkami).
  3. Znaku średnika.
- Możliwe jest deklarowanie niemal dowolnej ilości zmiennych, ale z koniecznością rozróżniania nazw.
- Przykład deklaracji zmiennej:

```
1 int main () {
2     int integerVariable;
3
4     return 0;
5     }
```

- W powyższym kodzie druga linia jest deklaracją zmiennej o nazwie `integerVariable`, o ***typie zmiennej*** określonym słowem **`int`**.
- Typ **`int`** posługuje się liczbami całkowitymi.

3.4.2 Wprowadzenie do typu zmiennej

- Nie jest prawdą, że typ zmiennej to sposób reprezentacji zmiennej w komputerze.
- Nawet powszechnie znany typ całkowity może mieć wiele możliwych różnych reprezentacji nie przestając być typem całkowitym.
- Nie jest prawdą, że typ zmiennej jest nieokreślonym, ale potocznie rozumianym rodzajem zmiennej.
- **Typ zmiennej** jest ściśle definiowalny:

> **Typ zmiennej** to zbiór wartości możliwych do pamiętania przez zmienną

- Tymczasowo, dla określenia typów zmiennych oprócz słowa **`int`** będzie używane również słowo **`float`** określające typ liczb rzeczywistych.

3.4.3 Nazewnictwo zmiennych

- Nazwy zmiennych wymyśla programista.
- Postać nazwy zmiennej musi być zgodna z regułami obowiązującymi dla danego języka programowania.
- Wymyślanie potocznie rozumianych „dobrych” nazw zmiennych nie jest ścisłe, jednakże …
- … kilkadziesiąt lat programowania pozwoliło wypracować zwyczaje powszechnie uznawane za dobre oraz zwyczaje powszechnie uznawane za złe.
- Tymczasowo przyjmujemy, że nazwy zmiennych są zawsze spójnym ciągiem liter lub cyfr, z konieczną pierwszą literą.

3.4.4 Wyświetlanie wartości zmiennej

- Wyświetlanie wartości zmiennej odbywa się identycznie z wyświetlaniem napisów, liczb lub wyrażeń - czyli poprzez podanie nazwy zmiennej.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      float realVariable;
6
7      cout << "realVariable wynosi " << realVariable << endl;
8
9      return 0;
10     }
```

- Powyższy program wyświetli zawartość zmiennej i będzie to wartość PRZYPADKOWA!!! Na innym komputerze, w innym systemie operacyjnym, o innym czasie może być inna.
- Zmienna zajmuje miejsce w pamięci, miejsce musi być czymś wypełnione i bez polecenia zawartość jest zastana, a więc przypadkowa.
- Więcej w dalszej części wykładu.

3.4.5 Nadawanie wartości zmiennej

W ogólnym programowaniu zmiennym można nadawać wartości na trzy najczęściej występujące sposoby:

1. Poprzez specjalne polecenie jawnie nadające wartość zmiennej, zwane ***podstawieniem do zmiennej***.
2. Poprzez różnorodne operacje przekazujące wartość do zmiennej z urządzeń wejściowych.
3. Poprzez nadanie wartości przy deklaracji zmiennej, zwane ***inicjalizacją zmiennej***.

3.4.5.1. Podstawienie do zmiennej

- Podstawienie do zmiennej jest jedną z najbardziej fundamentalnych czynności w programowaniu.
- Podstawienie do zmiennej jest operacja złożoną, więcej na przedmiocie *Wstęp do informatyki*.
- Języki programowania stosują różne konwencje podstawienia.
- W języku C/C++ najprostsza wersja podstawienia do zmiennej jest poleceniem złożonym z trzech składowych:
  1. Nazwy zmiennej.
  2. Znaku równości = (w środowisku programistów powszechnie, ale niezbyt poprawnie nazywanego znakiem podstawienia).
  3. Wyrażenia określającego podstawianą wartość.

- Przykład:

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      float floatVariable;
6
7      floatVariable = 1234.56;
8      cout << "floatVariable = " << floatVariable << endl;
9
10     return 0;
11     }
```

siódma linia powyższego kodu źródłowego jest realizacją opisanej reguły.

3.4.5.2. Wczytywanie do zmiennej

- Języki programowania znają wiele operacji wejścia/wyjścia, zatem również wczytanie do zmiennej może odbywać się na wiele sposobów, więcej w dalszej części wykładu.
- Najprostsza wersja polecenia wczytującego ma postać:

  `cin >> zmienna;`

  gdzie:
  - `cin` jest określeniem urządzenia wejściowego (od *console input*), najczęściej tożsamego z klawiaturą.
  - Sekwencja `>>` jest zapisem polecenia przekazującej zawartość z wejścia do zmiennej.
  - *`zmienna`* jest nazwą istniejącej zmiennej.

- Przykładowy kod:

```
#include <iostream>
using namespace std;

int main () {
    int i;

    cout << "podaj liczbe -> ";
    cin >> i;
    cout << "podana liczba wynosi " << i;

    return 0;
    }
```

- Działanie:
  - Komputer czeka na podanie danych z klawiatury.
  - Aż do naciśnięcia klawisza Enter można dowolnie edytować wpisywany ciąg znaków, a komputer nie interesuje się naszym wpisywaniem.
  - Dopiero naciśnięcie klawisza `Enter` powoduje, że komputer podejmuje dalsze działanie, zaczynające się interpretacją podanego ciągu znaków.
- Więcej o wczytywaniu w dalszej części wykładu.

3.4.5.3. Inicjalizacja zmiennej

- Zmienna w programowaniu zawsze zawiera jakąś wartość.
- Nawet najprostsza deklaracja zmiennej oznacza nadanie jakiejś wartości, jednakże może to być wartość niezależna od naszej woli, a co najwyżej zgodną z naszą wiedzą.
- Skoro nawet po deklaracji zmienna zawiera wartość, można nadać zmiennej wartość zamierzoną, a nie przypadkową. Właśnie taka jest istota inicjalizacji zmiennej.
- W języku C/C++ inicjalizacja zmiennej polega na rozszerzeniu deklaracji o fragment inicjujący, czyli połączeniu deklaracji zmiennej z instrukcją podstawienia.
- Pozostawienie zawartości zmiennych przypadkowi dyskwalifikuje nawet początkującego programistę.

- Deklaracja zmiennej połączona z inicjalizacją ma postać:
  1. Nazwy typu zmiennej.
  2. Nazwy zmiennej.
  3. Znaku podstawienia =.
  4. Wyrażenia określającego inicjowaną wartość zmiennej.
  5. Znaku średnika.
- Przykład:

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      float f = -98.76;
6
7      cout << "f = " << f << endl;
8
9      return 0;
10     }
```

3.4.6 Ewaluacja podstawienia

- W strukturze wyrażenia będącego prawą stroną instrukcji podstawienia mogą się pojawić zmienne identyczne ze zmienną po lewej stronie instrukcji podstawienia.
- Opisana sytuacja nie jest niczym nadzwyczajnym, a wprost przeciwnie, najzupełniej naturalnym, niezbędnym i nie stanowi żadnego problemu. Komputer nie zgłupieje, nie zachodzi również żaden konflikt między podstawianiem samego siebie do samego siebie.
- Brak problemów wynika z ogólnie informatycznego sposobu działania instrukcji podstawienia dla wszystkich komputerów.
- Ogólne działanie instrukcji podstawienia przebiega w dwu etapach:
  1. Wyliczeniu wartości wyrażenia po prawej stronie znaku podstawienia z zapamiętywaniem wartości cząstkowych i końcowej w miejscu pamięci komputera niezwiązanym z użytymi zmiennymi, a używane zmienne są wyłącznie odczytywane.
  2. Końcowa wartość wyrażenia jest przechowywana gdzieś w pamięci tymczasowej komputera i dopiero po kompletnym wyliczeniu jest wstawiana do zmiennej zapisanej z lewej strony instrukcji podstawienia.

- Inaczej mówiąc, dla wyrażenia pobierane są początkowe wartości używanych zmiennych, zaś zmiana wartości zmiennej jest ostatnią operacją instrukcji podstawienia.
- Więcej na przedmiocie *Wstęp do informatyki*.
- Przykładowy kod:

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6
7      cin >> i;
8      cout << "i = " << i << endl;
9
10     i = 2 * i;
11     cout << "podwojenie = " << i << endl;
12
14     i = i * i + i * 20 - 100;
15     cout << "wyrazenie = " << i << endl;
16
17     i = i - 123;
18     cout << "zmniejszenie = " << i << endl;
19
20     return 0;
21     }
```

4 Więcej o wczytywaniu

4.1 Ogólna postać wczytywania

Ogólna postać polecenia wczytującego jest zgodna ze strukturą:

wejście wczytywania;

gdzie:

- *wejście* - jest określeniem urządzenia przyjmującego dane z zewnątrz (wirtualnego lub fizycznego, analogicznie jak przy wypisywaniu), w naszym przypadku `std::cin`.
- *wczytywania* - są dowolną ilością sekwencji wczytujących postaci:

  >> *zmienna*

  gdzie *zmienna* oznacza nazwę istniejącej zmiennej.

4.2 Uzupełnienia wczytywania

- Z ogólnej postaci wczytywania do zmiennej wynika możliwość wielokrotnego wczytywania w jednym poleceniu.
- Mimo jednego polecenia, po wpisaniu każdej danej i naciśnięciu klawisza `Enter` komputer czeka na kolejne dane i sytuacja występuje tyle razy ile zmiennych występuje w poleceniu.
- Możliwe jest wykorzystanie klawisza `Enter` tylko raz. Wystarczy wszystkie dane wpisać w jedną linię, ale oddzielić je nie mniej niż jedną spacją. Na podstawie spacji komputer rozdzieli jedną linię testu na części podstawi do kolejnych zmiennych.
- Możliwe są różne wersje pośrednie w zależności od ilości danych w linii i ilości naciśnięć klawisza `Enter`.

- Kod źródłowy dla testowania wczytywania:

```
#include <iostream>
using namespace std;

int main () {
    float f;
    float g;
    float h;

    cin >> f >> g >> h;
    cout << "f = " << f << ", g = " << g << ", h = " << h << endl;

    return 0;
    }
```

- **Uwaga: wczytywanie poprzez sekwencję >> ignoruje wielokrotnie powtarzające się spacje. Możemy nie zwracać uwagi na ilość spacji, ale pojawia się problemy w sytuacji spacji stanowiącej element danych. Stąd konieczność istnienia innych sposobów wczytywania - więcej w dalszej części wykładu.**

5 Komentarze w kodzie źródłowym

- Dotychczasowe komentarze dotyczące pisanych programów były umieszczone poza kodem źródłowym w charakterze odpowiedników komentarzy mówionych, zaś pisanie komentarzy, (czyli fragmentów niepodlegających kompilacji, zatem opisujących kod źródłowy) jest przydatne i możliwe.
- Język C++/C dopuszcza dwa sposoby komentowania. Starszy zwany obecnie wielowierszowym i nowszy zwany obecnie jednowierszowym.
- **Komentarze wielowierszowe**
  ... polegają na ignorowaniu każdego fragmentu kodu źródłowego zapoczątkowanego sekwencją znaków /* aż do sekwencji znaków */ włącznie.
- **Komentarze jednowierszowe**
  … polegają na ignorowaniu każdego fragmentu linii kodu źródłowego począwszy od sekwencji // aż do końca linii.

Kod źródłowy dla prezentacji użycia komentarzy.

```
1  /*-----------------------
2  Program prostego sumatora
3  -----------------------*/
4  #include <iostream>
5  using namespace std;
6
7  int main () {
8      int a; // zmienna dla wczytania pierwszej liczby
9      int b; // zmienna dla wczytania drugiej liczby
10
11     cout << "Program sumujący dwie liczby." << endl;
12
14     cout << "podaj pierwsza liczbe: ";
15     cin >> a; // wczytanie pierwszej liczby
16
17     cout << "podaj druga liczbe: ";
18     cin >> b; // wczytanie drugiej liczby
19
20     // wypisanie wyniku
21     cout << a << " + " << b << " = " << a + b << endl;
22
23     return 0;
24     }
```

6 Podstawy instrukcji

6.1 Wprowadzenie

Każdy niebanalny program komputerowy jest zapisem ***algorytmu***.

6.1.1 Algorytm

- **Algorytm** jest podstawowym pojęciem informatyki, ważniejszym nawet niż pojęcie komputera.
- Algorytm nie jest tylko przepisem.
- Algorytm nawet będąc przepisem musi spełniać wiele warunków.
- Więcej na przedmiocie *Wstęp do informatyki* oraz w całym toku studiów.
-

Tymczasowo przyjmujemy pojęcie algorytmu za intuicyjnie znane.

- Algorytmy posługują się ***instrukcjami***.

6.1.2 Instrukcja

- Brak formalnej i powszechnie uznawanej definicji pojęcia instrukcji.
- Najważniejsze wspólne cechy wszystkich definicji instrukcji obejmują:
  - Instrukcja jest zapisem działania programu komputerowego.
  - Działanie opisane pojedynczą instrukcją jest wykonywane niepodzielnie.
- Tymczasowo przyjmujemy uproszczone określenie instrukcji:

Instrukcja jest zapisem niepodzielnie wykonywanego fragmentu działania programu.

- Istnieje wiele klasyfikacji instrukcji.
- Więcej na przedmiocie *Wstęp do informatyki* i w dalszej części materiału.

6.2 Instrukcja podstawienia

(przypomnienie)

- Wiele języków programowania ma wiele różnych postaci instrukcji podstawienia.
- W języku C/C++ instrukcja podstawienia ma jedną, ogólną, ściśle określoną strukturę obejmującą kolejno:
  1. Nazwę zmiennej.
  2. Znak podstawienia =.
  3. Wyrażenie odpowiadające (po wyliczeniu) wartości typu zmiennej.

6.3 Instrukcja blokowa

- **Blok** - ciąg instrukcji wykonywanych niepodzielnie.
- W języku C/C++ każdy spójny ciąg instrukcji można objąć nawiasami klamrowymi (czyli z użyciem znaków { oraz }), określając wymóg niepodzielnego wykonania całości instrukcji.
- Działanie głównego programu w języku C/C++ jest jedną instrukcją blokową.
- Różne języki programowania mają różne sposoby oznaczania bloków.
- Istnieją języki programowania bez dedykowanych oznaczeń bloków, co nie oznacza braku instrukcji blokowej.
- Przykłady użycia instrukcji blokowej w dalszej części materiału.

6.4 Instrukcja warunkowa krótka

6.4.1 Ogólności

- Instrukcja działania wykonywanego przy prawdziwości warunku logicznego.
- Postać w języku C/C++:

```
if ( warunek )
    działanie
```

  gdzie:
  - Słowo **`if`** Jest obowiązkowe.
  - *`warunek`* (logiczny) musi wystąpić w okrągłych nawiasach.
  - *`działanie`* oznacza dowolną instrukcję.
- W ogólnym programowaniu *`warunek`* jest wyłącznie warunkiem logicznym, jednak w języku C/C++ warunek może być dowolnego typu z racji szczególnego rozumienia typu logicznego. Więcej w dalszej części wykładu.

Kod źródłowy dla demonstracji instrukcji warunkowej krótkiej.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6      int j;
7
8      cin >> i >> j;
9
10     if ( i > j )
11         cout << "Zgadza sie, pierwsza wieksza";
12
13     return 0;
14     }
```

6.4.2 Najczęstsze błędy

- W języku C/C++ warunkiem może być również instrukcja, na przykład:

```
1  int main () {
2      int i;
3
4      cin >> i;
5      if ( i = 5 ) // To nie jest równość!!!
6      //if ( i = 0 ) // To nie jest równość!!!
7          cout << "Zgadza sie, wczytane piec";
8
9      return 0;
10     }
```

- Więcej w dalszej części materiału.

- Rozważmy poniższy kod:

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6
7      cin >> i;
8
9      if ( i == 5 )
10         cout << "Wczytane i ";
11         cout << "wynoszace 5" << endl;
12
13     return 0;
14     }
```

Napis `"wynoszace 5"` wyświetli się ZAWSZE!

- Poprawna wersja wcześniejszego kodu:

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6
7      cin >> i;
8
9      if ( i == 5 ) {
10         cout << "Wczytane i ";
11         cout << "wynoszace 5" << endl;
12         }
13
14     return 0;
15     }
```

Konieczna instrukcja blokowa!!!

6.5 Instrukcja warunkowa długa

- Instrukcja z podanym warunkiem oraz dwoma działaniami. Jednym działaniem wykonywanym przy prawdziwości warunku oraz drugim działaniem wykonywanym przy fałszywości warunku.
- Postać w języku C/C++:

```
if ( warunek )
    działanie gdy prawda
else
    działanie gdy fałsz
```

przy czym:

- Słowa **`if`** oraz **`else`** są obowiązkowe.
- Warunek (logiczny) musi wystąpić w okrągłych nawiasach.
- Oba działania są dowolnymi instrukcjami (jak dla instrukcji warunkowej krótkiej).

Kod źródłowy dla demonstracji instrukcji warunkowej długiej.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6
7      cin >> i;
8
9      if ( i >= 5 )
10         cout << "ok, co najmniej 5";
11     else
12         cout << "niestety, mniej niz 5";
13
14     return 0;
15     }
```

6.6 Instrukcje pętli

6.6.1 Wprowadzenie

- *„Pętla jest duszą programowania”*
- W ogólnej informatyce istnieje tylko jedna pętla, wszystkie inne pętle są jej szczególnymi wersjami dla ułatwienia zapisu.
- Algorytmice wystarczają dwie pętle znane pod ogólnie informatycznymi nazwami **pętli *while-do*** oraz **pętli *repeat-until*** i żadne inne pętle nie są konieczne. Więcej na przedmiocie *Wstęp do informatyki*.
- W szczególności, ogólnej informatyce nie jest konieczna **pętla *for*** niezależnie od wersji konkretnego języka programowania.
- Ogólne programowanie rozróżnia wiele rodzajów pętli, jak na przykład pętla ogólna, pętla zliczająca, pętla iteracyjna, pętla powtórzeniowa, pętla wyliczeniowa, pętla nieskończona i wiele innych. Będą poznawane na innych przedmiotach w toku studiów.
- Język C/C++ jest stosunkowo ubogo wyposażony w konstrukcje pętli, ale w zupełności wystarczająco.
- Równoważne wyrażenie jednych pętli przez inne pętle jest tematem zajęć laboratoryjnych.

6.6.2 Pętla `while`

- Z racji bliskości językowi naturalnemu jest pętlą najogólniejszą i najczęściej wykorzystywaną w ogólnym programowaniu.
- Odpowiada ogólnie informatycznej pętli *while-do*.
- Opis słowny: *jak długo spełniony jest warunek wykonuj działanie*.
- Struktura zapisu:

```
while ( warunek )
    działanie
```

gdzie:

- Słowo **`while`** (dopóki) jest obowiązkowe.
- *`warunek`* (logiczny) musi być umieszczony w okrągłych nawiasach.
- *`działanie`* jest dowolną instrukcją.

Kod źródłowy dla demonstracji instrukcji pętli *while*.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6
7      cout << "ile razy? ";
8      cin >> i;
9
10     while ( i > 0 ) {
11         cout << i << endl;
12         i = i - 1;
13         }
14
15     return 0;
16     }
```

6.6.3 Pętla do-while

- Odpowiednik ogólnie informatycznej pętli *repeat-until*.
- Opis słowny: *wykonuj działanie jak długo warunek jest spełniony*.
- Struktura zapisu:

```
do
    działanie
while
( warunek );
```

przy czym:

- Słowo **do** (wykonuj) jest obowiązkowe.
- *działanie* jest dowolną instrukcją.
- Słowo **while** (dopóki) jest obowiązkowe.
- *warunek* (logiczny) musi wystąpić w okrągłych nawiasach.

Kod źródłowy dla demonstracji instrukcji pętli *do-while*.

```
#include <iostream>
using namespace std;

int main () {
    int summary = 0;
    int i;

    do {
        cin >> i;

        summary = summary + i;
        }
    while
        ( i != 0 );

    cout << summary;

    return 0;
    }
```

6.7 Tymczasowe pominięcia

Poniższe elementarne instrukcje:

- **`switch`**
- **`for`**
- **`break`**
- **`continue`**
- **`goto`**

zostaną omówione osobno w dalszej części materiału.

7 Wprowadzenie do tablic

7.1 Inspiracje

- *Pytanie:*
  Co zrobić, gdy konieczne jest operowanie wieloma zmiennymi? Setkami, tysiącami, setkami tysięcy?
- *Można określić nazwy setek tysięcy pojedynczych zmiennych, ale wtedy każda operacja musiałaby odnosić się do każdej zmiennej z osobna, bez możliwości użycia pętli.*
- Wnioski:
  - Konieczna wspólna nazwa dla określenia wielu zmiennych składających się na jedną całość …
  - … wraz z możliwością posługiwania się zmienną składową poprzez nazwę całości oraz numer składowej.
  - **Wynalazek nazywany tablicą spełnia oba powyższe wymagania.**

7.2 Generalia

7.2.1 Pojęcie tablicy

- W ogólnym programowaniu tablica jest pojęciem złożonym, ściśle definiowalnym i różnie obsługiwanym. Więcej w dalszej części materiału.
- Tymczasowo przyjmujemy mocno uproszczone określenie tablicy:

Tablicą nazywamy ciąg ustalonej ilości numerowanych zmiennych jednego typu.

7.3 Deklaracja

7.3.1 Postać ogólna

W języku C/C++ deklaracja tablicy obejmuje kolejno:

1. Identyfikator typu danych.
2. Nazwę tablicy.
3. Umieszczone w prostokątnych nawiasach wyrażenie określające liczbę wszystkich elementów tablicy nazywaną **rozmiarem tablicy** (lub **długością tablicy**).
4. Znak średnika.

```
1 int main () {
2     float array [ 20 ];
3     // tablica wartości typu float o nazwie
4     // array i rozmiarze dwudziestu elementów
5
6     return 0;
7     }
```

7.3.2 Określenie rozmiaru tablicy

- Określenie rozmiaru tablicy:
  - Musi być wyrażeniem o wartości całkowitej dodatniej.
  - Może być puste, ale wtedy rozmiar tablicy jest ustalany niejawnie poprzez szczególną inicjalizację, więcej w dalszej części wykładu, jednakże …
  - … pusta deklaracja rozmiaru bez inicjalizacji skutkuje błędem kompilacji.
- Maksymalna możliwa wartość rozmiaru tablicy zależy w różnym stopniu od wielu czynników w rodzaju typu komputera, systemu operacyjnego, ustawień kompilatora, aktualnie dostępnej pamięci i kilku innych.

Uwaga:

- *Wybrane kompilatory (na przykład używany najpowszechniej na całym świecie kompilator gpp) dopuszczają zmienne w wyrażeniu określającym rozmiar tablicy.*
- *Opisana możliwość nie mieści się w standardzie języka C++, ale popularność kompilatora gpp czyni ją nieformalnym standardem.*
- *Poprawność działania kodu z zakresem tablicy określonym z użyciem zmiennej bywa przypadkowa, jest przyczyną wielu błędów (szczególnie na początku nauki programowania) i dlatego nie należy się kurczowo przywiązywać do opisanej możliwości.*

Kod źródłowy dla demonstracji deklaracji długości tablicy.

```
1  int main () {
2      int arrayOne [ 3 * 4 + 1 ];  // dobrze
3      float arrayTwo [ 4.0 ];      // źle
4      int arrayThree [ -1 ];       // źle
5      float arrayFour [ 0 ];        // źle
6      int arrayFive [ "1" ];        // źle
7      int arraySix [];              // źle
8
9      return 0;
10     }
```

7.4 Stosowanie

7.4.1 Generalia

- Jeżeli deklaracja tablicy się powiedzie, do dyspozycji programu jest dokładanie tyle zmiennych ile wynosi rozmiar tablicy.
- Wszystkie zmienne z tablicy są typu określonego deklaracją.
- Każdy element tablicy może być elementem wyrażenia, może przyjmować wartość w instrukcji podstawienia, możne przekazać wartość na wyjście lub wczytać wartość z wejścia.
- Konkretną pojedynczą składową zmienną będącą elementem tablicy określa sekwencją obejmująca:
  1. Nazwę tablicy
  2. Wyrażenie całkowite w prostokątnych nawiasach określające numer (indeks) konkretnej pojedynczej zmiennej.
- **W języku C/C++ składowe tablicy są numerowane począwszy od zera.**

Kod źródłowy dla demonstracji stosowania elementów tablicy.

```cpp
#include <iostream>
using namespace std;

int main () {
    int a [ 5 ];
    // dostęp do elementów a [ 0 ], a [ 1 ], a [ 2 ], a [ 3 ], a [ 4 ]
    int i;
    int summary = 0;

    a [ 0 ] = 2;
    cin >> a [ 1 ];

    a [ 2 ] = 2 * a [ 1 ] - a [ 0 ];

    i = 3;
    a [ i ] = 2 * a [ i - 1 ] - a [ i - 2 ];
    i = i + 1;
    a [ i ] = 2 * a [ i - 1 ] - a [ i - 2 ];

    i = 0;
    while ( i <= 4 ) {
        cout << "a [ " << i << " ] = " << a [ i ] << endl;
        summary = summary + a [ i ];
        i = i + 1;
        }

    cout << endl << "suma = " << summary << endl;

    return 0;
    }
```

7.4.2 Kontrola zakresu

- W przeciwieństwie do wielu innych języków programowania, kompilatory języka C/C++ nie kontrolują zawierania użytego indeksu w zakresie rozmiaru tablicy.
- **Użycie wartości indeksu spoza zakresu rozmiaru tablicy nie jest sygnalizowane przez kompilator**, dlatego …
- … użycie wartości indeksu poniżej lub powyżej zakresu wynikającego z rozmiaru może powodować:
  - **Zmianę wartości zmiennych pozornie nieużywanych**.
  - Każdorazowe jawnie błędne działanie programu (sytuacja korzystna z racji możliwości detekcji błędu).
  - Błędne działanie niewidoczne w przeprowadzonych testach i objawiane w zupełnie innym fragmencie programu lub w nieprzewidzianych okolicznościach (sytuacja najgorsza).
  - Przypadkowy brak problemów.

Kod źródłowy dla demonstracji braku kontroli zakresu tablic

(dla kompilatora gpp).

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int array [ 3 ];
6      int i = 777;
7
8      cin >> i;
9      cout << "przed podstawieniem i = " << i << endl;
10     array [ -1 ] = 111;
11     cout << "po podstawieniu i = " << i << endl;
12
13     return 0;
14     }
```

7.4.3 Indeksowanie

- Indeksowanie tablicy począwszy od zera nie jest udziwnieniem, ale odzwierciedleniem zgodności programowania z praktyczną realizacją komputerów.
- Informatycy naturalnie liczą począwszy od zera z racji sprzętowej realizacji tablic w komputerach.
- W innych językach programowania początki indeksowania:
  - Często zaczynają się od jedności (naturalne dla nie informatyków).
  - Pozwalają rozpoczynać indeksowanie od dowolnie ustalonego numeru, nawet ujemnego.
- **Wszystkie języki programowania indeksują elementy kolejnymi wartościami.**
- Algorytmika (oraz mniej restrykcyjnie ogólna informatyka) częściej posługuje się indeksowaniem tablic od 1.

7.5 Inicjalizacja

- Tablice mogą być inicjowane przy deklaracji.
- Inicjalizacja elementów tablicy polega na umieszczeniu po deklaracji tablicy i przed końcowym średnikiem, zawartego w klamrowych nawiasach ciągu wartości tablicy oddzielonych przecinkami.
- Liczba elementów inicjalizacji może być mniejsza od rozmiaru tablicy i inicjowane są tylko początkowe elementy.
- Elementy poza jawną inicjalizacją przybierają wartości domyślne, najczęściej zależne od typu odpowiedniki wartości zerowej.
- Liczba inicjowanych elementów nie może być większa od deklarowanego rozmiaru tablicy.
- Przy wykorzystaniu inicjalizacji wyrażenie określające rozmiar tablicy może być puste (prawy prostokątny nawias bezpośrednio po lewym) i wtedy rozmiar tablicy jest równy liczbie elementów inicjalizacji.
- Tablice z nieokreślonym zakresem muszą być inicjowane.
- **Tablice bez inicjalizacji przyjmują przypadkowe wartości elementów.**

Kod źródłowy dla demonstracji inicjalizacji tablic.

```cpp
#include <iostream>
using namespace std;

int main () {
    int arrayOne [ 3 ] = { 1, 2, 3 }; // poprawnie

    int i = 0;
    while ( i < 3 ) {
        cout << arrayOne [ i ] << " ";
        i = i + 1;
        }
    cout << endl;

    float arrayTwo [ 2 ] = { 1.1, 2.2, 3.3 }; // sprzeczność, źle

    float arrayThree [ 4 ] = { 1.11, 2.22 };
    // arrayThree [ 0 ] == 1.11, arrayThree [ 1 ] == 2.22,
    // arrayThree [ 2 ] == 0.0, arrayThree [ 3 ] == 0.0
    i = 0;
    while ( i < 4 ) {
        cout << arrayThree [ i ] << " ";
        i = i + 1;
        }
    cout << endl;

    int arrayFour [] = { 1, 2, 3, 4 };
    // równoważne int arrayFour [ 4 ] = { 1, 2, 3, 4 };

    //int arrayFive []; // błąd kompilacji

    int arraySix [ 4 ];
    i = 0;
    while ( i < 4 ) {
        cout << arraySix [ i ] << " ";
        i = i + 1;
        }
    cout << endl;

    return 0;
    }
```

7.6 Wady

- Największą wadą tablic jest konieczność określenia liczby elementów dla jej utworzenia, niemożliwej do zmiany w trakcie działania programu. Tym samym …
- … tablica może okazać się za krótka na potrzeby programu, zaś …
- … odpowiednio długa tablica (przewidziana z nadmiarem w stosunku do możliwej ilości danych) omijając problem ograniczoności, generuje ogromny program względem być może minimalnych potrzeb konkretnego uruchomienia.
- Rozwiązaniem jest tworzenie tablicy o liczbie elementów ustalanej każdorazowo w zależności od potrzeb i z możliwością zmiany długości, o czym w dalszej części materiału.
- Uwaga:
  - Nie istnieje żaden magiczny wynalazek pozwalający bez kosztów zwiększyć przewidzianą w deklaracji długość tablicy podczas jej używania.
  - Różne wynalazki programistyczne (na przykład `vector`) pozwalające dowolnie wydłużać ilość danych w trakcie działania i nawet posługujące się w dostępie do elementów prostokątnymi nawiasami (przypominając tablice) NIE SĄ TABLICAMI.

Część II

Uzupełnienie podstaw

8 Uzupełnienie typów

8.1 Typy liczbowe

8.2 Typ liczb całkowitych

- Typ liczb całkowitych jest identyfikowany słowem **`int`**.
- Wartości zapisywane są jak w powszechnym użyciu, ale …
- … istnieje wiele innych możliwości zapisu liczb całkowitych przedstawionych w dalszej części materiału.
- Słowo **`int`** może być poprzedzone jednym z kilku innych słów (na przykład **`short`** lub **`long`**) stanowiących tak zwane ***modyfikatory typu*** opisane w dalszej części wykładu.

8.3 Typ liczb rzeczywistych

- Typ liczb rzeczywistych jest identyfikowany słowem **`float`** lub słowem **`double`**.
- Różnica między typem **`float`** a typem **`double`** sprowadza się do zakresu i dokładności stosowanych liczb, zajętości w pamięci komputera oraz szybkości działania - więcej na przedmiocie *Wstęp do informatyki* i w dalszej części wykładu.
- Wartości typu rzeczywistego zapisywane są jak w powszechnym użyciu, ale z zastąpieniem znaku przecinka przez znak kropki.
- Możliwych jest wiele dodatkowych zapisów liczb rzeczywistych, więcej w dalszej części wykładu.
- Typ rzeczywisty również przewiduje poprzedzanie słów **`float`** oraz **`double`** modyfikatorami typów.

8.4 Typ znakowy

- Typ znakowy jest identyfikowany słowem **char**.
- Obejmuje pojedyncze znaki w rozumieniu znaków liter, znaków cyfr, znaków specjalnych i innych.
- Wartości typu znakowego nie mogą być stosowane w rozumieniu zapisów pojedynczych liter lub cyfr, ponieważ myliłyby się na przykład z jednoliterowymi zmiennymi lub jednocyfrowymi liczbami. Stąd konieczność specjalnego opisywania pojedynczych znaków i dlatego …
- … w języku C/C++ wartości typu znakowego muszą być obłożone znakami apostrofu (potocznie pojedynczego apostrofu), wymuszając postać:

  '*znak*'

  gdzie *znak* jest pojedynczym znakiem dostępnym z klawiatury.
- Istnieje wiele dodatkowych sposobów zapisu wartości typu znakowego.
- Typ znakowy posiada własne modyfikatory typu.
- W rzeczywistości typ znakowy to szczególna wersja typu nieujemnych liczb całkowitych szczególnie traktowanych przy wyświetlaniu i szczególnej formie używania wartości.

8.5 Kod źródłowy dla prezentacji typu znakowego.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      char c1 = '1';
6      char cA = 'A';
7      char cHash = '#';
8      char cSpace = ' ';
9
10     cout << c1 << c_A << 'b';
11     cout << '3' << cHash << "->" << cSpace << "<-" << endl;
12
14     c1 = 49;
15     char c0 = 49 - 1;
16     char cC = cA + 2;
17     cout << c1 << c0 << cC << endl;
18
19     char c;
20     cin >> c;
21     cout << c << endl;
22
23     return 0;
24     }
```

8.6 Typ logiczny

- W standardzie języka C typ logiczny nie występuje i nawet dla niezbyt zaawansowanych programistów nie jest konieczny. Jednakże …
- … kod źródłowy jest czytelniejszy przy dostępnym typie logicznym.
- Standard języka C/C++ przewiduje, że każda wartość złożona wyłącznie z bitów `0` jest równoważna wartości fałszu, zaś każda wartość z przynajmniej jednym bitem `1` jest równoważna wartości prawdy.
- W języku C++ wprowadzono typ logiczny identyfikowany słowem **`bool`**.
- Wraz z nazwą typu wprowadzono identyfikatory wartości typu, to jest słowo **`true`** (prawda) oraz słowo **`false`** (fałsz).
- Przy wczytywaniu wartości logicznych bez szczególnych zabiegów obowiązuje ich prawdziwa natura, czyli `1` oraz `0` dla odpowiednio **`true`** oraz **`false`**.
- Domyślnie i bez specjalnych zabiegów wartości **`true`** oraz **`false`** są wyświetlane w postaci `1` oraz `0`.
- Typ logiczny jest szczególnie traktowanym typem całkowitym.

Kod źródłowy dla prezentacji typu logicznego.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      bool bTrue = true;
6      bool bFalse = false;
7
8      cout << "bTrue = " << bTrue << endl;
9      cout << "bFalse = " << bFalse << endl << endl;
10
11     bFalse = bFalse - bTrue;
12     cout << "bFalse = " << bFalse << endl;
13
14     bFalse = bTrue / bTrue;
15     cout << "bFalse = " << bFalse << endl;
16
17     cout << "bTrue + bTrue = " << bTrue + bTrue << endl;
18
19     bTrue = bTrue + bTrue + bTrue;
20     cout << "bTrue = " << bTrue << endl;
21
22     bTrue = bTrue - 1;
23     cout << "bTrue = " << bTrue << endl;
24
25     return 0;
26     }
27
```

9 Wyrażenia i operatory

9.1 Wyrażenia

9.1.1 Generalia

- Wyrażenie to jedno z ważniejszych pojęć w informatyce, a jeszcze ważniejsze w programowaniu.
- Istnieje wiele rodzajów wyrażeń poznawanych w trakcie studiów.
- W różnych językach programowania występują specjalne (wyłącznie programistyczne) klasyfikacje wyrażeń.
- Tymczasowo przyjmujemy przedstawioną dalej wersję pojęcia wyrażenia bardziej formalną niż dotychczasowe potoczne rozumienie, wciąż jednak mocno uproszczoną.

9.1.2 Definicja pojęcia wyrażenie

- Dla określenia pojęcia wyrażenia przyjmujemy, że znane są pojęcia:
  - **stała** - napis o ustalonej niezmiennej wartości.
  - **zmienna** - napis oznaczający nazwę zmiennej zdefiniowanej jak w programowaniu.
  - **działanie** - napis oznaczający związanie z ciągiem argumentów ciągu wartości - więcej na przedmiotach matematycznych i na przedmiocie *Wstęp do informatyki*.
- **Wyrażenie** to napis spełniający jeden z poniższych warunków:
  - Każda pojedyncza stała jest wyrażeniem.
  - Każda pojedyncza zmienna jest wyrażeniem.
  - Wyrażeniem jest zapis każdego działania z argumentami będącymi wyrażeniami.

9.1.3 Pułapki wyrażeń

- Kod źródłowy dla demonstracji pułapek wyrażeń.

```cpp
#include <iostream>
using namespace std;

int main () {
    int i = 34.57 + 123.33;
    cout << i << endl; // 157

    char c = -80 + 5.91 + i;
    cout << c << endl; // R

    i = '3' + '7' + c;
    cout << i << endl; // 188

    float f = '\n' + 3.7 + 2 + i - c; // 121.7
    cout << f << endl;

    return 0;
    }
```

- W przeciwieństwie do matematyki, gdzie wartość wyrażenia może występować sama w sobie, w programowaniu z racji wymogu podstawienia ostateczna wartość jest ściśle zależna od typu zmiennej przyjmującej wartość.
- Nieznajomość mechanizmów wartościowania wyrażeń w języku C/C++ bywa przyczyną problemów na początku nauki.
- Przedstawione przykłady braku błędu (pozornie samoczynnego poprawnego działania) nie są dobrodziejstwem, a wprost przeciwnie, mankamentem języka C/C++.
- Istnieją języki programowania ze ścisłą kontrolą typów składowych wyrażeń, gdzie powyższe wyrażenie powodują błąd. Jednakże trudniejsze tworzenie kodu jest dobrodziejstwem, ponieważ pozwala uniknąć błędów.

9.2 Ważne wstępne pojęcia

9.2.1 Operator i operand

Dla precyzyjności wiedzy wprowadzamy (w uproszczonej wersji, więcej na przedmiocie *Wstęp do informatyki*) wybrane pojęcia ogólnie wiedzy o wyrażeniach:

- **operand - napis określający obiekt podlegający działaniu**
  Na przykład składniki wyrażenia opisującego działanie sumy lub czynniki wyrażenia opisującego działanie mnożenia.
- **operator - napis oznaczający rodzaj działania**
  Na przykład szkolne znaki działań arytmetycznych.

9.2.2 Krotność operatora

Podobnie dla uproszczenia nazewnictwa przyjmijmy specjalne nazwy dla operatorów zależne od ilość operandów uczestniczących w działaniu (tak zwanej krotności operatora). I tak:

- operator **binarny** - (czyli dwuargumentowy) działający na dwu operandach (jak większość szkolnych działań)
- operator **unarny** - (czyli jednoargumentowy) działający na jednym operandzie (jak szkolny znak ujemności)
- operator **ternarny** - (czyli trójargumentowy) działający na trzech operandach - tymczasowo mogący wydawać się nieco dziwnym.

9.2.3 Umiejscowienie operatora względem operandów

Kolejnym uproszczeniem nazewnictwa będzie przyjęcie rozróżniania operatorów z punktu widzenia ich umiejscowienia względem operandów, prowadzące do pojęć:

- operatora **infiksowego** - umieszczanego między operandami (jak większość szkolnych operatorów).
- operatora **prefiksowego** - umieszczanego przed operandami (jak szkolny minus).
- operatora **postfiksowego** - umieszczanego po operandach (brak szkolnego zastosowania).

9.3 Ogólności operatorów

- W matematyce, informatyce i programowaniu operatory są zagadnieniem bardziej skomplikowanym niż operatory znanych ze szkoły działań arytmetycznych.
- W zapisie wyrażeń operator nie musi być umieszczony w środku i równie dobrze może znajdować się przed operandami jak i po operandach.
- Nie zawsze operand jest jednoznakowy.
- Bywają operatory niebędące spójnym napisem.
- Ilość operandów w działaniu może być dowolnie duża.
- Szczegółowo zagadnienie omawiane będzie na przedmiocie *Wstęp do informatyki*.

- Języki programowania zawsze przewidują pewną ilość predefiniowanych operatorów o stałym znaczeniu.
- Wybrane języki programowania (w tym C++) pozwalają nadawać nowe znaczenia istniejącym operatorom, a nawet tworzyć własne operatory.
- W zależności od wersji język C obejmuje około 50 operatorów, zaś język C++ około 60. Z różnym stopniem powszechności stosowania.
- Operatory w programowaniu dzieli się na różnego rodzaju grupy z niejednoznacznością i subiektywnością podziału.

9.4 Podstawowe operatory języka C/C++

9.4.1 Operator podstawienia

- W algorytmice i w ślad za algorytmiką w wielu językach programowania akcja/działanie/instrukcja nie oznacza zaistnienia wartości.
- W algorytmice znak podstawienia będąc elementem instrukcji podstawienia nie jest operatorem.
- Ponieważ na poziomie diagnostyczno/technologiczno/sprzętowym każde działanie posiada odzwierciedlenie w pamięci komputera, język C/C++ wraz z językami pochodnymi traktuje niemal każdą akcję jak wyrażenie reprezentujące wartość.
- Instrukcja podstawienia również pozostawia wartość, dlatego podstawienie jest działaniem, a znak podstawienia operatorem działania postawienia.
- **Wartością operacji podstawienia jest wartość prawej strony instrukcji podstawienia.**
- Podstawienie będące operatorem jest stosunkowo rzadką cechą języków programowania. Jeżeli występuje, najczęściej przekazuje informację o poprawności podstawienia.

Kod źródłowy dla demonstracji działania instrukcji podstawienia

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6      float f;
7      char c;
8      bool b;
9
10     // nawiasy są konieczne !!!
11     cout << ( i = 234 ) << endl;
12     cout << ( f = 56.78 ) << endl;
14     cout << ( c = 'X' ) << endl;
15     cout << ( b = true ) << endl;
16
17     return 0;
18     }
```

9.4.2 Operatory arytmetyczne

9.4.2.1. Wprowadzenie

- W algorytmice i w ślad za algorytmiką w wielu językach programowania akcja/działanie/instrukcja nie oznacza zaistnienia wartości.
- Wszystkie języki programowania (również język C/C++) umożliwiają znane ze szkoły działania arytmetyczne na liczbach.
- Najczęściej oznaczenia operatorów działań są zgodne ze stosowanymi powszechnie, ale nie jest to reguła powszechna. Operatorem mnożenia jest powszechnie znak * (znak · nie jest łatwo dostępny z klawiatury), zaś operatorem dzielenia jest znak / (znak : pełni najczęściej inna rolę, zaś znak ÷ również nie jest łatwo dostępny z klawiatury).
- W języku C/C++ pewnych znanych ze szkoły działań arytmetycznych po prostu nie ma (na przykład potęgowania, chociaż zdarzają się w innych językach programowania), zaś inne operatory (na przykład pierwiastkowania) są dostępne w szczególny sposób.
- Języki programowania często udostępniają działania arytmetyczne bardzo przydatne, ale nieznane w szkole (na przykład działanie modulo).
- W przeciwieństwie do matematyki, w informatyce (w programowaniu) znane operacje arytmetyczne mogą dawać nieoczekiwane (na początku nauki) rezultaty. Stąd rozłożenie operatorów arytmetycznych osobno dla liczb całkowitych i rzeczywistych.

9.4.2.2. Operatory liczb całkowitych

- Język programowania C/C++ przewiduje dla liczba całkowitych sześć działań arytmetycznych stosujących pięć operatorów binarnych oraz jeden operator unarny, jak w poniższej tabeli:

Operator	nazwa	opis i uwagi
+	suma	Szkolna suma liczb całkowitych
−	różnica	Szkolna różnica liczb całkowitych
*	iloczyn	Szkolny iloczyn liczb całkowitych
/	iloraz	Iloraz CAŁKOWITY! Wylicza, ile razy drugi argument mieści się w pierwszym. NIE ZAOKRĄGLA!
%	modulo	Reszta z dzielenia całkowitego pierwszego argumentu przez drugi
−	przeciwność	Operator unarny, zmienia znak argumentu na przeciwny

- Z różnych powodów (więcej na przedmiocie *Wstęp do informatyki*) wyrażenia całkowite mogą dawać zaskakujące (tymczasowo) wyniki.

Kod źródłowy dla demonstracji działania operatorów arytmetycznych liczba całkowitych:

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int a; // pierwsza liczba
6      int b; // druga liczba
7
8      cin >> a; // wczytanie pierwszej liczby
9      cin >> b; // wczytanie drugiej liczby
10
11     // wypisanie wyników
12     cout << a << " + " << b << " = " << a + b << endl;
14     cout << a << " - " << b << " = " << a - b << endl;
15     cout << a << " * " << b << " = " << a * b << endl;
16     cout << a << " / " << b << " = " << a / b << endl;
17     cout << a << " % " << b << " = " << a % b << endl;
18     cout << " - " << a << " = " << -a << endl;
19
20     return 0;
21     }
```

9.4.2.3. Operatory liczb rzeczywistych

- Język programowania C/C++ przewiduje dla liczba rzeczywistych pięć standardowych działań arytmetycznych z czterema operatorami binarnymi i jednym operatorem unarnym o działaniach zgodnych z powszechnym rozumieniem.
- Oznaczenia operatorów są identyczne jak dla liczb całkowitych, to znaczy znaki +, −, *, / dla binarnych działań dodawania, odejmowania, mnożenia i dzielenia, oraz znak – dla działania przeciwności.
- Z racji faktycznego braku na komputerach prawdziwych liczb rzeczywistych wyniki działań mogą się różnić od wartości matematycznych, więcej na *Wstępie do informatyki*.

Kod źródłowy dla demonstracji działania operatorów arytmetycznych dla liczb rzeczywistych

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      double f; // pierwsza liczba
6      double g; // druga liczba
7
8      cin >> f; // wczytanie pierwszej liczby
9      cin >> g; // wczytanie drugiej liczby
10
11     // wypisanie wyników
12     cout << f << " + " << g << " = " << f + g << endl;
14     cout << f << " - " << g << " = " << f - g << endl;
15     cout << f << " * " << g << " = " << f * g << endl;
16     cout << f << " / " << g << " = " << f / g << endl;
16     cout << " - " << f << " = " << -f << endl;
17
18     return 0;
19     }
```

9.4.3 Operatory relacyjne

- Język programowania C/C++ przewiduje standardowe dla niemal wszystkich języków programowania działania relacji silnej mniejszości oraz silnej większości używające powszechnie znanych operatorów < oraz >.
- Ponieważ standardowa klawiatura nie przewiduje znaków ⩽ oraz ⩾ dlatego w działaniach słabej mniejszości i słabej większości stosowane są różne wieloznakowe konwencje. W języku C/C++ dwuznakowe sekwencje <= oraz >= oznaczają odpowiednio operator słabej mniejszość oraz słabej większość.
- Operatorem równości w języku C/C++ jest sekwencja DWÓCH znaków równości ==. W momencie wprowadzenia operator == wydawał się wynaturzeniem i marnowaniem istniejącego „gotowego" operatora, ale zastosowane rozwiązania w stosunku do alternatywnej rezygnacji z operatora podstawienia w postaci = wykazuje znaczące korzyści.
- Operatorem różności jest sekwencja !=.

Kod źródłowy dla demonstracji działania operatorów arytmetycznych dla liczb rzeczywistych.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      float f; // pierwsza liczba // 12345678901234567.4
6      float g; // druga liczba    // 12345678901234567.6
7
8      cin >> f; // wczytanie pierwszej liczby
9      cin >> g; // wczytanie drugiej liczby
10
11     cout << f << " < " << g << " ? " << ( f < g ) << endl;
12     cout << f << " <= " << g << " ? " << ( f <= g ) << endl;
14     cout << f << " == " << g << " ? " << ( f == g ) << endl;
15     cout << f << " >= " << g << " ? " << ( f >= g ) << endl;
16     cout << f << " > " << g << " ? " << ( f > g ) << endl;
16     cout << f << " != " << g << " ? " << ( f != g ) << endl;
17
18     return 0;
19     }
```

9.4.4 Operatory logiczne

- Język programowania C/C++ przewiduje poniższe działania logiczne:
  - **Koniunkcję logiczną** - z binarnym operatorem w postaci dwuznakowej sekwencji && (znak & nazywany jest *et*, a poprawniej: handlowe „i").
  - **Alternatywę logiczną** - z binarnym operatorem w postaci dwuznakowej sekwencji || (znak | jest nazywany z angielska *pipe*, a poprawniej pionową kreską).
  - **Negację logiczną** - z unarnym operatorem w postaci znaku wykrzyknika !.
  - **Logiczne równość** oraz **różność** - używające operatorów jak dla typów liczbowych.
- W języku C/C++ nie występuje przewidywane w wielu językach programowania działanie logicznej alternatywy rozłącznej.
- Ważnym zagadnieniem w informatyce (szczególnie ważnym w programowaniu) są (niewystępujące w matematyce) bitowe operatory logiczne - więcej w dalszej części wykładu.

Kod źródłowy dla demonstracji działania operatorów logicznych.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      bool a; // pierwsza wartość
6      bool b; // druga wartość
7
8      cin >> a; // wczytanie pierwszej wartości
9      cin >> b; // wczytanie drugiej wartości
10
11     // wypisanie wyników
12     cout << a << " && " << b << " ? " << ( a && b ) << endl;
14     cout << a << " || " << b << " ? " << ( a || b ) << endl;
15     cout << "! " << a << " ? " << ! a << endl;
16     cout << "! " << b << " ? " << ! b << endl;
16
17     return 0;
18     }
```

9.5 Hierarchia operatorów, priorytety i nawiasy

9.5.1 Wprowadzające przykłady

- Rozważmy wyrażenie:

$$3 * 7 + 2 * 5;$$

Jaka będzie wartość wyrażenia?

- Zgodnie ze szkolną kolejnością działań (najpierw mnożenia, później dodawanie) powinniśmy otrzymać:

$$3 * 7 = 21,\ 2 * 5 = 10,\ 21 + 10 = 31$$

Zgadza się, wynik działania jest zgodny z nawiasowaniem:

$$(3 * 7) + (2 * 5)$$

- Rozważmy inne wyrażenie:

$$3 * 7 / 2 * 5$$

Jaka będzie wartość wyrażenia?

- Zgodnie ze szkolną kolejnością działań (najpierw mnożenia, później dzielenia) powinniśmy otrzymać: $3 * 7 = 21$, $2 * 5 = 10$, $21 / 10 = 2$. Ale wyświetlany wynik: 50! *Gdzie jest problem?*
- Wersja wyrażenia zgodna z zamierzeniami szkolnego wyliczania:

$$(3 * 7) / (2 * 5)$$

Wyświetlany wynik 2!

- Wersja wyrażenia jawnie zgodna z działaniem komputera:

$$((3 * 7) / 2) * 5$$

Wyświetlany wynik 50!

- Jeszcze inna kolejność działań:

$$3 * ((7 / 2) * 5)$$

Wyświetlany wynik 45!

Kod źródłowy dla demonstracji wyliczania wyrażeń arytmetycznych.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      cout << "3 * 7 + 2 * 5 = ";
6      cout <<  3 * 7 + 2 * 5 << endl;
7
8      cout << "( 3 * 7 ) + ( 2 * 5 ) = ";
9      cout <<  ( 3 * 7 ) + ( 2 * 5 ) << endl;
10
11     cout << endl;
12
13     cout << "3 * 7 / 2 * 5 = ";
14     cout <<  3 * 7 / 2 * 5 << endl;
15
16     cout << "( 3 * 7 ) / ( 2 * 5 ) = ";
17     cout <<  ( 3 * 7 ) / ( 2 * 5 ) << endl;
18
19     cout << "( ( 3 * 7 ) / 2 ) * 5 = ";
20     cout <<  ( ( 3 * 7 ) / 2 ) * 5 << endl;
21
22     cout << "3 * ( ( 7 / 2 ) * 5 ) = ";
23     cout <<  3 * ( ( 7 / 2 ) * 5 ) << endl;
24
25     return 0;
26     }
```

- Rozważmy inne wyrażenie:

true && **true** || **false** && **false**

Jaka jest jego wartość?

- *Zgodna z ideą wcześniejszego przykładu?*

```
( ( true && true ) || false ) && false
```

czyli **false***?*

- Nie!!! Stosowana kolejność:

```
( true && true ) || ( false && false )
```

czyli **true**!!!

Kod źródłowy dla demonstracji wyliczania wyrażeń logicznych.

```
#include <iostream>
using namespace std;

int main () {
    cout << boolalpha; // słowne wypisanie wartości typu logicznego

    cout <<   "true && true || false && false = ";
    cout  << ( true && true || false && false ) << endl;

    cout <<   "( ( true && true ) || false ) && false = ";
    cout  << ( ( ( true && true ) || false ) && false ) << endl;

    cout <<   "( ( true && true ) || ( false && false ) = ";
    cout  << ( ( true && true ) || ( false && false ) ) << endl;

    return 0;
    }
```

9.5.2 Wnioski z przykładów

- W programowaniu, szkolna wiedza dotycząca kolejności działań w wyrażeniach jest przydatna w ograniczonym stopniu, wyłącznie na zasadzie istnienia zjawiska. Zajmuje się tylko arytmetyką, nie zajmując się innymi działaniami - na przykład logicznymi, czy tez łączeniem arytmetyki z logiką.
- **Programistyczna kolejność działań (nawet arytmetycznych) nie musi być zgodna ze szkolną kolejnością.**
- Można zmusić kompilator do szkolnego rozumienia kolejności działań, ale z różnych powodów optymalniejsze jest zastosowanie innych rozwiązań.
- Ogólnie rozumiana „kolejność działań” w programowaniu nazywana jest **hierarchią operatorów**, zaś wzajemna kolejność stosowania operatorów bazuje na wartości liczbowej zwanej **priorytetem operatora**.
- Programistyczna hierarchia operatorów nie może obejmować tylko operatorów arytmetycznych, ale musi obejmować wszystkie możliwe operatory w liczbie kilkudziesięciu.
- Pełna hierarchia operatorów języka C/C++ operatorów jest sporą tabelą, dostępną w dokumentacji języków C/C++.
- W programowaniu (w przeciwieństwie do szkoły) operatory mogą mieć równoważny priorytet.
- **Nawiasy dla poprawnego wyliczania wyrażeń mogą być konieczne!!!**
- Więcej o hierarchii operatorów w dalszej części wykładu i na przedmiocie *Wstęp do informatyki*.

9.5.3 Hierarchia operatorów i reguły ewaluacji

- Hierarcha dotychczas używanych operatorów:

operator	poziom priorytetu (niższa wartość oznacza wyższy priorytet)
`!` `-(unarny)`	3
`*` `/` `%`	5
`+` `-(binarny)`	6
`<` `>` `<=` `>=`	8
`==` `!=`	9
`&&`	13
`||`	14
`=`	16

- Ogólna reguła wyliczania (ewaluacji) wyrażeń w języku C/C++ sprowadza się do warunków:
  1. Wyliczenie przebiega zgodnie z priorytetem operatorów.
  2. W przypadku operatorów o tym samym priorytecie o kolejności działań decyduje kolejność zapisu w wyrażeniu.

9.5.4 Nawiasy

- Możliwym sposobem unikania problemów wynikłych z hierarchii operatorów jest pamiętanie wszystkich priorytetów, ale:
  - Ludzka pamięć jest zawodna.
  - W innych językach programowania priorytety tych samych operatorów mogą być inne, zwiększając podatność na błędy.
  - **Istnieje prosty sposób uniknięcia problemów z hierarchią operatorów, czyli NAWIASY!!!**
  - **Notacja bez nawiasów, nawet będąc poprawną nie pokazuje sposobu wyliczania wyrażenia.**
- Ważne prawdy dotyczące nawiasów:

Nawiasów jest za dużo tylko, gdy są powtórzone.

Dodatkowa para nawiasów kosztuje mniej niż błąd!!!

10 Uzupełnienie instrukcji

10.1 Wprowadzenie

- Z angielskojęzycznej Wikipedii:

> *In computer programming a* ***statement*** *is the smallest standalone element of an imperative programming language that expresses some action to be carried out.*

- W swobodnym tłumaczeniu:

> W programowaniu komputerowym **instrukcja** jest najmniejszym samodzielnym (niepodzielnym) elementem imperatywnych języków programowania wyrażającym działanie do wykonania.

- *Imperatywny język programowania?* W dużym uproszczeniu - język programowania z instrukcjami zmieniającymi stan komputera z istniejącego na inny.
- **Programowanie imperatywne** (w imperatywnym języku programowania) - jeden z paradygmatów programowania.
- **Paradygmat programowania** - wybrany zbiór pojęć, teorii, zaleceń i dążeń przestrzeganych, preferowanych i oczekiwanych w danym języku programowania.
- Programowanie imperatywne (jak w języku C/C++) nie jest jedynym w praktyce stosowanym sposobem programowania.
- Możliwe sposoby programowania bez używania instrukcji. Na przykład programowanie funkcyjne lub programowanie deklaratywne, więcej na innych przedmiotach.

10.2 Klasyfikacja instrukcji

10.2.1 Złożonościowy podział instrukcji

10.2.1.1. Instrukcje proste

Instrukcje proste są instrukcjami wykonującymi pojedyncze działanie i najczęściej obejmują instrukcje:

- Przypisania (podstawienia) - jak wprowadzona wcześniej.
- Skoku - przerwanie aktualnie wykonywanej sekwencji instrukcji i przejście do wykonywania innego fragmentu programu, więcej w aktualnym rozdziale.
- Wywołania podprogramu, opisana w dalszej części materiału.
- Powrotu - … z wnętrza fragmentu kodu wywołanego zewnętrznym poleceniem do miejsca wywołania. Więcej w dalszej części materiału.
- Pusta - niepowodująca żadnych działań, wbrew pozorom istnieją języki programowania z istotnym znaczeniem instrukcji pustej.

10.2.1.2. Instrukcje złożone

Instrukcje złożone (zwane też **instrukcjami strukturalnymi**) obejmują więcej niż jedno działanie, pozostając niepodzielnymi w rozumieniu całości w zgodzie z najbardziej ogólną definicją instrukcji. Najczęściej obejmują instrukcje:

- **Blokową** - grupującą w jedną całość pewną ilość instrukcji, wprowadzona wcześniej.
- **Sterujące** - w rozumieniu sterowania przebiegiem programu. Obejmujące najczęściej instrukcje:
  - **Warunkowe** - uzależniające wykonanie fragmentów kodu od spełnienia warunku (wprowadzone wcześniej).
  - **Pętle** - określających wielokrotne wykonanie czynności (częściowo wprowadzone wcześniej).
  - **Selekcji** (w rozumieniu selekcji sterowania) - dokonujące wyboru jednego działania spośród wielu (więcej niż dwóch) możliwości, a nie tylko na podstawie dwóch wartości logicznych. Więcej w aktualnym rozdziale.

10.2.2 Funkcjonalny podział instrukcji

- Instrukcja blokowa - jak dla podziału złożonościowego.
- Instrukcja podstawienia - jak dla podziału złożonościowego.
- Instrukcje sterujące:
  - warunkowe, …
  - … pętli, …
  - … powrotu, …
  - …, selekcji – jak opisane wyżej.
  - Skoków (opisana w aktualnym rozdziale).
- Wejścia/wyjścia.

10.3 Instrukcja selekcji

10.3.1 Wprowadzenie

- Idea: wybór (selekcja) działania programu spośród więcej niż dwóch możliwości i w zależności od wartości wyrażenia.
- W innych językach programowania instrukcja selekcji ma wiele różnych postaci z wieloma różnicami w działaniu.
- Algorytmika i wiele języków programowania obywa się bez instrukcji selekcji.
- Każdą instrukcję selekcji da się zastąpić odpowiednią liczbą instrukcji warunkowych (zadanie na zajęcia laboratoryjne).

10.3.2 Struktura

1. Obowiązkowe słowo **`switch`**.
2. Umieszczone w okrągłych nawiasach dowolne *wyrażenie*.
3. Otwierający nawias klamrowy.
4. Dowolna liczba określeń działań zależnych od wartości wyrażenia, o ogólnej strukturze postaci:
   a. Obowiązkowe słowo **`case`**.
   b. Ustalona wartość *wyrażenia*.
   c. Obowiązkowy znak dwukropka.
   d. Działanie wykonywane, gdy wartość *wyrażenia* w nawiasie jest równa wartości po słowie **`case`**.
5. W zastępstwie napisu o konstrukcji **`case`** *`wyrażenie`* **`:`** może się opcjonalnie znaleźć nie więcej niż jedna konstrukcja postaci:
   a. Słowo **`default`**.
   b. Obowiązkowy znak dwukropka.
   c. Działanie wykonywane, gdy *wyrażenie* nie osiągnie żadnej z wartości po słowach **`case`**.
6. Zamykający nawias klamrowy.

Kod źródłowy dla demonstracji instrukcji selekcji.

```
#include <iostream>
using namespace std;

int main () {
    int i;
    cin >> i;

    switch ( i ) {
        case 0:
            cout << "zero" << endl;

        case 11:
            cout << "jedenascie" << endl;

        case 100:
            cout << "sto" << endl;

        default:
            cout << "nierozpoznane" << endl;
        }

    return 0;
    }
```

10.3.3 Dodatkowe uwarunkowania

- Uruchomienie ostatniego kodu z podaniem na przykład wartości 11 spowoduje wyświetlenie trzech napisów!!!
- W przeciwieństwie do wielu innych języków programowania, w języku C/C++ wszystkie instrukcje bloku są jedną sekwencją, zatem nie są wykluczającymi się fragmentami.
- Wartości po słowach **`case`** są jedynie miejscem rozpoczęcia wykonywania sekwencji działań. Tym samym, następstwo kolejnego słowa **`case`** lub **`default`** nie powoduje przerwania działania sekwencji.
- Dla ograniczenia wykonania sterowania jedynie do instrukcji związanej z daną wartością po słowie **`case`** konieczne jest dodanie w formie ostatniej „instrukcji" słowa **`break`** powodującego wyjście sterowania poza blok instrukcji.

Kod źródłowy poprawnej wersji zastosowania instrukcji selekcji.

```cpp
#include <iostream>
using namespace std;

int main () {
    int i;
    cin >> i;

    switch ( i ) {
        case 0:
            cout << "zero" << endl;
            break;

        case 11:
            cout << "jedenascie" << endl;
            break;

        case 100:
            cout << "sto" << endl;
            break;

        default:
            cout << "nierozpoznane" << endl;
        }

    return 0;
    }
```

10.4 Pętla for

10.4.1 Wprowadzenie

- Pętla **for** nie jest konieczna dla programowania, ale jest przydatna dla uproszczenia kodu źródłowego.
- W wersji języka C/C++ pętla **for** jest jednym z najgenialniejszych wynalazków programistycznych.
- Pętla **for** jest najpowszechniej używaną pętlą w praktyce programowania.
- Zapis podobny do przykładowego:

```
for ( int i = 0; i <= 10; i++ )
```

  jest zaledwie jedną z wielu możliwych postaci pętli **for**.
- Pętla **for** jest w rzeczywistości uboższą wersją pętli *while*.
- W wielu zastosowaniach informatycznych (na przykład algorytmika) pętla *for* z uwagi na jej ścisły związek z konkretnym językiem programowania bywa wykluczana z możliwości stosowania.

10.4.2 Ogólna struktura

- Ogólna struktura pętli **for** jest opisana trzema instrukcjami składowymi oraz pojedynczym warunkiem, to jest:
  - Instrukcją inicjującą (*inicjalizacja*).
  - Warunkiem końca działania (*koniec*).
  - Zasadniczym powtarzalnym działaniem (*ciało*).
  - Instrukcją automatycznej inkrementacji/dekrementacji (*inkrementacja*).
- Opis słowny:
  Po wykonaniu instrukcji inicjującej, tak długo jak długo warunek jest spełniony wykonuj ciało pętli wraz z następującym każdorazowym uruchomieniem instrukcji inkrementującej/dekrementującej.

- Struktura zapisu:
  1. Słowo **for**.
  2. Otwierający nawias okrągły.
  3. Instrukcja inicjująca z obowiązkowym znakiem średnika – *inicjalizacja*.
  4. Zapis warunku końcowego z obowiązkowym znakiem średnika - *koniec*.
  5. Instrukcji inkrementująca bez średnika – *inkrementacja*.
  6. Zamykający nawias okrągły.
  7. Powtarzana instrukcja - *ciało*.

```
for ( inicjalizacja; koniec; inkrementacja )
    ciało
```

Kod źródłowy dla demonstracji pętli *for*.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6      int silnia;
7
8      cin >> i;
9      cout << i << "! = ";
10
11     for ( silnia = 1; i >= 2; i = i - 1 )
12         silnia = silnia * i;
13
14     cout << silnia << endl;
15
16     return 0;
17     }
```

10.4.3 Dodatkowe uwarunkowania

- Instrukcja inicjująca i instrukcja inkrementująca:
  - Muszą być instrukcjami prostymi (w języku C/C++ nie oznaczając pojedynczej czynności).
  - Inaczej mówiąc, nie mogą być instrukcją sterującą czy blokową, ale mogą być instrukcją podstawienia, wczytania, a nawet warunkiem. Więcej w dalszej części materiału.
- Warunek końca działania pętli również może być instrukcją, o ile jest instrukcją reprezentująca wartość. Więcej w dalszej części materiału.
- Instrukcja ciała pętli może być dowolną instrukcją w najszerszym rozumieniu.
- Każdy z trzech elementów zawartych w okrągłych nawiasach pętli **`for`** może być pusty oznaczając wtedy wartość **`true`**.
- W języku C++ (jednakże nie w czystym języku C) możliwe jest deklarowanie zmiennej w instrukcji inicjalizującej, oznaczając jedną z najczęstszych form użycia w postaci:

```
for ( int i …
```

10.4.4 Związek z pętlą `while`

- Pętla *for* to nic innego jak upraszczający zapis wybranej postaci pętli *while*.
- Zapis postaci:

```
for ( inicjalizacja; koniec; inkrementacja )
    ciało
```

jest równoważny zapisowi:

```
inicjalizacja;
while ( koniec ) {
    ciało;
    inkrementacja;
    }
```

Kod źródłowy dla demonstracji zastąpienia pętli *for* pętlą *while*.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i;
6      int silnia;
7
8      cin >> i;
9      cout << i << "! = ";
10
11     silnia = 1;
12     while ( i >= 2 ) {
13         silnia = silnia * i;
14         i = i - 1;
15         }
16
16     cout << silnia << endl;
17     return 0;
18     }
```

10.5 Instrukcja skoku bezwarunkowego

10.5.1 Wprowadzenie

- Idea: przekazanie sterowania programu we wskazane miejsce programu.
- Wymagane: oznaczenie miejsca skoku i zapisanie rozkazu skoku do miejsca skoku.
- W ogólnym programowaniu miejsce skoku nosi nazwę **etykiety**, jest zwykła nazwą z następującym znakiem dwukropka.
- Nazewnictwo etykiet podlega ogólnym warunkom nazewnictwa w programowaniu, dlatego tymczasowo przyjmijmy obowiązywanie identycznych reguł jak dla nazw zmiennych.
- Etykieta może być umieszczona w niemal dowolnym miejscu bloku kodu źródłowego.
- Rozkazem skoku do etykiety jest słowo **`goto`** z następującą po nim nazwą etykiety.
- Etykiety nie mogą się powtarzać, natomiast jest możliwe stosowanie wiele skoków do jednej etykiety.

10.5.2 Dodatki

- Stosowanie instrukcji skoku bezwarunkowego radykalnie zaburza strukturalność kodu źródłowego. Więcej na przedmiocie *Wstęp do informatyki* oraz w dalszej części wykładu.
- **Z powyższego powodu używanie instrukcji skoku bezwarunkowego jest bezdyskusyjnie uznawane za objaw skrajnego braku kompetencji programistycznych.**
- **Informatyka nie zna powodów, dla których używanie instrukcji skoku bezwarunkowego byłoby konieczne.**
- Mimo powyższego braku, jedną z kilku sytuacji uzasadniających użycie instrukcji skoku bezwarunkowego jest sytuacja głęboko zagnieżdżonych pętli ze skokiem z wnętrza pętli najbardziej wewnętrznej na zewnątrz pętli najbardziej zewnętrznej. Jednakże nawet w opisanej sytuacji za lepsze uznawane jest rozwiązanie niestosujące instrukcji skoku bezwarunkowego.

Kod źródłowy dla demonstracji instrukcji skoku bezwarunkowego.

```
1  // Pierwsza trójka pitagorejska
2  #include <iostream>
3  using namespace std;
4
5  int main () {
6      for ( int i = 100; i <= 999; i = i + 1 ) {
7          for ( int j = 100; j <= 999; j = j + 1 ) {
8              for ( int k = 100; k <= 999; k = k + 1 )
9                  if ( i * i + j * j == k * k ) {
10                     cout << i << " " << j << " " << k << endl;
11                     goto found;
12                     }
13             }
14         }
15     found:
16
17     return 0;
18     }
```

10.6 Instrukcje sterowania pętlą

10.6.1 Wprowadzenie

- Język C wprowadził do programowania specjalne instrukcje sterowania pętlą, to jest **`continue`** oraz **`break`**.
- Omawianie instrukcje mogą być stosowane wyłącznie w bloku instrukcji pętli (zarówno *while*, *do-while* jak i pętli *for*).
- Instrukcja **`continue`** powoduje przerwanie aktualnego sterowania pętli i powrót do sprawdzenia warunku kontynuacji pętli.
- Instrukcja **`break`** powoduje przerwanie działania pętli i przekazanie sterowania do pierwszej instrukcji poza pętlą.
- Omawiane rozkazy są szczególną wersją instrukcji skoku bezwarunkowego, ale z określonym i ustalonym miejscem skoku.
- W związku z powyższym odium konstrukcji **`goto`** dotyczy omawianych instrukcji w mniejszym stopniu.
- Stosowanie poleceń **`continue`** oraz **`break`** zaburza strukturę kodu źródłowego, zatem nie powinno być nadużywane.
- W algorytmice (ogólniejsze od programowania) omawiane rozkazy są wykluczone z możliwości stosowania.

10.6.2 Instrukcja continue

… powoduje skok sterowania z miejsca wystąpienia do miejsca testowania warunku końca pętli.

```
1  // Zgadywanie ilorazowe
2  #include <iostream>
3  using namespace std;
4
5  int main () {
6      int i;
7
8      do {
9          cin >> i;
10
11         if ( i == 0 ) {
12             cout << "bez zera!!!" << endl;
13             // i == 87
14             continue;
15             }
16
17         cout << "x / " << i << " = " << 87.0 / i << endl;
18         }
19     while
20         ( i != 87 );
21
22     cout << "trafione" << endl;
23
24     return 0;
25     }
```

- Idea działania instrukcji **continue** dla pętli *do-while*.

```
do
    przed
    continue;
    po
while
( warunek );
```

```
    do
        przed
        goto label;
        po
label:
    while
    ( warunek );
```

- Idea działania instrukcji **continue** dla pętli *while-do*.

```
while ( warunek ) {
    przed
    continue;
    po
    }
```

```
 while ( warunek ) {
        przed
        goto label;
        po
label:
        }
```

- Idea działania instrukcji **continue** dla pętli **for** pozostaje zadaniem na zajęcia laboratoryjne.

10.6.3 Instrukcja break

Idea działania: w ciele pętli powoduje skok sterowania z miejsca użycia do pierwszej instrukcji za pętlą.

```
1  // Zgadywanie do 15 razy
2  #include <iostream>
3  using namespace std;
4
5  int main () {
6      int x = 87;
7      int c;
8
9      for ( int i = 1; i <= 15; i = i + 1 ) {
10         cout << "Proba " << i << "/15 : ";
11
12         cin >> c;
13
14         if ( c == x ) {
15             cout << "trafione !!!" << endl;
16             break;
17             }
18
19         cout << "pomylka" << endl;
20         }
21
22     return 0;
23     }
```

- Idea działania instrukcji **break** dla pętli *do-while*.

```
do
    przed
    break;
    po
while
    ( warunek );
po pętli
```

```
    do
        przed
        goto label;
        po
    while
        ( warunek );
label:
    po pętli
```

- Działanie instrukcji **break** dla pętli *for*.
  Wersja z instrukcją:

```
for ( inicjalizacja; koniec; inkrementacja ) {
    przed
    break;
    po
    }
po pętli
```

  Równoważna wersja bez instrukcji:

```
for ( inicjalizacja; koniec; inkrementacja ) {
    przed
    goto label;
    po
    }
label:
po pętli
```

10.7 Instrukcja `return`

- Szczególną i ważną instrukcją sterującą jest używana dotąd bez wyjaśnienia instrukcja **`return`**.
- Ogólne działanie polega na porzuceniu aktualnie wykonywanego fragmentu kodu i przejście do innego fragmentu.
- Z racji ścisłego związania z zagadnieniem podprogramów, instrukcja **`return`** zostanie omówiona w dalszej części wykładu.

10.8 Instrukcja wielokrotnego podstawienia

W języku C/C++ możliwe jest łączenie wielu podstawień w pojedynczej instrukcji. Przykładowa instrukcja postaci:

$$\texttt{zmienna}_1 = \texttt{zmienna}_2 = \texttt{zmienna}_3 = \textit{wyrażenie};$$

jest równoważna sekwencji pojedynczych podstawień:

$$\begin{aligned}\texttt{zmienna}_3 &= \textit{wyrażenie};\\ \texttt{zmienna}_2 &= \texttt{zmienna}_3;\\ \texttt{zmienna}_1 &= \texttt{zmienna}_2;\end{aligned}$$

Kod źródłowy dla demonstracji wielokrotnego podstawiania.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i = 1;
6      int j = 2;
7      int k = 3;
8
9      cout << i << j << k << endl;
10     k = j = i = 4;
11     cout << i << j << k << endl;
12
13     return 0;
14     }
```

10.9 Instrukcje wejścia/wyjścia

- W przeciwieństwie do wielu innych języków programowania, język C/C++ nie przewiduje specjalnych instrukcji wejścia/wyjścia.
- Działania wejściowo/wyjściowe w języku programowania C/C++ realizowane są poprzez specjalne podprogramy i zostaną omówione osobno.

11 Szczególność tablic znakowych

11.1 Wypisywanie i wczytywanie

- W przeciwieństwie do tablic innych typów, tablice znaków mogą być wypisywane całą swoją zawartością wyłącznie poprzez podanie nazwy tablicy.
- Tablice znaków przy wypisywaniu są wyświetlane w formie ciągu znakowego.
- Ogranicznikiem wypisywania jest bajt zerowy, kończący wypisywanie.
- Przy wypisywaniu tablicy, zadana długość tablicy nie ma znaczenia i mogą być wypisywane znaki spoza wypisywanej tablicy.
- W przeciwieństwie do tablic innych typów, tablice znaków mogą być wczytywane jedną instrukcją z podaniem całej nazwy tablicy.
- Przy wczytywaniu, długość tablicy danych nie jest kontrolowana, zatem przy ilości wczytanych danych przekraczających długość tablicy możliwe są różnorodne błędy.
- Wczytywanie samoczynnie dokłada bajt zerowy na koniec wczytywanych danych.
- Wobec powyższego, dla uniknięcia problemów, tablica znaków przeznaczona do wczytania powinna mieć długość o jeden większą niż ilość widocznych znaków dla pamiętania ostatniego zerowego bajtu.

Kod źródłowy dla demonstracji wypisywania i wczytywania tablic znaków.

```cpp
#include <iostream>
using namespace std;

int main () {
    char arrayOne [ 8 ]   = { '0', '1', '2', '3', '4', '5', '6', '7' };
    char arrayTwo [ 16 ]  = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p' };

    cout << arrayOne << endl;
    cout << arrayTwo << endl << endl;

    arrayOne [ 7 ] = 0;
    arrayTwo [ 7 ] = 0;

    cout << arrayOne << endl;
    cout << arrayTwo << endl << endl;

    cin >> arrayOne; // "0123"
    cout << arrayOne << endl;
    cout << " [ 3 ] >" << arrayOne [ 3 ] << "< " << endl;
    cout << " [ 4 ] >" << arrayOne [ 4 ] << "< " << endl;
    cout << " [ 4 ] >" << (int)arrayOne [ 4 ] << "< " << endl << endl;

    char arrayThree [ 16 ]  = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 0 };
    char arrayFour [ 8 ]   = { '1', '2', '3', '4', '5', '6', '7', 0 };

    cout << "arrayThree: " << arrayThree << endl;
    cout << "arrayFour: " << arrayFour << endl << endl;

    cin >> arrayFour; // "abc_abc_abc_abc_abc_"

    cout << endl << "arrayThree: " << arrayThree << endl;
    cout << "arrayFour: " << arrayFour << endl << endl;

    cout << "arrayThree [ 4 ]: " << (int)arrayThree [ 4 ] << endl;
    cout << "arrayFour [ 20 ]: " << (int)arrayFour [ 20 ] << endl;

    return 0;
    }
```

11.2 Inicjalizacja napisem

- W przeciwieństwie do tablic innych typów, tablice typu znakowego mogą być inicjowane nie tylko element po elemencie w klamrowych nawiasach, ale w szczególny sposób poprzez podstawienie napisu w podwójnych apostrofach.
- Inicjalizacja tablicy znaków napisem powoduje dodanie na końcu niewidocznego bajta zerowego.
- Deklarowana długość tablicy znaków inicjowanej napisem musi być większa o nie mniej niż jeden dla pamiętania bajta zerowego z konsekwencją rozlicznych błędów w przypadku nieprzestrzegania.
- Z podobnych powodów, tablica znaków bez deklaracji długości inicjowana napisem ma ilość elementów większą o jeden niż ilość widocznych znaków.
- Pomijając powyższe szczególności, tablice znaków są zwykłymi tablicami.
- Więcej o tablicach znakowych w dalszej części wykładu w ramach osobnych rozdziałów dotyczących napisów.

Kod źródłowy dla demonstracji inicjalizacji tablic znakowych napisami.

```cpp
#include <iostream>
using namespace std;

int main () {
    char arrayOne [ 20 ] = "0123456";

    cout << ">" << arrayOne [ 6 ] << "<" << endl;       // 6
    cout << ">" << arrayOne [ 7 ] << "<" << endl;       //
    cout << ">" << (int)arrayOne [ 7 ] << "<" << endl;  // 0, (int) wyprzedzenie

    cout << endl << sizeof ( arrayOne ) << endl;  // 20, sizeof wyprzedzenie

    char arrayTwo [] = "0123456789";
    cout << sizeof ( arrayTwo ) << endl;  // 11

    cout << endl << arrayOne << " " << arrayTwo << endl; // 0123456 0123456789

    //char arrayThree [ 5 ] = "12345"; // blednie
    char arrayThree [ 5 ] = "1234"; // poprawnie

    char arrayFour [ 10 ]; // "abc"

    cin >> arrayFour;
    cout << arrayFour << endl;

    cout << endl << "arrayThree: " << arrayThree << endl;
    cin >> arrayFour; // "123456789_12345_XY"
    cout << "arrayThree: " << arrayThree << endl;

    return 0;
    }
```

12 Podprogramy

12.1 Generalia

- **Podprogram** jest wyodrębnionym fragmentem kodu źródłowego z unikalną nazwą i możliwością użycia wywołania nazwy w zastępstwie działania wyodrębnionego fragmentu.
- Ogólne programowanie wyróżnia dwa zasadnicze rodzaje podprogramów, to jest **funkcje** oraz **procedury** - omówione dalej osobno.
- Programowanie przewiduje zależność działania podprogramów od wartości podanych przy wywołaniu, zwanych **argumentami podprogramów** lub **parametrami podprogramów**.
- Argumenty pozwalają na odmienne działania podprogramów mimo zawsze jednakowego kodu źródłowego.

12.2 Kategorie podprogramów

12.2.1 Funkcja

- **Funkcja** jest podprogramem realizującym działanie, ale po zakończeniu działania poprzez nazwę reprezentującym wartość.
- Funkcja w rozumieniu programistycznym odpowiada funkcji w najogólniejszym rozumieniu matematycznym.
- Nazwa funkcji:
  - Może być składową wyrażeń.
  - Nie może występować w formie samoistnej instrukcji.

12.2.2 Procedura

- **Procedura** jest podprogramem wyłącznie realizującym działanie i niepozostawiającym żadnej wartości po zakończeniu działania.
- Nazwa procedury:
  - Nie może być składową wyrażenia.
  - Może być użyta wyłącznie w formie samodzielnej instrukcji.

12.3 Podprogramy w języku C/C++

- W języku C/C++ brak specjalnych słów na odróżnienie procedur i funkcji (w przeciwieństwie do wielu innych języków programowania), jednakże …
- …domyślnie (bez specjalnego zaznaczenia) każdy podprogram jest funkcją, a zarazem …
- … każdy podprogram (nawet ścisłe funkcje) ZAWSZE może być wywołany w formie procedury, czyli samodzielnej instrukcji i wtedy …
- …wartość zwracana przez funkcję wywołaną w formie procedury jest ignorowana.
- Ścisłe procedury (bez możliwości użycia zwracanej wartości) wymagają specjalnego zaznaczenia.
- Więcej w dalszej części wykładu.

Kod źródłowy dla wstępnej demonstracji podprogramów

```cpp
#include <iostream>
using namespace std;

// podprogram, w tym wypadku funkcja
int Delta ( int a, int b, int c ) {// deklaracja funkcji - typ, nazwa, argumenty
    int d = b * b - 4 * a * c; // kod źródłowy podprogramu

    return d; // instrukcja zastępująca użycie wartością
    }

int main () {
    Delta ( 1, -2, 1 ); // wywołanie jak procedura
    int k = 5 * Delta ( 1, -5, 6 ) - 10; // -5, przykład użycia
    cout << k << endl;

    cout << Delta ( k, -5 * k, 6 * k ) << endl; // 25, przykład użycia
    cout << Delta ( k, -3 * k, Delta ( -1, -2, 2 ) ) << endl; // 465, przykład użycia

    for ( int i = 1; i <= 5; i = i + 1 ) {
        int p;
        int q;
        int r;

        cin >> p >> q >> r;
        cout << "Delta ( " << p << ", " << q << ", " << r << " ) = ";
        cout << Delta ( p, q, r ) << endl; // przykład użycia
        }

    return 0;
    }
```

12.4 Funkcje

12.4.1 Postać ogólna

```
typ nazwa ( argumenty ) {
    kod
    }
```

gdzie:

- *typ* - jest nazwą typu wartości pozostałej po działaniu.
- *nazwa* - jest unikalną nazwą podprogramu.
- *argumenty* - z racji obszerności zagadnienia są omówione osobno.
- *kod* - rozkazy określające działanie podprogramu.

Uwaga: Brak określenia typu oznacza domyślnie typ **`int`**.

12.4.2 Zwracanie wartości

- W kodzie źródłowym rozkazów funkcji konieczne jest umieszczenie przynajmniej jednej instrukcji postaci:

  return *wyrażenie*

- Wartość *wyrażenie* po instrukcji **return** określa wartość pamiętaną przez nazwę podprogramu po zakończeniu działania i musi być typu określonego w deklaracji funkcji.
- Niektóre kompilatory pozwalają na brak instrukcji **return**, ale wtedy zwracana wartość jest przypadkowa.
- Użycie instrukcji **return** oznacza wyjście z funkcji, stanowiąc zakamuflowane użycie instrukcji **goto**.
- Eleganckim sposobem zapobiegającym wielu błędom jest nie tylko użycie jednej instrukcji **return**, ale jeszcze umieszczonej na końcu kodu funkcji.
- Umieszczanie w kodzie funkcji wielu instrukcji **return** (z racji odpowiadania szczególnemu użyciu instrukcji **goto**) fatalnie świadczy o programiście.

Kod źródłowy dla demonstracji zwracania wartości funkcji.

```
#include <iostream>
using namespace std;

int WasThree ( int a, int b ) {
    cout << "a = " << a << " b = " << b << ", ";

    if ( a == 3 ) {
        cout << "pierwszy trzy" << endl;
        return 1;
        }

    if ( b == 3 ) {
        cout << "drugi trzy" << endl;
        return 2;
        }

    cout << "zaden trzy" << endl;
    }

int main () {
    cout << "zwrocone: " << WasThree ( 3, 3 ) << endl << endl;
    cout << "zwrocone: " << WasThree ( 3, 1 ) << endl << endl;
    cout << "zwrocone: " << WasThree ( 1, 3 ) << endl << endl;
    cout << "zwrocone: " << WasThree ( 1, 1 ) << endl << endl;

    return 0;
    }
```

12.5 Procedury

- Brak typu przy określaniu funkcji nie oznacza braku zwracanej wartości, ale domyślnie określa typ **`int`**.
- Dla zadeklarowania podprogramu w postaci procedury konieczne jest określenie typu specjalnym słowem **`void`** oznaczającym brak typu.
- Zawarcie w kodzie procedury słowa **`return`** skutkuje błędem kompilacji.
- Użycie procedury w wyrażeniu skutkuje błędem kompilacji.

Kod źródłowy dla demonstracji podprogramów w formie procedur

```
1  #include <iostream>
2  using namespace std;
3
4  void Delta ( int a, int b, int c ) {
5      cout << "a = " << a << endl;
6      cout << "b = " << b << endl;
7      cout << "c = " << c << endl;
8
9      int d = b * b - 4 * a * c;
10     cout << endl << "delta = " << d << endl;
11     //return d; // błąd
12     }
13
14 int main () {
15     int i = 2 * Delta ( 1, 2, 3 ); // błąd
16     Delta ( 1, 2, 3 );
17
18     return 0;
19     }
```

12.6 Umiejscowienie

- Zasadniczo kody podprogramów powinny być umieszczane zanim wystąpi ich pierwsze użycie.
- W przeciwieństwie do wielu innych języków programowania kody podprogramów nie mogą być umieszczone w głównym programie. Tym samym …
- … kody podprogramów muszą być umieszczone poza głównym programem.
- Ilość podprogramów może być dowolnie duża, ale konieczne jest utrzymanie zależności deklaracji przed użyciem.

12.7 Argumenty podprogramów

12.7.1 Generalia

- Deklaracja argumentów jest ciągiem napisów analogicznych do deklaracji zmiennych, ale oddzielonych przecinkami.
- W ciele podprogramu identyfikatory deklarowanych argumentów traktowane są jak zmienne inicjowane wartościami podanymi przy wywołaniu podprogramu.
- W wywołaniu podprogramu ilość argumentów musi być zgodna z deklaracją.
- Argumenty podprogramów mogą być puste.
- Istnieje wiele ważnych (w początkowej nauce trudnych) zagadnień związanych zarówno z deklaracją argumentów jak i z ich użyciem. Dlatego zostaną omówione osobno w dalszej części materiału.

12.7.2 Przekazywanie argumentów

- Ogólne programowanie rozróżnia dwie różne obsługi przekazywania argumentów, nazywane **przekazywaniem przez wartość** oraz **przekazywaniem przez zmienną**.
- Przekazywanie argumentów **przez wartość**
  - Zostało wstępnie użyte wcześniej.
  - Pozwala tylko na operowanie wartościami przekazanymi w wywołaniu podprogramu.
  - Nie daje możliwości modyfikacji wartości zmiennych użytych w argumentach.
- Przekazywanie argumentów **przez zmienną** umożliwia modyfikacji zewnętrznych zmiennych podanych w argumentach.
- W języku C/C++ przekazywanie argumentów przez wartość nazywane jest jak w ogólnym programowaniu.
- W języku C istnieje jeden mechanizm przekazywania argumentów przez zmienną, zaś język C++ wprowadził dodatkową możliwość.
- Dla odróżnienia dwóch sposobów przekazywania argumentów przez zmienną, przekazywanie wprowadzone w języku C nazywane jest przekazywaniem argumentów **przez wskaźnik**, zaś przekazywanie argumentów wprowadzone w języku C++ nosi nazwę przekazywania argumentów **przez referencję**.
- Przekazywania tablic w postaci argumentów tymczasowo potraktujemy jak osobne zagadnienie, zaś z czasem okaże się szczególnym przypadkiem wcześniej poznanych mechanizmów.
- Zagadnienie rozróżniania różnych sposobów przekazywania parametrów bywa trudne na początkowym etapie nauki, ale z czasem staje się naturalne.

12.7.3 Przekazywanie przez wartość

- Wewnątrz podprogramów argumenty są zmiennymi o wartościach inicjowanych wyrażeniami użytymi w wywołaniu.
- Zmienne z argumentów są tworzone tylko na czas istnienia podprogramu i po zakończeniu podprogramu przestają istnieć.
- Nawet użycie w postaci argumentu zewnętrznej zmiennej o identycznej nazwie jak nazwa argumentu (nie mówiąc o argumentach będących wyrażeniami) nie zmienia zawartości użytej zewnętrznej zmiennej.

Kod źródłowy dla demonstracji argumentów podprogramów przekazywanych przez wartość.

```
1  #include <iostream>
2  using namespace std;
3
4  void Subprogram ( int a )
5      cout << endl << "wewnatrz przed a = " << a << endl; // 2
6      a = 20 * a + 300;
7      cout << "wewnatrz po a = " << a << endl; // 340
8      }
9
10 int main () {
11     int a = 2;
12
13     cout << "zewnatrz przed a = " << a << endl; // 2
14     Subprogram ( a );
15     cout << endl << "zewnatrz po a = " << a << endl; // 2
16
17     return 0;
18     }
```

12.7.4 Przekazywanie przez wskaźnik

12.7.4.1. Zapis i stosowanie

- Tworzenie podprogramów modyfikujących zmienne użyte w argumentach jest fundamentalną cechą ogólnego programowania, jednakże …
- … przekazywanie zmiennych w argumentach przez wartość (z racji działania na zmiennych wewnętrznych, przestających istnieć wraz z końcem działania podprogramu) nie pozwala modyfikować użytych zmiennych.
- Rozwiązaniem potrzeby działania wewnątrz podprogramu na zmiennych zewnętrznych użytych w argumentach jest przekazywanie **argumentów przez wskaźnik**.
- Przekazywanie argumentów przez wskaźnik nie jest nadzwyczajnym mechanizmem, lecz przykładem użycia szerszego, niezbędnego i naturalnego elementu ogólnego programowania stanowionego przez typ wskaźnikowy omówiony w dalszej części wykładu.
- Przekazywanie argumentów przez wskaźnik wymaga uwzględnienia trzech warunków:
  - Deklaracja argumentu przekazywanego przez wskaźnik wymaga pomiędzy identyfikatorem typu a identyfikatorem argumentu znaku gwiazdki *.
  - Użycie identyfikatora argumentu w kodzie źródłowym podprogramu rozumiane w postaci posługiwanie się przekazywaną zmienną musi być poprzedzone znakiem *.
  - Każde użycie identyfikatora przekazywanej zmiennej w argumencie musi być poprzedzone znakiem &.

Kod źródłowy dla demonstracji argumentów podprogramów przekazywanych przez wskaźnik

```
#include <iostream>
using namespace std;

void SubProgram ( int* a ) {
    cout << endl << "wewnatrz przed a = " << *a << endl;
    *a = *a * 20 + 300;
    cout << "wewnatrz po a = " << *a << endl;
    }

int main () {
    int a = 2;
    int b = 333;

    SubProgram ( 2 ); // błąd
    SubProgram ( a + 1 ); // błąd
    SubProgram ( a + b ); // błąd
    SubProgram ( a ); // błąd

    cout << "zewnatrz przed a = " << a << endl; // 2
    SubProgram ( &a );
    cout << endl << "zewnatrz po a = " << a << endl; // 340

    cout << endl << "zewnatrz przed b = " << b << endl; // 333
    SubProgram ( &b );
    cout << endl << "zewnatrz po b = " << b << endl; // 6960

    return 0;
    }
```

12.7.4.2. Dodatki

- Znak & użyty w kodzie źródłowym zewnętrznym w stosunku do podprogramu dla oznaczenia argumentów przekazywanych przez wskaźnik jest użyciem tak zwanego **operatora adresu**, stanowiącego element fundamentalnego zagadnienia programistycznego opisanego w dalszej części wykładu.
- Znak * użyty w treści podprogramu dla identyfikatorów argumentów przekazywanych przez wskaźnik jest użycia tak zwanego **operatora wyłuskania**, stanowiącego element wspomnianego wyżej fundamentalnego zagadnienia programistycznego opisanego w dalszej części wykładu.
- Znak * użyty w deklaracji argumentów podprogramów dla argumentów przekazywanych przez wskaźnik jest podobnie użyty jak operator wyłuskania, ale nim nie jest.
- Podobieństwa użycia znaku * (dodatkowo wzmocnione jednakowością operatora mnożenia) bywa przyczyną pomyłek na początkowym etapie nauki. Jednakże trudności uniknąć się nie da, zaś zjawisko jednego symbolu dla różnych działań nie jest wyjątkowym.

12.7.5 Przekazywanie przez referencję

- Przekazywanie argumentów przez wskaźnik z racji wymogu pamiętania kilku warunków bywa kłopotliwe dla początkujących programistów.
- Dla ułatwienia, w języku C++ wprowadzono dodatkowe przekazywanie argumentów przez zmienną, nazywane przekazywaniem przez referencję.
- Dla implementacji przekazania argumentu przez referencję wystarczy w deklaracji argumentu poprzedzić nazwę argumentu znakiem &.
- Argumentu przekazanego przez referencję w treści podprogramu używa się jak zwykłych zmiennych, czyli analogicznie do przekazywania parametrów przez wartość i bez konieczności poprzedzania znakiem *.
- Przy wywołaniu podprogramu argument przekazywany przez referencję przekazywany jest w postaci identyfikatora zmiennej bez żadnych dodatkowych znaków.
- **Mimo opisanego podobieństwa do przekazywania argumentów przez wartość używane w kodzie źródłowym modyfikacje zmiennych danych argumentami dotyczą przekazanej zmiennej zewnętrznej.**

Kod źródłowy dla demonstracji argumentów podprogramów przekazywanych przez referencję

```cpp
#include <iostream>
using namespace std;

void SubProgram ( int& a ) {
    //cout << *a; // błąd
    cout << endl << "wewnatrz przed a = " << a << endl;
    a = a * 20 + 300;
    cout << "wewnatrz po a = " << a << endl;
    }

int main () {
    int a = 2;
    int b = 333;

    SubProgram ( 2 ); // błąd
    SubProgram ( a + 1 ); // błąd
    SubProgram ( a + b ); // błąd
    SubProgram ( &a ); // błąd
    SubProgram ( *a ); // błąd

    cout << "zewnatrz przed a = " << a << endl; // 2
    SubProgram ( a );
    cout << endl << "zewnatrz po a = " << a << endl; // 340

    cout << endl << "zewnatrz przed b = " << b << endl; // 333
    SubProgram ( b );
    cout << endl << "zewnatrz po b = " << b << endl; // 6960

    return 0;
    }
```

12.7.6 Wskaźniki czy referencje - wady i zalety

Wskaźniki

- Wady:
  Komplikacja stosowania wynikająca z konieczności uwzględniania trzech warunków.
- Zalety:
  - Szczególny przykład działania ogólnego mechanizmu, który i tak trzeba poznać.
  - W momencie wywołania wiadomy jest zastosowany rodzaj przekazywania argumentów, a tym samym …
  - … mniejsza szansa na błąd.

Referencje

- Zalety:
  Prostota zapisu i stosowania, tylko jeden znak w stosunku do przekazywania argumentów przez wartość.
- Wady:
  - Specjalny zapis stosowany tylko w jednym celu, dodatkowa, niekonieczna nauka.
  - Wywołanie nierozróżniające między przekazywaniem argumentów przez wartość a przekazywaniem przez zmienną, oznaczające …
  - … konieczność dodatkowego pamiętania, implikującą możliwość zapomnienia i w konsekwencji …
  - … większą podatność na pomyłki.

Wniosek:
Mimo, że przekazywanie argumentów przez referencję stworzono dla początkujących programistów mających kłopoty z przekazywaniem argumentów przez wskaźnik, lepszym rozwiązaniem nawet na wstępnym etapie nauki wydaję się stosowanie przekazywania argumentów przez wskaźnik.

12.8 Tablice a argumenty

12.8.1 Pojedyncze elementy

Pojedyncze elementy tablicy będąc zwykłymi zmiennymi podlegają opisanym wcześniej zasadom przekazywania argumentów.

```
#include <iostream>
using namespace std;

void PowerOf4Value ( int a ) {
    a = a * a * a * a;
    cout << a << endl;
    }

void PowerOf4Pointer ( int* a ) {
    *a = *a * *a * *a * *a;
    cout << *a << endl;
    }

void PowerOf4Reference ( int& a ) {
    a = a * a * a * a;
    cout << a << endl;
    }

int main () {
    int array [] = { 0, 1, 2, 3, 4 };

    PowerOf4Value ( array [ 2 ] );       // 16
    cout << array [ 2 ] << endl;         // 2

    PowerOf4Pointer ( &array [ 3 ] );    // 81
    cout << array [ 3 ] << endl;         // 81

    PowerOf4Reference ( array [ 4 ] );   // 256
    cout << array [ 4 ] << endl;         // 256

    return 0;
    }
```

12.8.2 Tablice w argumentach

- Argument w postaci tablicy przy wywołaniu jest wyłącznie nazwą tablicy.
- Tablice przekazane w argumentach są ZAWSZE przekazywane przez zmienną, czyli …
- … podprogramy z argumentem zewnętrznej tablicy działają na elementach tablicy.
- Dla przekazania do podprogramu całej tablicy w deklaracji argumentu po jej identyfikatorze musi następować para prostokątnych nawiasów [ ].
- Zawartość nawiasów nie ma znaczenia i może nawet pozostać pusta.
- Możliwe jest podanie deklaracji argumentu w sposób analogiczny do przekazywania argumentów przez wskaźnik, ale znaczenie jest nieco inne i zostanie wyjaśnione w dalszej części wykładu.
- W przeciwieństwie do wielu innych języków programowania, w języku C/C++ tablice nie mają rozmiaru dostępnego poprzez swój identyfikator, zatem zakres używanej w podprogramie tablicy najczęściej również przekazywany jest argumentem.
- Tablice znaków mogą być traktowane analogicznie z innymi tablicami, ale ze specyfiki tablic znakowych wynika możliwość używania wartości ciągów znakowych w argumentach z niemożnością ich modyfikowania.

Kod źródłowy dla demonstracji przekazywania całych tablic w postaci argumentów

```
#include <iostream>
using namespace std;

// void ArraySqr ( int array, int size ) { // problem
// void ArraySqr ( int* array, int size ) { // poprawnie
// void ArraySqr ( int& array, int size ) { // błąd
// void ArraySqr ( int array [ 0 ], int size ) { // poprawnie, bez znaczenia
// void ArraySqr ( int array [ 123 ], int size ) { // poprawnie, bez znaczenia
// void ArraySqr ( int array [ -1 ], int size ) { // błąd
void ArraySqr ( int array [], int size ) {
    for ( int i = 0; i < size; i = i + 1 )
        array [ i ] = array [ i ] * array [ i ];
    }

int main () {
    int arrayOne [ 5 ] = { 0, 1, 2, 3, 4 };
    int arrayTwo [ 7 ] = { 10, 11, 12, 13, 14, 15, 16 };

    //ArraySqr ( *arrayOne, 5 ); // błąd
    //ArraySqr ( &arrayTwo, 7 ); // błąd
    //ArraySqr ( arrayOne [ 0 ], 5 ); // błąd
    ArraySqr ( arrayOne, 5 );
    for ( int i = 0; i < 5; i = i + 1 )
        cout << i << ": " << arrayOne [ i ] << endl;

    cout << endl;

    ArraySqr ( arrayTwo, 7 );
    for ( int i = 0; i < 7; i = i + 1 )
        cout << i << ": " << arrayTwo [ i ] << endl;

    return 0;
    }
```

Kod źródłowy dla demonstracji przekazywania tablic znakowych w postaci argumentów

```
#include <iostream>
using namespace std;

void DisplayString ( char firstArgString [], char secondArgString [], bool change ) {
    cout << firstArgString << ", " << secondArgString << endl;

    if ( change ) {
        firstArgString [ 0 ] = '1';
        secondArgString [ 0 ] = '2';
        }
    }

int main () {
    char stringOne [] = "pierwszy ciag znakowy";
    char stringTwo [] = { 'd', 'r', 'u', 'g', 'i', ' ', 'c', 'i', 'a', 'g', 0 };

    DisplayString ( stringOne, stringTwo, true );
    cout << stringOne << ", " << stringTwo << endl;

    //DisplayString ( "pierwszy", stringOne, true ); // blad
    DisplayString ( "pierwszy", stringOne, false );

    DisplayString ( stringTwo, "drugi", false );
    DisplayString ( "raz", "dwa", false );

    return 0;
    }
```

12.9 Tymczasowe pominięcia

Poniższe ważne zagadnienia dotyczące podprogramów, to jest:

- Rekurencja.
- Nagłówki podprogramów.
- Szczególny podprogram `main`.
- Przeładowanie identyfikatorów podprogramów.
- Podprogramy w formie argumentów.
- Zmienna liczba argumentów.

…zostaną omówione w specjalnym rozdziale uzupełniającym wiedzę o podprogramach lub poprzez osobne zagadnienia.

13 Identyfikatory

13.1 Pojęcie identyfikatora

- Dotąd użyto kilku szczególnych słów jak `main` (główny program), **`using`** oraz **`namespace`** (obsługa zdefiniowanych bytów), **`int`**, **`float`** i inne (nazwy typów), **`if`**, **`else`**, **`while`**, **`do`** i inne (nazwy instrukcji) oraz nazwy własnych zmiennych lub podprogramów.
- W programowaniu dla nazwy różnorodnych bytów stosowane jest słowo **identyfikator**.
- Popularne słowo *nazwa* nie jest używane, ponieważ:
  - Nazwy bytów programistycznych są zagadnieniem bardziej złożonym.
  - Identyfikatory podlegają wielu regułom różniącym się między językami programowania.
- W języku C/C++ poprawnym identyfikatorem jest napis spełniający warunki:
  - Składa się wyłącznie z liter, cyfr lub znaku podkreślenia _ (*underscore*).
  - Jest ściśle spójny, to znaczy nie zawiera żadnych innych kategorii znaków poza wyżej wymienionymi.
  - Pierwszy znak nie może być cyfrą.
  - Znak _ jest uznawany za literę.

Kod źródłowy dla demonstracji pojęcia identyfikatora.

```
1  int main () {
2      // poprawne identyfikatory
3      int _abc;
4      float CIRCLE_DIAMETER;
5      int Path3;
6      int __stacklen;
7
8      // niepoprawne identyfikatory
9      int 3ident;           // cyfra na początku
10     int unproper&name;    // niepoprawny znak w środku
11     int unproper name;    // niespójność napisów
12
13     return 0;
14     }
```

13.2 Ranga identyfikatorów

- Wszystkie identyfikatory występujące w językach programowania można dzielić na różne kategorie.
- Najpowszechniejszym podziałem jest podział na trzy kategorie:
  - **słowa kluczowe**
  - **słowa predefiniowane**
  - pozostałe (własne) identyfikatory

13.2.1 Słowa kluczowe

- … mają ściśle ustalone znaczenie i nie mogą być używane w innym znaczeniu.
- Są jednym z fundamentalnych rozróżnień języków programowania między sobą.
- Różne języki programowania zawierają od kilkunastu do kilkudziesięciu słów kluczowych, o różnej wadze i zakresie stosowania.
- W zależności od wersji język C zawiera około 30 słów kluczowych, język C++ około 70, zaś na przykład dialekt C++ używany w *Microsoft Visual C++* około 150.

- Dotychczas używane słowa kluczowe:
  - **`using`** - identyfikator określenia używania różnych mechanizmów upraszczających zapis.
  - **`namespace`** - identyfikator określenia użycia przestrzeni nazw - więcej w dalszej części materiału.
  - **`int`** - identyfikator typu całkowitego.
  - **`float`** - identyfikator typu rzeczywistego pojedynczej precyzji.
  - **`double`** - identyfikator typu rzeczywistego podwójnej precyzji.
  - **`char`** - identyfikator typu znakowego.
  - **`bool`** - identyfikator typu logicznego.
  - **`if`**, **`else`** - identyfikatory instrukcji warunkowych.
  - **`do`**, **`while`**, **`for`** - identyfikatory instrukcji pętli.
  - **`goto`**, **`break`**, **`continue`** - identyfikatory skoków sterowania.
  - **`return`** - instrukcja powrotu.
- Wbrew częstym mniemaniom, słowo `main` nie jest słowem kluczowym.
- Wraz z postępem materiału wprowadzane będą kolejne słowa kluczowe.

Kod źródłowy dla demonstracji kluczowości słowa `main`.

```cpp
#include <iostream>
using namespace std;

//double main () {
//int WinMain () {
int main () {
    //double main = 1.23;
    //char main = 'c';
    int main = 45;

    cout << main << endl;
    //cout << WinMain << endl;

    return 0;
    }
```

13.2.2 Słowa predefiniowane

- Słowa predefiniowane są identyfikatorami spełniającymi poniższe warunki:
  - Są dostarczone wraz z kompilatorem.
  - Mają zdefiniowane znaczenie, ale …
  - … zdefiniowane znaczenie można zmienić.
- **Możliwości określenia własnych znaczeń predefiniowanych identyfikatorów lepiej unikać.**
- .Dotychczas używane słowa predefiniowane:
  - `main` - w dotychczasowym znaczeniu nazwa głównego bloku programu, a faktycznie predefiniowana nazwa podprogramu wywoływanego najpierw.
  - `include` - słowo kluczowe, ale nie języka C czy też C++, ale specjalnego języka preprocesora. Więcej we fragmencie wykładu dotyczącym preprocesingu.
  - `iostream` - nazwa jednego z predefiniowanych zestawów dostępnych danych.
  - `std` - nazwa jednej z predefiniowanych przestrzeni nazw.
  - `cout` - nazwa szczególnej zmiennej pozwalającej wysyłać dane na zewnątrz, w tym na konsolę i ekran.
  - `cin` - nazwa szczególnej zmiennej służącej wczytywaniu danych z zewnątrz, w tym z konsoli i klawiatury.
  - `endl` - szczególna wartość powodująca skok kursora ekranu do początku następnej linii.
- Znajomość słów predefiniowanych danego języka potwierdza doświadczenie w danym języku programowania (danym środowisku, danym kompilatorze), ale niewiele więcej.

13.3 Własne identyfikatory

- Zagadnienie odpowiedniego nazywania własnych bytów programistycznych jest na tyle ważne, że podlega naukowym analizom oraz różnorodnym próbom sformalizowania.
- Właściwy dobór identyfikatorów jest poniekąd sztuką, ale istnieją reguły powszechnie uznawane za dobre, zaś pewne zwyczaje są powszechnie uznawane za błędne.
- Zwyczaje stosowane we własnym nazewnictwie są jednym z najlepszych wyróżników stylu programowania i doświadczenia programisty, zgodnie z zasadą: „*Im leniwiej tworzone identyfikatory tym gorzej*”.
- Inaczej mówiąc:
  - Lekceważenie wyboru dobrego identyfikatora dla oszczędzenia czasu objawia negatywne skutki przy każdym użyciu. Natomiast …
  - … jednokrotnie poświęcony większy czas dla określenia dobrego identyfikatora daje korzyść przy każdorazowym użyciu.
- Więcej w dalszej części wykładu, a przede wszystkim na ćwiczeniach.

14 Uzupełnienie operatorów

14.1 Operatory złożone

- Twórca języka C (Dennis M. Ritchie) wymyślił wiele uproszczeń zapisu kodu źródłowego uznawanych obecnie za genialne.
- Wśród wynalazków Dennisa Ritchie są operatory zwane **operatorami złożonymi**, łączące działanie operatorów arytmetycznych z operatorem podstawienia.
- Operatory złożone obejmują operatory inkrementacyjne oraz operatory kompozycyjne.

14.2 Operatory inkrementacyjne

14.2.1 Generalia

- Operatory inkrementacyjne oznaczone są sekwencjami **++** oraz **--**.
- Operator ++ również jest nazywany **operatorem inkrementacyjnym** (analogicznie do nazwy całej grupy).
- Operator -- nazywany jest **operatorem dekrementacyjnym**.
- Oba operatory inkrementacyjne:
  - Są operatorami unarnymi.
  - Operand musi być zmienną typu pozwalającego zmodyfikować wartość o 1.
  - Występują w wersji prefiksowej oraz w wersji postfiksowej, o istotnie odmiennym działaniu.
- Operator inkrementacyjny zwiększa wartość operanda o 1, zaś operator dekrementacyjny zmniejsza wartość operanda o 1.
- Operatory inkrementacyjne stosuje się:
  - Głównie do typów całkowitych.
  - Rzadziej do typu znakowego.
  - Zastosowanie do typów rzeczywistych jest marginalne i może powodować nieoczekiwane rezultaty.

14.2.2 Wersje prefiksowe a wersje postfiksowe

- Operatory inkrementacyjne są elementem instrukcji podstawienia, ale zarazem są wyrażeniem reprezentującym różną wartość operanda w zależności od prefiksowości lub postfiksowości.
- Prefiksowość lub postfiksowość operatorów inkrementacyjnych nie ma znaczenia o ile wyrażenie z ich udziałem występuje samodzielnie.
- **Wyrażenie w wersjach prefiksowych zwraca wartość operanda przed zadziałaniem operatora, zaś wersje postfiksowe zwracają wartość operanda po zadziałaniu operatora.**
- Inaczej mówiąc:
  - W wersjach prefiksowych najpierw wykonywane jest działanie arytmetyczne, a następnie działanie z użyciem zmienionej wartości operanda. Natomiast …
  - … w wersjach postfiksowych najpierw wykonywane jest wyrażenie z użyciem niezmienionej wartości operanda, a dopiero następnie wykonywane jest działanie arytmetyczne.

- Kod źródłowy dla demonstracji różnic prefiksowego i postfiksowego operatora dekrementacji.

```cpp
#include <iostream>
using namespace std;

int main () {
    int i;
    int j;

    i = 3;
    j = --i;
    cout << "i = " << i << ", j = " << j << endl;
    // i == 2, j == 2

    i = 3;
    j = i--;
    cout << "i = " << i << ", j = " << j << endl;
    // i == 2, j == 3

    return 0;
    }
```

- Różnice działania powyższego kodu:

```cpp
9  j = --i;
```

```cpp
i = i - 1;
j = i;
```

```cpp
14  j = i--;
```

```cpp
j = i;
i = i - 1;
```

14.2.3 Priorytety

- **Operatory inkrementacyjne mają wysoki priorytet.**
- W stosowanym wcześniej wartościowaniu operatorów:
  - Prefiksowe operatory inkrementacyjne (postaci na przykład `++i`, czy też `--i`) mają priorytet wynoszący 3 (równy najwyższemu priorytetowi dotąd stosowanemu).
  - Postfiksowe operatory inkrementacyjne (postaci na przykład `i++` lub `i--`) mają priorytet wynoszący 2 (najwyższy z dotąd stosowanych).

14.3 Operatory kompozycyjne

- Innym godnym uznania wynalazkiem Dennisa M. Ritchie są operatory upraszczające jedne z najczęstszych czynności programistycznych, czyli modyfikacje zmiennej z użyciem operacji arytmetycznych.
- Każde podstawienie postaci:

  zmienna = zmienna operator wyrażenie;

  gdzie *operator* jest jednym z dopuszczalnych operatorów arytmetycznych można zastąpić podstawieniem postaci:

  zmienna operator= wyrażenie;

  gdzie sekwencja *operator*= jest osobnym operatorem kompozycyjnym.
- Przykładowe postawienie:

```
i = i * 5;
```

  jest równoważne wyrażeniu:

```
i *= 5;
```

- Operatory kompozycyjne mają niski priorytet i bez względu na priorytet użytego działania arytmetycznego są na poziomie wynoszącym 16.
- Niski priorytet operatorów kompozycyjnych na początkowym etapie nauki bywa źródłem problemów i dlatego minimalna oszczędność w zapisie nie równoważy czasu straconego na poszukiwanie błędów.

Kod źródłowy dla demonstracji operatorów kompozycyjnych.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i = 3;
6      int j = 5;
7
8      i -= j; // i = i - j
9      cout << i << endl;
10
11     i += i; // i = i + i
12     cout << i << endl;
14
15     i *= 2; // i = i * 2
16     cout << i << endl;
16
17     i /= 2 * i - j; // i = i / ( 2 * i - j )
18     cout << i << endl;
19
20     return 0;
21     }
```

14.4 Tabela priorytetów dotychczas użytych operatorów

operator	poziom priorytetu *(niższa wartość oznacza wyższy priorytet)*
`++` *(postfiksowy)* `--` *(postfiksowy)*	2
`++` *(prefiksowy)* `--` *(prefiksowy)* `!` `-` *(unarny)*	3
`*` `/` `%`	5
`+` `-` *(binarny)*	6
`<` `>` `<=` `>=`	8
`==` `!=`	9
`&&`	13
`||`	14
`=` `+=` `-=` `*-` `/=` `%=`	16

14.5 Stosowanie

- Operatory kompozycyjne oraz operatory inkrementacyjne są specyficzne dla języka C/C++ oraz języków inspirowanych językiem C/C++ (java, python).
- Ogólna informatyka nie musi wiedzieć o istnieniu wąsko programistycznych operatorów kompozycyjnych i inkrementacyjnych, zaś algorytmika (nie-przeznaczona wyłącznie dla informatyków) wyklucza ich stosowanie.
- Wielokrotne stosowanie operatorów kompozycyjnych oraz inkrementacyjnych w jednym wyrażeniu prowadzi do niemożności łatwego zrozumieniu sensu wyrażenia i w konsekwencji jest przyczyną błędów.
- Nadużywanie operatorów inkrementacyjnych nie świadczy o zaawansowanych umiejętnościach programistycznych, ale raczej potwierdza fascynację właściwą wyłącznie początkującym programistom, ograniczającym informatykę wyłącznie do programowania.

Kod źródłowy dla demonstracji nadużywania operatorów kompozycyjnych i operatorów inkrementacyjnych.

```cpp
#include <iostream>
using namespace std;

int main () {
    int i = 3;

    i = ++i + i++;
    cout << "0. " i << endl;

    i = 3;
    i = i++ + ++i;
    cout << "1. " i << endl;

    i = 3;
    i = (--i)++;
    cout << "2. " i << endl;

    i = 3;
    i = (++i)--;
    cout << "3. " i << endl;

    i = 3;
    i = ++++++i;
    cout << "4. " i << endl;

    i = 3;
    i = +(+(+(+(+(+i))))); 
    cout << "5. " i << endl;

    i = 3;
    i -= ++--++--++i;
    cout << "6. " i << endl;

    i = 3;
    i -= +-+-+i;
    cout << "7. " i << endl;

    // i = +++i;   // błąd
    // i = i+++;   // błąd
    // i = i++++;  // błąd

    i = 3;
    i =- i;
    cout << "8. " i << endl;

    i = 3;
    i -=- i;
    cout << "9. " i << endl;

    return 0;
    }
```

15 Literały

15.1 Pojęcie literału

- Stałe są potocznie konkretnymi wartościami. Jednakże programowanie odróżnia stałe w potocznym rozumieniu od stałych w rozumieniu programistycznym.
- **Literały to różnorodne stałe wartości podawane w postaci napisów (czyli podane literalnie)**. Przy czym …
- … nie chodzi o wymyślną (nie wiadomo czy potrzebną) nazwę dla czegoś powszechnie znanego, ale o fakt, że w programowaniu konkretne wartości z różnych powodów mogą i muszą być zapisywane na wiele sposobów nieznanych poza programowaniem.

15.2 Literały typu całkowitego

- Literały liczb w systemie dziesiętnym mają powszechnie stosowaną postać.
- Literały liczb w systemie ósemkowym mają postać ciągu cyfr systemu ósemkowego poprzedzony znakiem `0`.
- Literały liczb w systemie szesnastkowym:
  - Mają postać zapisu ciągu cyfr systemu szesnastkowego z użyciem kolejnych początkowych liter alfabetu dla cyfr powyżej 9.
  - Małe i wielkie litery nie są rozróżniane.
  - Zapis cyfr jest poprzedzony sekwencją `0x` lub `0X`.
- Nowsze wersje standardu języka C++ przewidują literały systemu binarnego z ciągiem cyfr `0` lub `1` poprzedzony sekwencją `0b` lub `0B`.

Kod źródłowy dla demonstracji literałów całkowitoliczbowych.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int i = +123;
6
7      cout << i << endl;   // 123
8
9      i = -0123;
10     cout << i << endl;   // -83
11
12     i = 0x123;
14     cout << i << endl;   // 291
15
16     i = 0b10100110;
17     cout << i << endl;   // 291
18
19     return 0;
20     }
```

15.3 Literały typu rzeczywistego

15.3.1 Literały stałopozycyje

- Ogólna informatyka posługuje się dwoma zapisami liczb rzeczywistych, z których tradycyjny nosi nazwę zapisu **stałopozycyjnego**.
- Stałopozycyjne literały typu rzeczywistego zapisywane są w tradycyjny sposób, ale ze znakiem kropki **.** dla oddzielenia części całkowitej od ułamkowej.
- Możliwe są zapisy liczby rzeczywistej wyłącznie z częścią ułamkową, bez cyfr części całkowitej oraz zapisy wyłącznie części całkowitej zakończonej tylko znakiem kropki (bez cyfr części ułamkowej).
- Zapis liczby rzeczywistej zakończony wyłącznie kropką jest powszechnie praktykowanym prostym sposobem wymuszenia typu rzeczywistego mimo całkowitej wartości.

15.3.2 Literały zmiennopozycyjne

- W ogólnej informatyce powszechną w użyciu jest notacja liczb rzeczywistych zwana najczęściej **notacją naukową** (ze szczególną wersją zwaną **notacją inżynierską**), w matematyce zwana **notacją cecha-mantysa**, **notacją wykładniczą** lub **notacją potęgową**, zaś w informatyce **notacją zmiennopozycyjną**.
- Zmiennopozycyjne literały liczb w systemie dziesiętnym mają postać obejmującą kolejno:
  - Cechę zapisaną stałopozycyjne w systemie dziesiętnym.
  - Rozdzielenie cechy od mantysy literą e lub E.
  - Mantysę w systemie dziesiętnym.
- Nowsze standardy języka C++ przewidują literały liczb zmiennopozycyjnych w mieszanym systemie szesnastkowo-dwójkowo-dziesiętnym. Podstawa używanej potęgi wynosi 2, zaś postać literału obejmuje kolejno:
  - Cechę zapisaną stałopozycyjnie w systemie szesnastkowym poprzedzoną sekwencją `0x` lub `0X`.
  - Rozdzielenie cechy od mantysy literą p lub P.
  - Mantysę w systemie dziesiętnym.

Kod źródłowy dla demonstracji literałów liczb rzeczywistych.

```
#include <iostream>
using namespace std;

int main () {
    float f = -123.456;

    cout << f << endl; // 123.567

    f = 123,456; // niepoprawność
    cout << f << endl; // 123

    f = +.456;
    cout << f << endl; //  0.456

    f = 123.;
    cout << f << endl; // 123

    f = -123.4566;
    cout << f << endl; // -123.457

    f = +123.4564;
    cout << f << endl; // 123.456

    f = -1234567;
    cout << f << endl; // -1.23457e+006

    f = -1.23457e+006;
    cout << f << endl; // -1.23457e+006

    f = -0x1.2D68Ap+20;
    cout << f << endl; // -1.23457e+006

    return 0;
    }
```

15.4 Specjalne literały znakowe

15.4.1 Wprowadzenie

Rozważmy kod źródłowy:

```
1  // Problematyczne literały znakowe
2  #include <iostream>
3  using namespace std;
4
5  int main () {
6      char c;
7
8      c = 'abc'; // dziwność
9      cout << c << endl;
10
11     c = '';    // błąd kompilacji lub wykonania
12                // ewentualnie zawieszenie
14     c = ''';   // błąd kompilacji
15     c = '\';   // błąd kompilacji
16
17     return 0;
18     }
```

15.4.2 Pytania i wątpliwości

- *Czy możliwy jest zapis znaku pustego? Na przykład w postaci literału* `''` *(dwa znaki pojedynczego apostrofu bezpośrednio po sobie)?*
- *Jak wypisać znak pojedynczego apostrofu gdyby zaszła ku temu konieczność? Literał* `'''` *(trzy znaki pojedynczego apostrofu bezpośrednio po sobie) sygnalizuje błąd!*
- *Dlaczego nie można wypisać literału* `'\'`*?*
- *Jak wypisać znaki osiągalne z klawiatury, ale niewidoczne? Enter, Tab, Backspace i inne?*
- *Jak wypisać działanie klawiszy strzałek, czyli zmianę położenia kursora na monitorze?*

15.4.3 Znaki specjalne

- Znaki mające znaczenie w obsłudze edytora w rodzaju znaku nowej linii, znaku tabulacji, znaku cofnięcia kursora (i inne bez objawów na ekranie) oraz znaki konwencji zapisu „zwykłych” znaków (jak znak pojedynczego apostrofu ' są nazywane **znakami specjalnymi** i ich literały złożone są z dwóch znaków w pojedynczych apostrofach z pierwszym znakiem lewego ukośnika \ (*backslash*) i wybranym pojedynczym drugim znakiem.
- Najpopularniejsze literały znaków specjalnych:

```
'\''  -  znak pojedynczego apostrofu
'\n'  -  przejście do nowej linii na jej początek
'\f'  -  przejście do nowej linii bezpośrednio poniżej
'\t'  -  znak tabulacji
'\b'  -  cofnięcie kursora
'\r'  -  przesunięcie kursora do lewego brzegu ekranu
'\\'  -  znak lewego ukośnika, konieczny z racji dodatkowego
         znaku w konwencji zapisu znaków specjalnych
```

- Więcej specjalnych literałów znakowych można znaleźć w szczegółowej dokumentacji języka C/C++.

Kod źródłowy dla demonstracji specjalnych literałów znakowych.

```
#include <iostream>
using namespace std;

int main () {
    cout << '\'' << 'a' << 'b' << '\'' << endl;
    cout << '1' << '\t' << '2' << endl;
    cout << "abc" << '\n' << "xyz" << endl;
    cout << "abc" << '\b' << "xyz" << endl;
    cout << "abc" << '\b' << '\b' << '\b' << "xyz" << endl;
    cout << "abc" << '\r' << "xyz" << endl;
    cout << '\\' << "n" << '\\' << "t " << endl;

    return 0;
    }
```

15.4.4 Dodatki

- Specjalnych literałów znakowych możemy używać w napisach zawartych w podwójnych apostrofach.
- Tym samym w zastępstwie identyfikatora `endl` możemy użyć literału `'\n'`, ewentualnie napisu `"\n"` lub zwyczajnie dodać dwuznakową sekwencję `\n` na koniec wyświetlanego napisu.
- Znak podwójnego apostrofu `"` nie może być użyty w napisach z racji bycia znakiem edytora określającym zakres napisu. Dlatego do zestawu powszechnie używanych specjalnych literałów znakowych należy dodać sekwencję:

  `\"`

  stosowaną zarówno w postaci pojedynczego znaku jak i w postaci składowej napisu.
- Dla napisów szczególne znaczenie ma znak specjalny opisany literałem `\0`, oznaczający koniec wyświetlania.
- Istnieją jeszcze inne sposoby zadawania literałów znakowych, wykorzystujące jeszcze więcej znaków w literałach i korzystające z innych systemów liczbowych, więcej w dalszej części wykładu.

Kod źródłowy dla demonstracji dodatkowych znaczeń specjalnych literałów znakowych.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      cout << "pierwsza linia" << "\n" << "druga linia" << '\n';
6      cout << "trzecia linia\n";
7
8      cout << "tresc przed znakiem\0tresc po znaku\n";
9
10     cout << endl;
11     char c = '\80';    cout << (int)c;
12     c = '\101';        cout << (int)c;
13     c = '\x41' + 32;   cout << (int)c << endl;
14
15     return 0;
16     }
17
```