

# Tryb zaliczenia kursu

- Ocenianie na kursie będzie oparte na skali procentowej.
- Wykład zakończy się egzaminem.
- Ostateczna ocena procentowa kursu będzie średnią z oceny z egzaminu i końcowej oceny z ćwiczeń.

# Tryb zaliczenia kursu

- Ostateczny wynik procentowy zostanie przeliczony na ocenę końcową zgodnie z następującą tabelą:

0-49 %	niedostateczny
50-59 %	dostateczny
60-69 %	plus dostateczny
70-77 %	dobry
78-85 %	plus dobry
86-94 %	bardzo dobry
95-100 %	celujący

# Zarys historii rozwoju języków programowania

# Wieża Babel języków programowania <sup>2</sup>

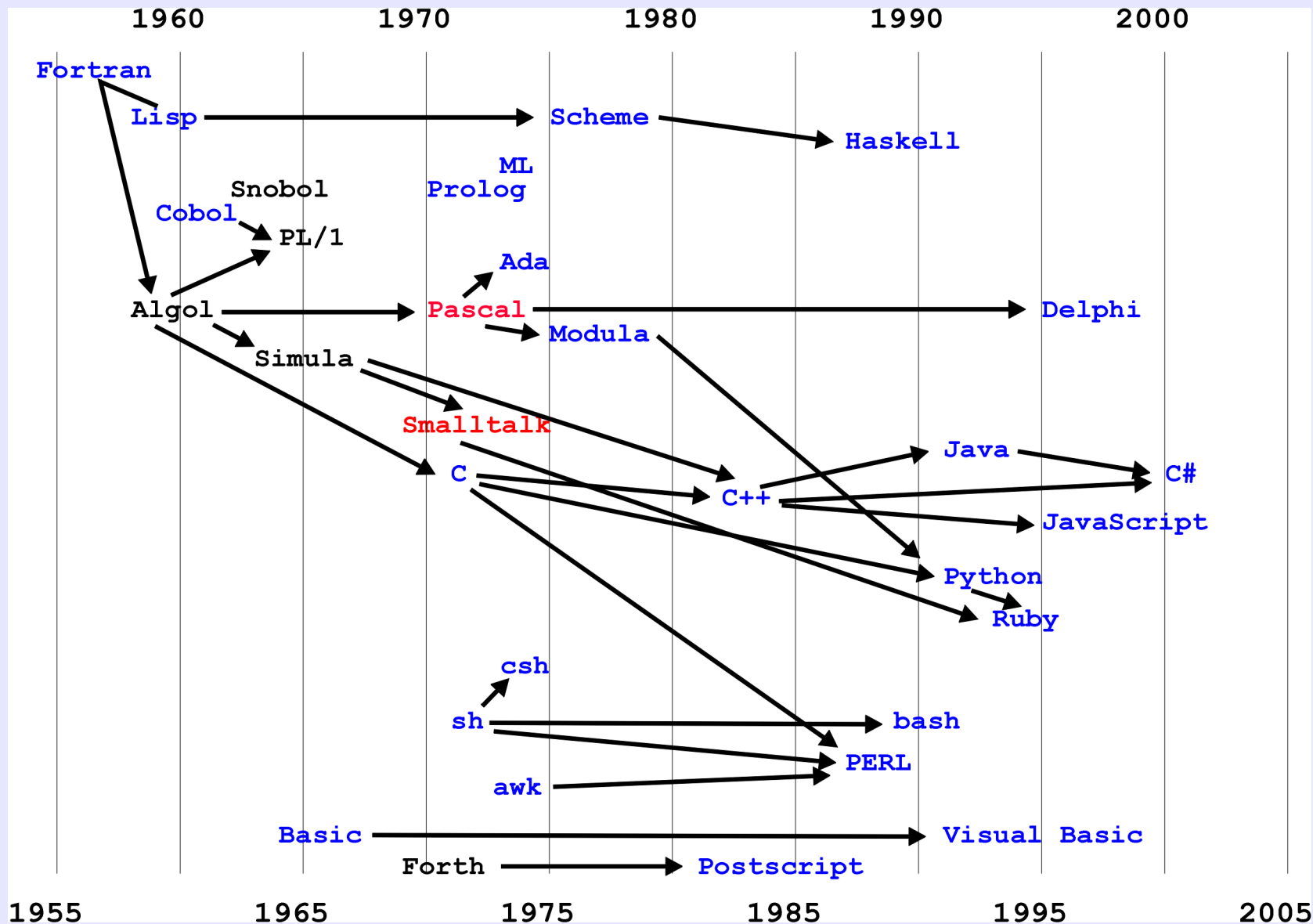


Piotr Bruegel Starszy, *Budowa wieży Babel* , 1553

# Wieża Babel języków programowania 3

- kilka tysięcy języków programowania ■
- kilkadziesiąt aktualnie używanych. ■
- ciągle powstają nowe ■

## 4



# Algorytmy i programy <sup>5</sup>

- **algorytm**: ściśle opisany sposób postępowania w celu rozwiązania pewnego problemu ■
- **program**: algorytm zapisany w sformalizowany sposób, umożliwiający wykonanie algorytmu przy pomocy komputera ■
- program
  - przyjmuje **dane wejściowe**, przetwarza je i produkuje **dane wyjściowe** ■
  - najczęściej ma postać ciągu **instrukcji**, określających sposób przetwarzania danych ■



# Języki programowania <sup>6</sup>

- **język programowania**: formalnie zdefiniowany sposób zapisywania algorytmów, umożliwiający pisanie programów ■
- język programowania określa■
  - jakiego typu dane mogą być przetwarzane i jak te dane są zapisywane ■
  - jakie **instrukcje** mogą być stosowane i jakim podlegają regułom ■





# Techniki programowania <sup>7</sup>

- **technika programowania**: pewien zestaw mniej lub bardziej precyzyjnych reguł, których stosowanie ułatwia osiągnięcie określonych celów w trakcie tworzenia oprogramowania. ■
- konkretną technikę najłatwiej stosować w języku specjalnie dla niej zaprojektowanym ■

# Rozwój technik programowania <sup>8</sup>

Z perspektywy historycznej główne etapy rozwoju programowania to pojawianie się nowych technik programowania

- programowanie linearne ■
- programowanie proceduralne ■
- programowanie funkcyjne (funkcjonalne) ■
- programowanie strukturalne ■
- programowanie w języku logiki (logic programming) ■
- programowanie bazujące na obiektach ■
- programowanie obiektowo orientowane ■
- programowanie orientowane zdarzeniami ■
- programowanie generyczne ■
- programowanie wielowątkowe i rozproszone ■
- programowanie wieloparadygmatowe ■
- metaprogramowanie, w tym programowanie refleksywne i programowanie generatywne ■

# Rozwój technik programowania <sup>9</sup>

- nowa technika  $\rightarrow$  nowe języki ■
- techniki programowania i związane z nią klasy języków: na przykład języki proceduralne, funkcyjne czy obiektowe. ■
- programowania wieloparadygmatowe ■

- zwiększona moc —> zwiększone apetyty ■
- większe programy —> więcej błędów ■
- coraz trudniejsze uruchamianie, testowanie, pielęgnacja ■
- przeciwdziałanie: nowe rozwiązania teoretyczne —> nowe języki programowania ■
- przeżywają te koncepcje i języki, które najlepiej spisują się w praktyce ■

Wszystkie tendencje i trendy w rozwoju języków programowania można streścić w dewizie:



Małe jest piękne!

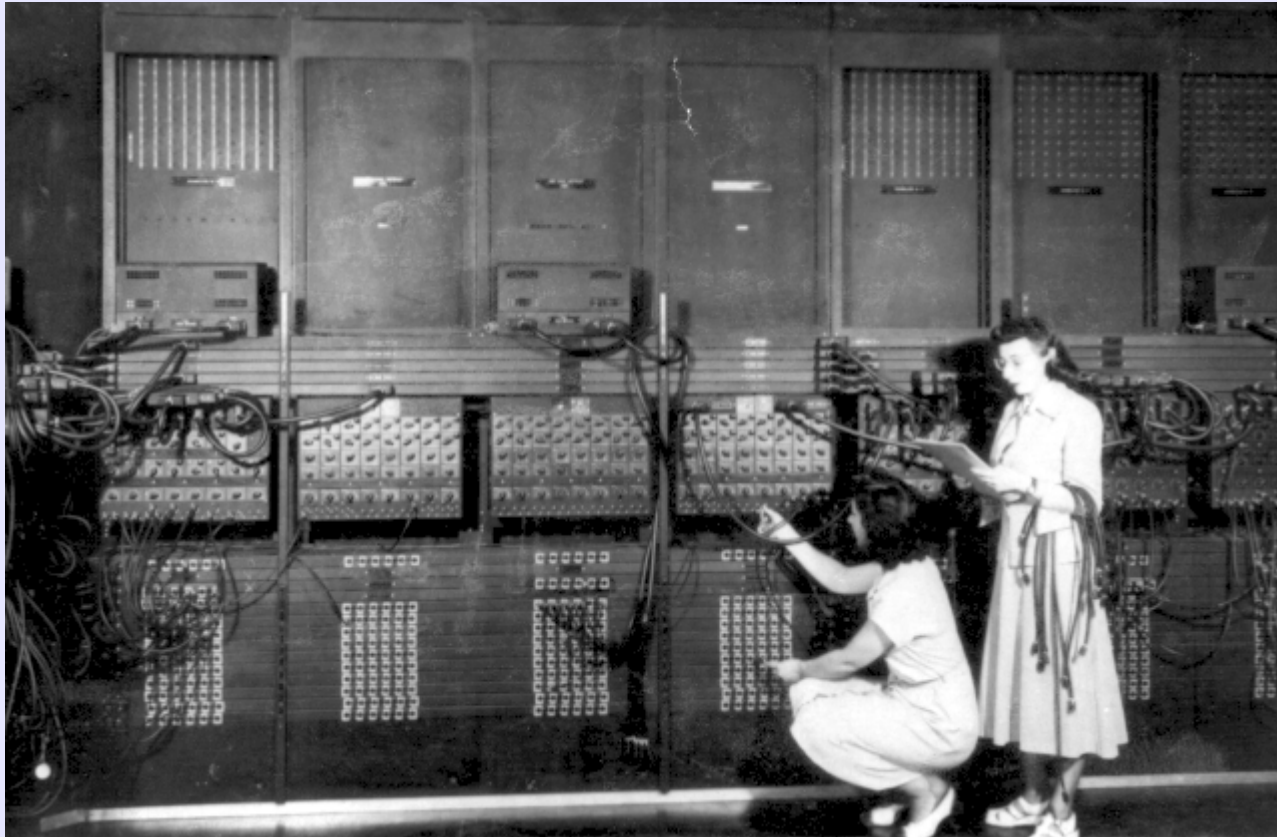
# Kryteria oceny technik i języków programowania <sup>12</sup>

Użyteczność języków programowania oceniana jest przy użyciu różnych kryteriów. Najczęściej stosowane kryteria to: ■

- **Efektywność:** programy w danym języku wykonują się bardzo szybko i zajmują mało pamięci ■
- **Produktywność:** programy w danym języku można szybko napisać i uruchomić ■
- **Odporność na błędy:** struktura języka umożliwia wyłapywanie znacznej ilości błędów na etapie pisania programu, a przed etapem testowania programu ■
- **Stabilność:** język ułatwia tworzenie oprogramowania, w którym stosunkowo rzadko pojawiają się błędy w trakcie wykonywania, a jeśli się pojawiają, są dobrze udokumentowane, więc łatwo ustalić ich przyczynę ■
- **Analizowalność:** o programie relatywnie łatwo jest udowodnić, że robi rzeczywiście to, czego się od niego oczekuje ■
- **Klarowność:** język ułatwia pisanie programów, które w znacznym stopniu same się dokumentują ■
- **Trwałość:** raz przygotowane oprogramowanie może być długo użytkowane w różnych projektach bez konieczności przepisywania go od nowa. ■

Tak na prawdę każdy komputer rozumie tylko jeden język: język wewnętrzny swojego procesora.

- program w języku wewnętrznym: ciąg zer i jedynek kodujących rozkazy procesora oraz dane ■
- programowanie w języku wewnętrznym: bezpośrednie wprowadzanie zer i jedynek do pamięci ■



ENIAC w trakcie programowania w języku wewnętrznym. "U.S. Army Photo"

Pierwsze komputery musiały być programowane w języku wewnętrznym, bo na początku nie było żadnych programów!



Choć w zasadzie każdy język wewnętrzny w pewnym stopniu wspiera programowanie proceduralne, programowanie w języku wewnętrznym ze względu na stopień komplikacji w większości ograniczało się do programowania linearnego.



**Programowanie linearne** to najprostsza technika programowania polegająca na przedstawieniu programu jako ciągu kolejno wykonywanych instrukcji

Typowy asembler umożliwia:

- zapis rozkazów procesora w relatywnie łatwej do zapamiętania przez człowieka postaci symbolicznej, najczęściej w postaci kilkuliterowych słów. ■
- zapis liczb w postaci szesnastkowej i dziesiętnej ■
- symboliczne adresowanie
  - zmiennych. ■
  - miejsc w programie, do których prowadzą instrukcje skoku ■



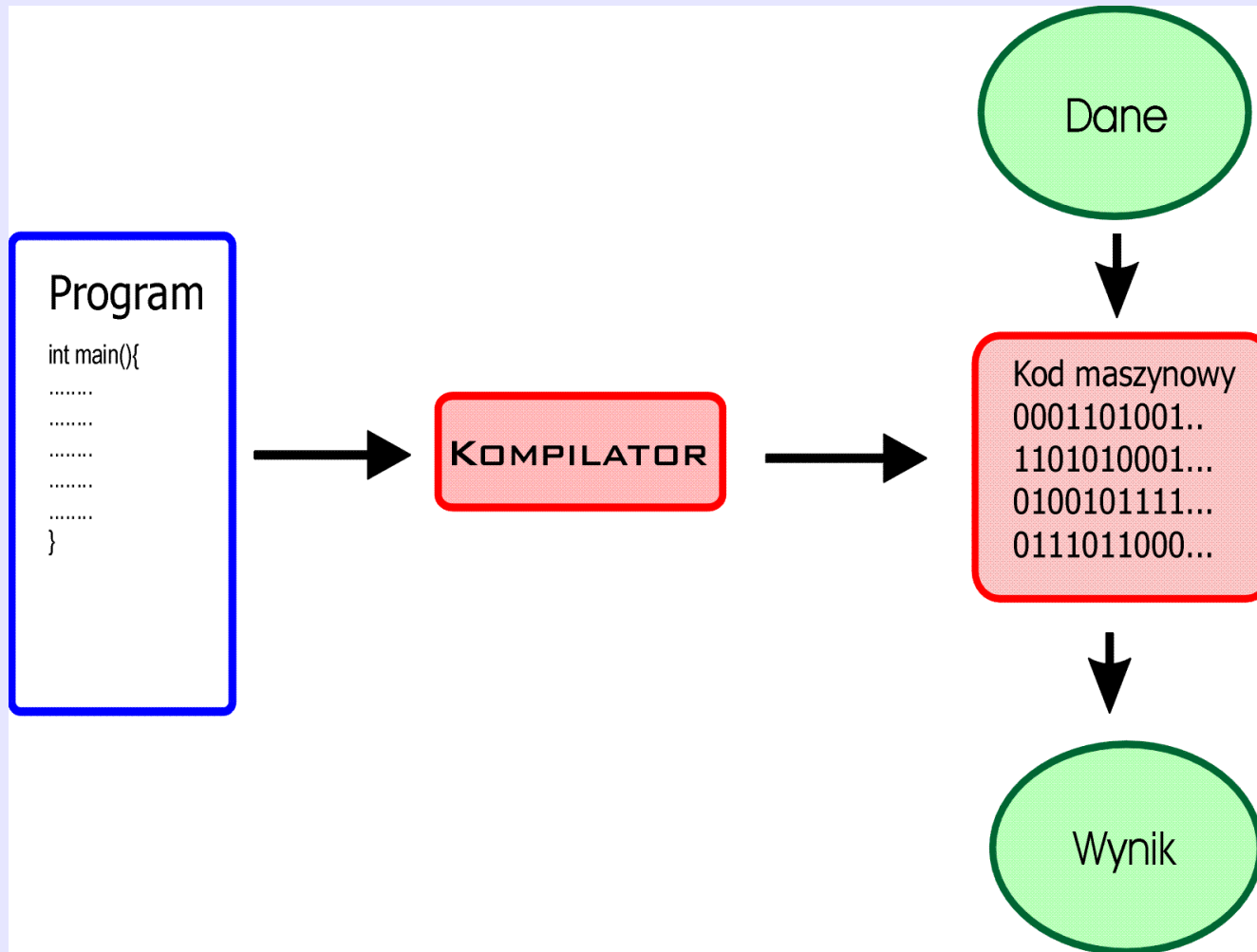
- Teoretycznie assembler umożliwia stosowanie wszystkich technik programowania. ■
- W praktyce jednak jest to najczęściej programowanie linearne i proceduralne. ■
- Z dzisiejszej perspektywy jedyną zaletą programowania w assemblerze jest wysoka efektywność jednak opłacona bardzo niską produktywnością. ■

Assembler jest najprostszym przykładem **programu tworzącego programy** (metaprogramu), a więc programu, który na swoim wejściu przyjmuje ciąg znaków reprezentujących zapisany w pewnej umownej konwencji program, a na wyjściu jego odpowiednik w języku wewnętrznym procesora.



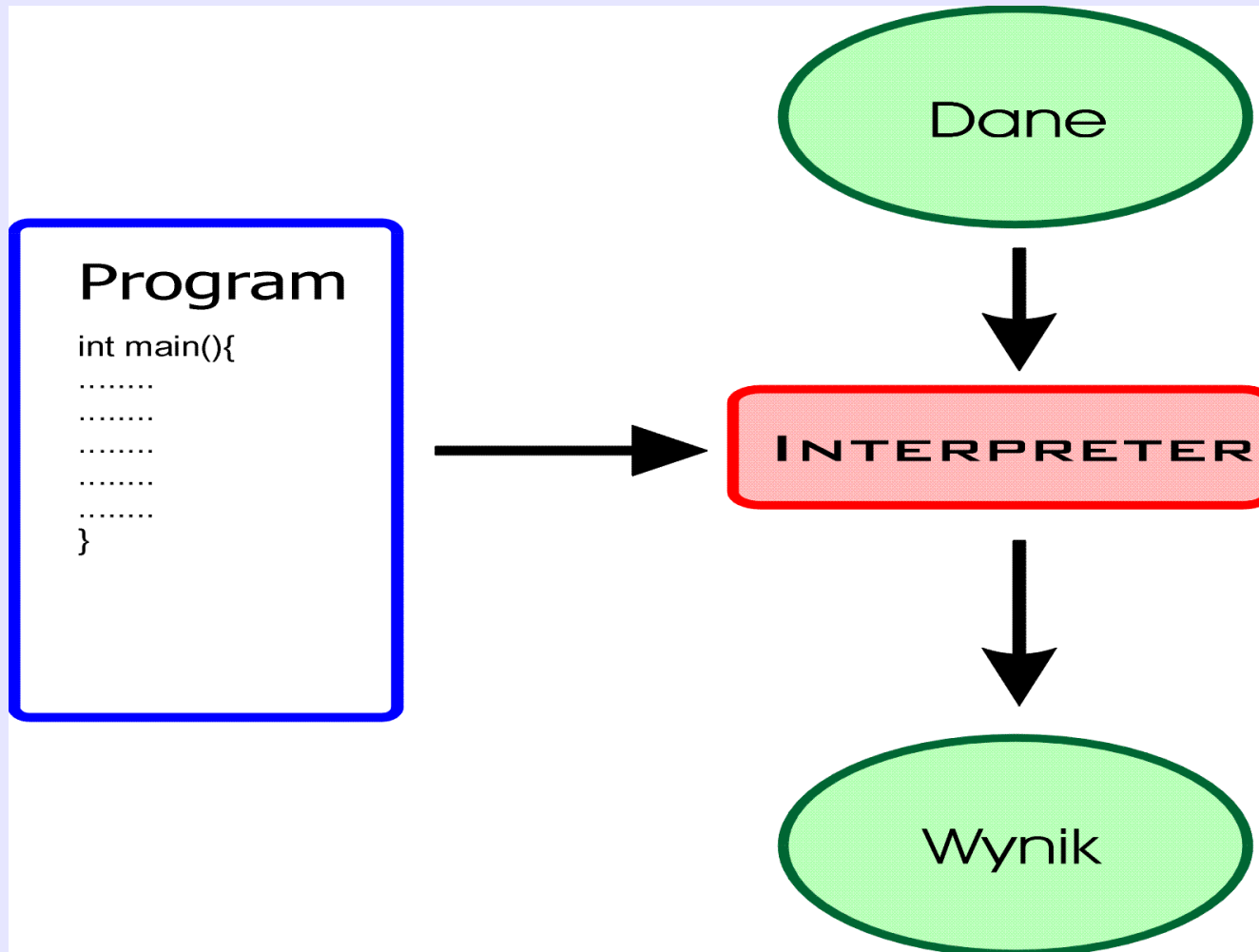
Przykład programu w assemblerze

- **Język wysokiego poziomu** to język, który zaprojektowany jest z myślą o łatwości pisania w nim programów, a nie zgodności z architekturą konkretnego procesora. ■
- Aby wykonać program napisany w języku wysokiego poziomu potrzebny jest program pośredniczący, zwany **translatorem**. ■



Działanie kompilatora

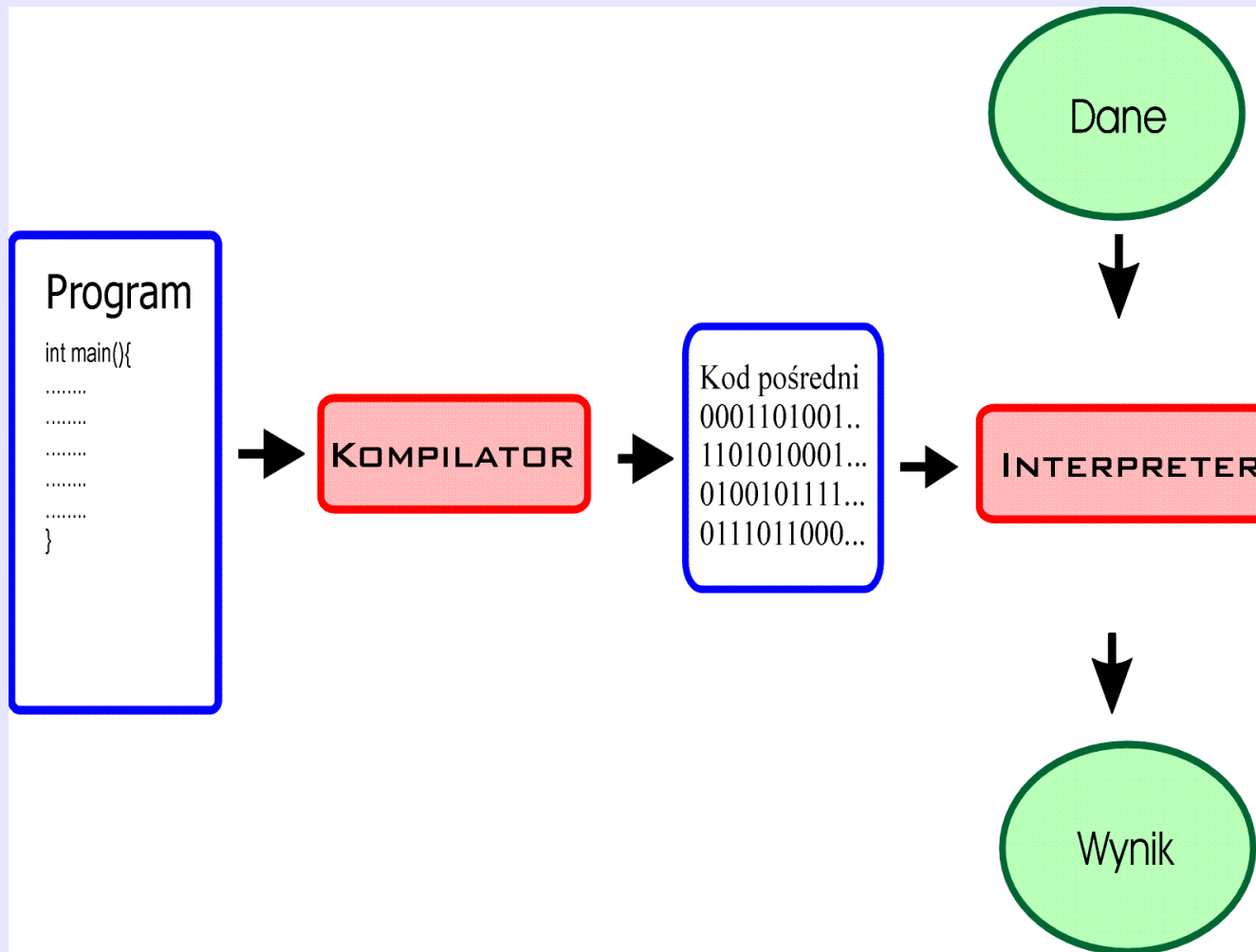
**Kompilator** to program, który na wejściu przyjmuje tekst programu w języku wysokiego poziomu, a na wyjściu dostarcza jego odpowiednik zapisany w języku maszynowym.



Działanie interpretera



**Interpreter** to program, który na wejściu przyjmuje tekst programu w języku wysokiego poziomu oraz dane wejściowe dla tego programu, a na wyjściu dostarcza wynik wykonania programu przeczytanego na wejściu.



Rozwiązanie pośrednie

W rozwiązaniu pośrednim program najpierw jest tłumaczony przez kompilator do formy pośredniej, a następnie forma pośrednia jest interpretowana przez interpreter.

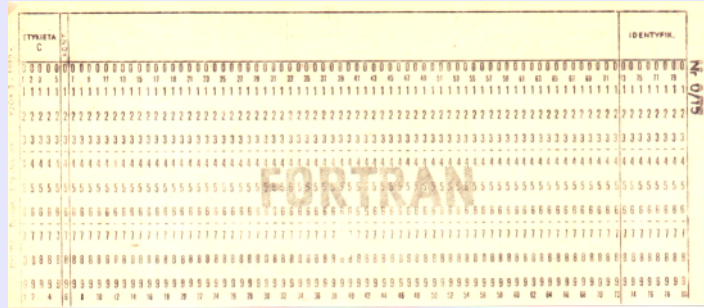
# Zalety i wady kompilatorów i interpreterów. 26

	Zalety	Wady
Kompilator	<ul style="list-style-type: none"><li>• Wysoka efektywność ■</li><li>• Końcowy program może być wykonywany na komputerze, na którym nie ma kompilatora ■</li></ul>	<ul style="list-style-type: none"><li>• Relatywnie niska produktywność ■</li></ul>
Interpreter	<ul style="list-style-type: none"><li>• Relatywnie wysoka produktywność ■</li></ul>	<ul style="list-style-type: none"><li>• Niska efektywność ■</li><li>• Na komputerze, na którym ma być wykonywany program musi być zainstalowany interpreter ■</li></ul>

Dla potrzeb obliczeń naukowych stworzono **Fortran** (FORmula TRANslator). ■  
Z historii Fortranu: ■

- IBM Laboratories, Johna Backus, opracowany dla IBM 704. ■
- projekt w 1954, kompilator (Fortran I) pojawił w 1957. ■
- opracowanie kompilatora zajęło 18 roboczo—lat. ■
- "tradycyjny" zapis formuł matematycznych oraz przechowywanie zmiennych w tablicach ■
- instrukcja pętli. ■
- formatowane instrukcje wejścia—wyjścia ■
- w Fortranie II (1958) możliwość niezależnej kompilacji podprogramów. ■
- Fortran IV (1962): deklaracje typu zmiennej, użycie nazwy podprogramu jako parametru innego podprogramu. ■
- Fortran IV powszechnie używany do 1978, potem Fortran 77, Fortran 90, Fortran 95, Fortran 2003 ■

## Przykład programu w Fortranie



Karta perforowana używana do zapisywania programów w Fortranie

- Fortran uważany jest za pierwszy przykład języka wysokiego poziomu, który wszedł do powszechnego użycia. ■
- Wraz z Fortranem zaczęło się na szeroką skalę programowanie proceduralne ■

**Programowanie proceduralne** polega na

- wyodrębnieniu w problemie szeregu mniejszych, powtarzających się podproblemów ■
- przedstawieniu rozwiązania każdego z podproblemów w postaci samodzielnego małego programu, tak zwanego **podprogramu** ■
- rozwiązaniu właściwego problemu poprzez odwołania do podprogramów ■

- Programowanie proceduralne jest powszechnie stosowane do dziś, współistniejąc z innymi, bardziej zaawansowanymi technikami programowania. ■
- Obok nazwy podprogram stosuje się również nazwę **procedura** oraz **funkcja**. ■

Język **Lisp** (LISt Processor) zaprojektowano dla zastosowań w obliczeniach symbolicznych i sztucznej inteligencji.

- opracowany w latach 1956–58 przez Johna McCarthy'ego w MIT. ■
- pierwszy interpreter 1959–62. ■
- zmiennej długości listy i drzewa jako podstawowe struktury danych ■
- traktowanie kodu programu na równi z danymi ■
- mechanizm automatycznej obsługi pamięci na stercie, t.zw. garbage collection. ■

 Przykład programu w Lispie ■



Język Lisp jest pierwszym językiem programowania wspierającym technikę programowania funkcyjnego. ■

**Programowanie funkcyjne** to technika programowania, w której

- funkcja traktowana jest na równi z danymi i może być argumentem innych funkcji ■
- poza t.zw. wyrażeniem warunkowym, jedyną stosowaną instrukcją jest instrukcja wyliczenia funkcji ■

- programowanie funkcyjne cechuje się m.in.
  - brakiem zmiennych ■
  - traktowaniem programu jako funkcji skonstruowanej z funkcji elementarnych poprzez superpozycję, wyrażenie warunkowe i definicje rekurencyjne. ■
- 
- najważniejszą zaletą: wysoki poziom analizowalności programu ■
- technika ciągle rozwijana tam gdzie kluczowa jest pewność poprawnej implementacji algorytmu ■

- Fortran — własność IBM ■
- 1958 — międzynarodowy komitet dla opracowania specyfikacji uniwersalnego języka komputerowego. ■
- **Algol** (ALGOarithmic Language), (Algol 58, Algol 60). ■

- instrukcja złożona, a w konsekwencji
  - drastyczne ograniczenie konieczności używania instrukcji **goto** ■
  - elegancki i czytelny zapis instrukcji warunkowych (**if—then—else**) i instrukcji pętli (**for**) ■
- 
- pojęcie bloku i zmiennej lokalnej ■
- swobodna forma zapisu ■

# Podwaliny programowania strukturalnego: Algol 35

Podstawową wadą języka Algol był brak specyfikacji dla instrukcji wejścia—wyjścia.

 Przykład programu w Algolu

- **Cobol** (COmmon Business Oriented Language) zaprojektowany przez komitet CODASYL, 1960. ■
- zapis programu był zbliżony do języka naturalnego ■
- przeznaczony do prostych obliczeń na dużych ilościach danych. ■
- jako pierwszy wprowadził hierarchiczne struktury danych (rekordy) ■

 Przykład programu w Cobolu■

Algol zapoczątkował technikę programowania strukturalnego ■

■ **Programowanie strukturalne** to technika programowania, w której uwypukla się strukturę logiczną programu i danych poprzez stosowanie instrukcji strukturalnych, instrukcji złożonych, zmiennych lokalnych oraz grupowanie danych w t.zw. strukturach.

Niepodważalny jest wpływ Algolu na swoich następców: Pascal i C, które filozofię programowania strukturalnego przejęły z właśnie z Algolu.

- 1964: **Basic** (Beginner's All Purpose Symbolic Instruction Code) ■
- opracowany z myślą o początkujących programistach ■
- w latach 70—tych i 80—tych język programistów — amatorów ■

Niestety, popularność ta, ze względu na niestukturalność Basicu, przyczyniła się znacznie do szerzenia się t.zw. **spaghetti code**, t.j. kodu o dużej ilości instrukcji skoku prowadzących w najróżniejsze miejsca w programie, nad którymi trudno zapanaować.

■  
 Mały przykład spaghetti code w Basicu ■

 Przykład programu w Basicu (również spaghetti code) ■

Do tradycji Basicu nawiązuje dzisiejszy Visual Basic, jednak z oryginałem łączy go niewiele więcej ponad nazwę i przeznaczenie ■



# Inne ważniejsze przykłady wczesnych języków wysokiego poziomu 39

## Snobol

- 1962, AT&T Bell Laboratories ■
- przeznaczony specjalnie do przetwarzania tekstów ■

## PL/I

- 1964, IBM ■
- łączył najlepsze cechy Fortranu, Algolu i Cobolu ■

## Pascal

- 1970, Niklausa Wirth ■
- język programowania strukturalnego przeznaczony dla celów dydaktycznych ■

# Pierwszy język wysokiego poziomu dla profesjonalistów: C <sup>40</sup>

Z początkiem lat 70—tych XX wieku Dennis Ritchie z Bell Labs firmy AT&T opracował **język C**.

Cechy języka C: ■

- w pełni strukturalny, ■
- bardzo zwarty; blok oznaczany przez { i } w miejsce **begin** i **end**, ■
- szereg instrukcji bliskich typowym instrukcjom języka maszynowego, np.  $i++$ ,  $++i$ ,  $i+=5$ , ■
- daje prawie nieograniczone możliwości dostępu do pamięci, poprzez wprowadzenie **arytmetyki wskaźników**. ■

# Pierwszy język wysokiego poziomu dla profesjonalistów: C <sup>41</sup>

Olbrzymi sukces języka C w znacznej części spowodowany był faktem, że Ritchie przepisał w języku C system operacyjny Unix, a kompilator języka C był rozpowszechniany razem z Unixem.

**Programowanie w języku logiki** to specyficzna technika programowania, która opiera się na deklarowaniu faktów, a do rozwiązywania problemów wykorzystuje mechanizmy automatycznego dowodzenia twierdzeń.

## Prolog

- Język stworzony w 1972 roku przez francuskiego informatyka Alaina Colmerauera. ■
- Zaprojektowany od podstaw do programowania w języku logiki. ■
- Pierwotnie przeznaczony do analizy zdań w językach naturalnych. ■

## Simula

- 1962—1967, Kristten Nygaard i Ole—Johan Dahl z Norwegian Computing Center ■
- do symulacji procesów rzeczywistych, ■
- traktowany jako bardzo specjalistyczny ■
- wprowadził wszystkie koncepcje fundamentalne dla programowania obiektowego, w szczególności pojęcia **obiekt** oraz **klasa** ■

## Smalltalk

- 1972—1980, Alan Kaya, Dana Ingalls i Adele Goldberg, Xerox Palo Alto Research Center ■
- demonstrował jak idee wprowadzone w Simuli wykorzystać do efektywnego pisania programów niekoniecznie związanych z symulacją. ■
- interpreter języka wyposażono w graficzne środowisko programistyczne oraz środowisko użytkownika (GUI) obsługiwane poprzez myszkę napisane w całości w Smalltalku. ■

# Programowanie obiektowe <sup>44</sup>

Język Simula stworzył podwaliny pod programowanie obiektowe, ugruntowane później w językach Smalltalk i C++.

**Programowanie obiektowe** to technika programowania, która umożliwia rozszerzanie języka programowania tak, by w naturalny sposób można było w nim odzwierciedlić tak dane jak i operacje na nich pojawiające się w rozwiązywanym problemie.

- W technice programowania obiektowego wyróżnia się dwie odmiany:
  - elementarna: programowanie bazujące na obiektach
  - zaawansowana: programowanie obiektowo orientowane
- Programowanie obiektowe bazujące na obiektach charakteryzuje się gromadzeniem danych i operujących na nich funkcji w t.zw. obiektach oraz łączeniem obiektów o wspólnych cechach w tak zwane klasy.
- W programowaniu obiektowo orientowanym dodatkowo wykorzystuje się t.zw. polimorfizm dynamiczny, czyli zróżnicowane zachowanie obiektów w zależności od ich miejsca w tak zwanej hierarchii dziedziczenia.

- 1977–1983, Jean Ichbiah, w ramach kontraktu zleconego przez marynarkę wojenną USA ■
- Ada Lovelace (Augusta Ada King, hrabina Lovelace, 1815–1852) ■
- języki strukturalny, zorientowany na niezawodność przy zachowaniu relatywnie wysokiej efektywności ■
- wysoki stopień kontroli typów, ■
- zaawansowane mechanizmy obsługi pamięci ■
- obsługa sytuacji wyjątkowych ■
- jako pierwszy wprowadził techniki programowania uogólnionego (generycznego) ■
- od 1995 (Ada 95) wspiera programowanie obiektowe ■

W roku 1986 Bjarne Stroustrup pracujący w AT&T Bell Laboratories opracował język C++. ■



Bjarne Stroustrup można odwiedzić pod adresem  
<http://www.research.att.com/~bs/>



- Język C++ jest rozszerzeniem języka C o techniki
  - programowania obiektowego ■
  - programowania uogólnionego ■
  - w pewnym stopniu również programowania funkcyjnego ■
- Jest stosowany między innymi
  - w Microsoft (m.in. Windows XX, Office) ■
  - w Google (search engine) ■
  - w Amazon.com (całe oprogramowanie e-commerce) ■
  - w Sun (HotSpot Java Virtual Machine ) ■
  - we wszystkich najważniejszych produktach Adobe ■
  - w Maya, wykorzystanym m.in. przy produkcji filmów Star Wars Episode I, Spider-Man, Lord of the Rings, Stuart Little ■
  - przy realizacji projektów Apple, AT&T, CERN, Ericsson, HP, IBM, Intel, NASA, Nokia, SGI, Siemens, ■

C++ jest jednym z najczęściej wykorzystywanych języków kompilowanych ogólnego przeznaczenia.

- programowanie obiektów —> pierwsze graficzne interfejsy użytkownika i graficzne systemy operacyjne ■
- rozproszony charakter danych wejściowych generowanych przez mysz —> inne podejście do przetwarzania danych ■
- użytkownik posługując się myszką i klawiaturą generuje zdarzenia ■
- programowanie sprowadzało się do dodania funkcji do wybranych klas, opisujących sposób reakcji obiektów tych klas na zaistniałe zdarzenia. ■
- technika projektowania obiektowego doskonale nadaje się do takiego podejścia do programowania ■
- zdarzenia są reprezentowane jako obiekty pewnych klas ■
- **programowanie orientowane zdarzeniami**: technika programowania oparta na obsłudze zdarzeń ■

Programowanie orientowane zdarzeniami zazwyczaj nie jest wspierane bezpośrednio przez język programowania, ale przez bibliotekę implementującą stosowną hierarchię zdarzeń

# Programowanie uogólnione (generyczne) <sup>50</sup>

Ada, a króko po niej język C++ umożliwiły stosowanie techniki programowania uogólnionego (generycznego). ■

**Programowanie uogólnione** to technika programowania, w której typ danych traktowany jest jako dopuszczalny parametr podprogramu.

- W języku C++ programowanie uogólnione realizowane jest poprzez t.zw. **szablony**. ■
- Programowanie uogólnione pozwala na tworzenie uniwersalnego, bardzo efektywnego kodu na wysokim poziomie abstrakcji ■

- lat 90te XX wieku, Green Team, SUN ■
- oparty na wybranych cechach C++ ■
- zorientowany bardziej na produktywność niż efektywność ■
- język kompilowano—interpretowany. ■
- łatwa przenaszalność oprogramowania przy zachowaniu relatywnie wysokiego tempa wykonania. ■

Java, jako jeden z pierwszych języków wspiera programowanie wielowątkowe, a poprzez swoje biblioteki również programowanie rozproszone. ■

**Programowanie wielowątkowe i rozproszone** to technika programowania, która zakłada rozbicie programu na niezależne wątki, które mogą być wykonywane równolegle przez szereg procesorów (programowanie wielowątkowe), a nawet przez szereg niezależnych komputerów połączonych w sieć (programowanie rozproszone).

- 2000, Andres Hejlsberg, Scotta Wilamuth, Microsoft ■
- główna część przedsięwzięcia .NET (dot—net), które stawia sobie ambitny cel stworzenia uniwersalnego środowiska programistycznego obejmującego szereg narzędzi programistycznych, w szczególności szereg języków programowania, przygotowanych do wzajemnej interakcji na płaszczyźnie Internetu. ■
- język kompilowano—interpretowany ■

- **Języki skryptowe**, to języki zaprojektowane dla szybkiego i relatywnie łatwego pisania programów o stosunkowo małej złożoności obliczeniowej. ■
- Pozwala to na zastąpienie cyklu kompilacja—wykonanie bezpośrednią interpretacją programu ■
- Interpreter, w przeciwieństwie do kompilatora jest w stanie wychwycić i komunikatywnie sygnalizować nie tylko błędy syntaktyczne, programu, ale również błędy pojawiające się w trakcie jego wykonywania ■



- lat 80'te XX wieku, Larry Wall ■
- PERL to skrót od *Practical Report and Extraction Language* ■
- pierwotnie przeznaczony do administrowania systemem operacyjnym UNIX. ■
- obecnie bogaty język, wyposażonego w większość narzędzi współczesnych języków programowania ■
- swoisty styl ■
- interpreter Perla poprzedza proces interpretacji wstępną krótką kompilacją, która odbywa się bezpośrednio przed każdym wykonaniem programu. ■

- Atutem PERLa jest duża elastyczność języka, która pozwala programiście wybrać styl programowania dostosowany do potrzeb wynikających z wielkości zadania i własnych przyzwyczajień. ■
- PERL jest językiem dla programistów zdyscyplinowanych: niestosowne wykorzystanie jego elastyczności pozwala na pisanie bardzo złych i trudnych do pielęgnacji programów. ■

- latach 90'tych XX wieku, Guido van Rossum, National Research Institute for Mathematics and Computer Science w Amsterdamie ■
- wieloparadygmataowy, umożliwiający m.in. programowanie obiektowo orientowane, funkcyjne i uogólnione. ■
- wcięcia zamiast nawiasów do oznaczania bloków kodu ■

- lata 90'te XX wieku, Brendan Eich, Netscape ■
- przeznaczony do programowania w środowisku przeglądarki internetowej ■
- interpreter języka stanowi część przeglądarki ■
- język obiektowy, o składni w pewnym stopniu przypominającej Javę — stąd jego nieco myląca nazwa (wcześniej LiveWire i LiveScript) ■

# Wprowadzenie do C/C++

# C/C++<sub>2</sub>

- C: Dennis Ritchie, początek 1970 r. ■
- w pełni strukturalny, bardzo zwiezły i wydajny, bliski asemblera ■
- C ++: Bjarne Stroustrup, 1986 ■
- głębokie rozszerzenie C w kierunku
  - abstrakcji danych
  - programowania obiektowego
  - programowania uogólnionego



# C/C++<sub>3</sub>

- Od strony formalnej C i C++ to dwa różne języki, ale w praktyce C jest częścią C++.
- W ramach wykładu rozumiemy język C jako zawężenie języka C++ do technik programowania strukturalnego wzbogacone o kilka drobnych udoskonaleń języka C++, w szczególności operacje wejścia-wyjścia.
- Do kompilacji tak rozumianego programu w C używamy kompilatorów C++.

Umowna linia podziału:

- C: programowanie strukturalne
- C++: programowanie obiektowo orientowane■

C/C++ jest jednym z najważniejszych współczesnych języków programowania. Większość nowych języków, na przykład Java i C# jest konstruowanych na bazie C/C++.

# Literatura

- Jerzy Grębosz, *Opus magnum C++ 11. Programowanie w języku C++*
- Jerzy Grębosz, *Opus magnum C++. Misja w nadprzestrzeń C++14/17. Tom 4*
- Jerzy Grębosz, *Pasja C++*
- Jerzy Grębosz, <https://ifj.edu.pl/private/grebosz/> (m.in. kody źródłowe z *książek*)
- Brian W.Kernighan. Dennis M. Ritchie, *C language*
- Bjarne Stroustrup, *C++*
- Bruce Eckel, *Thinking in C++*
- Bruce Eckel, Chuck Allison, *Thinking in C++, Volume 2*
- David Vandevoorde, Nicolai M. Josuttis, *C++ Templates*
- Nicolai M. Josuttis, *C++: Object oriented programming*
- Nicolai M. Josuttis, *C++ Biblioteka standardowa. Programmer's Tutorial*



# Bibliografia<sup>5</sup>

- Tylko dla zainteresowanych.■
- Materiały do kursu są wystarczające.■

# C++ Biblioteka standardowa <sub>6</sub>

- wraz ze standardem C++ opracowana została też biblioteka standardowa
  - funkcje matematyczne■
  - operacje wejścia-wyjścia■
  - obsługa sytuacji wyjątkowych■
  - przetwarzanie tekstu ■
  - pojemniki standardowe (kolekcje)■

W praktyce nawet najprostszy program wymaga korzystania z biblioteki standardowej

# C/C++ Elementarz

# Pierwszy program <sub>8</sub>

```
#include <iostream>
int main(){
    std::cout << "Welcome to the world of C/C++ !!\n";
}
```

# Pierwszy program 9

- Każdy pełny program w C/C++ musi zawierać funkcję o nazwie **main**. To miejsce, w którym rozpoczyna się egzekucja.■

# Pisanie na standardowym wyjściu <sup>10</sup>

- W C++ instrukcja pisania na standardowym wyjściu ma postać

```
std::cout << "Trochę tekstu w cudzysłowie\n";
```

- **std::cout** - standardowe wyjście (ekran)
- `\n` - znak specjalny oznaczający nową linię

# Zmienne <sup>11</sup>

- **Zmienna** - nazwane miejsce w pamięci do przechowywania liczby.
- Zmienne muszą być zadeklarowane przed pierwszym użyciem

```
int number;  
int i, j;  
float celsjus;
```

- **int** - służy do deklarowania zmiennych przechowujących liczby całkowite
- **float** - służy do deklarowania zmiennych przechowujących liczby zmiennopozycyjne

# Czytanie ze standardowego wejścia <sup>12</sup>

- W C++ instrukcja read ma postać

```
std::in >> variable;
```

- **std::in** - standardowe wejście (klawiatura)



# Czytanie, przechowywanie i zapisywanie liczb <sup>13</sup>

```
#include <iostream>
int main(){
    std::cout << "Enter a number: ";
    int number; // We declare an integer variable
    std::cin >> number;
    std::cout << "Your number is " << number << "\n";
}
```

# Pliki nagłówkowe podprogramów bibliotecznych <sup>14</sup>

- **biblioteczny plik nagłówkowy** : konieczny do korzystania z biblioteki

- dla operacji wejścia/wyjścia w stylu c++:

```
#include <iostream>
```

- dla standardowej biblioteki C :

```
#include <cstdlib>
```

- dla funkcji matematycznych

```
#include <cmath>
```

## Instrukcja **if** 15

```
#include <iostream>
int main(){
    std::cout << "Enter a positive number : ";
    int number;
    std::cin >> number;
    if( number <= 0 ){
        std::cout << ". This is not a positive number. ";
    }
}
```

## Instrukcja **if-else** 16

```
#include <iostream>
int main(){
    std::cout << "Enter a positive number : ";
    int number;
    std::cin >> number;
    if( number <= 0 ){
        std::cout << ". This is not a positive number. ";
        std::cout << "Try again\n";
        std::cin >> number;
    } else {
        std::cout << "Thank you\n";
    }
}
```

# Instrukcja **if-else** 17

Wariant instrukcji **else**

```
if (number==0) std::cout << "You entered a zero \n";  
else std::cout << "You entered a nonzero number \n";
```

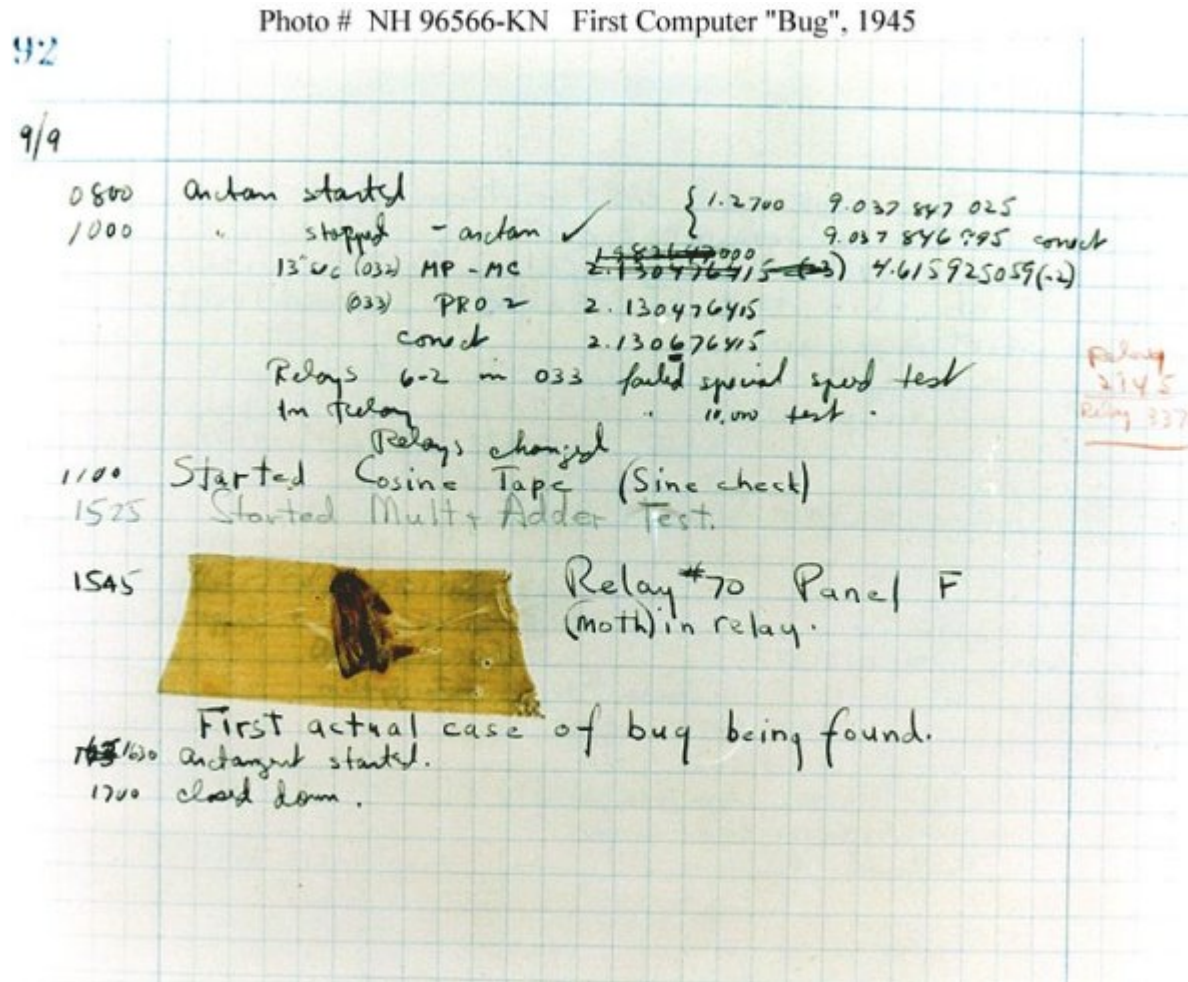
- **Blok** to ciąg instrukcji zamknięty w nawiasy wężsiaste { i }.

```
int n,m;
std::cout << "Enter nonzero n: ";
cin >> n;
if(n==0){
    std::cout << "You entered zero. Enter n again: ";
    cin >> n;
} else {
    std::cout << "Enter m:\n";
    cin >> m;
}
```

- Blok zwany jest również **instrukcją złożoną**

Nie dajemy średnika po nawiasie } zamykającym blok!

# Błędy w programach komputerowych <sup>19</sup>



Pierwszy komputerowy **bug** (robak). Wrzesień 1945. The Mark II computer.

Źródło: US Naval Historical Center

# Błędy w programach komputerowych <sup>20</sup>

- rodzaje błędów

- błędy kompilacji ■
- błędy wykonania ■
- błędy logiczne ■

- błędy kompilacji

- ostrzeżenia ■
- błędy uniemożliwiające dokończenie kompilacji (**Fatal errors**) ■



# Błędy kompilacji <sup>21</sup>

- Zdolności kompilatora do zlokalizowania źródła błędu są ograniczone ■
- Komunikat wskazuje tylko miejsce zagubienia się kompilatora ■
- Zwykle jeden błąd generuje wiele komunikatów o błędach ■

**Exercise 0.1.** Zlokalizuj i popraw błędy kompilacji w następującym programie:

-  `WelcomeWithErrors.cpp`

```
#include <iostream>
int main(){ // begin of the main function
    /* Below we write
    to the standart output */
    std::cout << "Welcome to the world of C/C++ !!\n";
}
```

Program bez dobrze dobranych, czytelnych komentarzy bardzo szybko staje się trudny do zrozumienia, a w konsekwencji jego aktualizacja staje się bardzo pracochłonna.

# Identyfikatory <sup>24</sup>

- **Identyfikator** albo po prostu **nazwa**: używany do nazywania zmiennych, funkcji, typów .... ■
- Identyfikator w C++ : ciąg składający się z liter i cyfr, zaczynający się od litery ■
- litery mogą być wielkie lub małe, ■
- znak podkreślenia jest traktowany jak litera ■
- przykłady: **x**, **alpha1**, **red\_car**, **numberOfCars** ■
- niektóre identyfikatory są zdefiniowane w bibliotece standardowej. Na przykład **std** or **cout** ■

# Formatowanie programów <sup>25</sup>

- Formatowanie służy do zwiększenia czytelności programu przez człowieka
- **białe znaki** — spacja, nowa linia, tabulator etc. ■
- Kompilator potrzebuje białych znaków tylko do oddzielenia identyfikatorów. ■

```
#include <iostream.h>
using namespace std;
int main () {cout << "Nasz pierwszy program"; }
```



Daje to nam swobodę używania białych znaków do zwiększenia czytelności programu na wiele różnych sposobów.

# Słowa kluczowe w C++ <sup>26</sup>

- **using**, **namespace**, **int** to przykłady słów kluczowych.
- **Słowo kluczowe**: identyfikator o specjalnym zastrzeżonym znaczeniu ■
- Nie można go używać w żadnym innym znaczeniu. W szczególności nie można go użyć do nazwania zmiennej ■
- To samo słowo kluczowe może mieć różne znaczenie w zależności od kontekstu. ■

# Słowa kluczowe w C++ <sup>27</sup>

<b>and</b>	<b>and_eq</b>	<b>asm</b>	<b>auto</b>	<b>bitand</b>
<b>bitor</b>	<b>bool</b>	<b>break</b>	<b>case</b>	<b>catch</b>
<b>char</b>	<b>class</b>	<b>compl</b>	<b>const</b>	<b>const_cast</b>
<b>continue</b>	<b>default</b>	<b>delete</b>	<b>for</b>	<b>double</b>
<b>dynamic_cast</b>	<b>else</b>	<b>enum</b>	<b>explicit</b>	<b>export</b>
<b>extern</b>	<b>false</b>	<b>float</b>	<b>for</b>	<b>friend</b>
<b>goto</b>	<b>if</b>	<b>inline</b>	<b>int</b>	<b>long</b>
<b>mutable</b>	<b>namespace</b>	<b>new</b>	<b>not</b>	<b>not_eq</b>
<b>operator</b>	<b>or</b>	<b>or_eq</b>	<b>private</b>	<b>protected</b>
<b>public</b>	<b>register</b>	<b>reinterpret_cast</b>	<b>return</b>	<b>short</b>
<b>signed</b>	<b>sizeof</b>	<b>static</b>	<b>static_cast</b>	<b>struct</b>
<b>switch</b>	<b>template</b>	<b>this</b>	<b>throw</b>	<b>true</b>
<b>try</b>	<b>typedef</b>	<b>typeid</b>	<b>typename</b>	<b>union</b>
<b>unsigned</b>	<b>using</b>	<b>virtual</b>	<b>void</b>	<b>volatile</b>
<b>wchar_t</b>	<b>while</b>	<b>xor</b>	<b>xor_eq</b>	

- **przestrzeń nazw** — blok chroniący swe identyfikatory przed przypadkowym użyciem w innym znaczeniu ■
- **std** — przestrzeń nazw chroniąca nazwy używane w standardowej bibliotece C ++ ■
- aby odwołać się do zmiennej wewnątrz przestrzeni nazw, piszemy nazwę przestrzeni nazw, parę dwukropków i nazwę zmiennej ■
- na przykład: **std::cout** ■



# Pierwszy program ponownie <sup>29</sup>

- Aby zaimportować całą nazwę z przestrzeni nazw, używamy dyrektywy

```
using namespace std;
```



```
#include <iostream.h>
using namespace std;
int main () {
    cout << "Our first program";
}
```

# Operacje arytmetyczne <sup>30</sup>

- Możemy dodawać, odejmować, mnożyć i dzielić liczby i zmienne za pomocą operatorów  $+$ ,  $-$ ,  $*$ ,  $/$ .■
- Aby zachować wynik, podstawiamy go do zmiennej za pomocą operatora  $=$

```
celsjus = (fahrenheit - 32) / 9 * 5;
```

```
#include <iostream>
int main(){
    float celsjus , fahrenheit;
    cout << "temperature in Farenheit degrees: ";
    cin >> fahrenheit; // read a number
    celsjus = (fahrenheit - 32)/9*5; // convert
    cout << fahrenheit << "F = " << celsjus << "C\n";
}
```

**Exercise 0.2.** Napisz, skompiluj i uruchom program, który konwertuje temperaturę w stopniach Celsjusza na temperaturę w stopniach Fahrenheita

**Exercise 0.3.** Napisz, skompiluj i uruchom program, który wczytuje trzy liczby zmiennoprzecinkowe i drukuje ich średnią.■

**Exercise 0.4.** Napisz, skompiluj i uruchom program, który konwertuje upływ czasu podany w godzinach, minutach i sekundach na czas w samych sekundach.

# Inkrementacja i dekrementacja <sup>33</sup>

- Wyrażenie  **$i+=d$**  jest skrótem dla  **$i=i+d$**  ■
- Wyrażenia  **$i++$**  i  **$++i$**  są skrótami dla zwiększenia o jeden:  **$i=i+1$**  ■
- Tak  **$i++$**  jak  **$++i$**  zwracają również wartość: pierwszy operator zwraca  **$i$**  przed inkrementacją, a drugi  **$i$**  po inkrementacji. ■
- Podobnie działają operatory zmniejszania:  **$i--$** ,  **$--i$** ,  **$i-=d$**

**Exercise 0.5.** Napisz, skompiluj i uruchom program, który wprowadza liczbę i wypisuje tę liczbę powiększoną o 1. Eksperymentuj z trzema różnymi, ale równoważnymi notacjami. ■

# Dzielenie całkowite i operator modulo <sup>35</sup>

```
int main(){  
    cout << "7/3= " << 7/3 << "\n";  
    cout << "7/3.= " << 7/3. << "\n";  
    cout << "7%3= " << 7%3 << "\n";  
    cout << "(-7)%3= " << (-7)%3 << "\n";  
}
```

7/3= 2  
7/3.= 2.33333  
7%3= 1  
(-7)%3= -1

## Dzielenie całkowite i operator modulo <sup>36</sup>

- Wynikiem dzielenia dwóch liczb całkowitych jest zawsze liczba całkowita!
- Operator modulo % zwraca resztę po podzieleniu jednej liczby całkowitej przez drugą.



- Utworzenie tablicy

```
int age[12];
```

- Dostęp do elementów

```
age[0] = 2; age[1] = 5; age[2] = 7;
```

Numerowanie obiektów w tablicy zaczyna się od zera!

- ```
for ( i=0; i < 12; i=i+1 ){  
    std::cout << "Podaj zarobki w miesiacu ";  
    std::cout << i+1 << ": ";  
    std::cin >> zarobki[i];  
}
```

**float** suma=0;  
**for** ( i=0; i < 12; i=i+1) suma=suma+zarobki[i];

## • Funkcje <sup>39</sup>

```
float cubeVolume(float r){  
    return r*r*r;  
}
```

- jeden argument typu **float** i zwracana wartość typu **float** ■
- struktura: **nagłówek** oraz **ciało funkcji** ■
- lista argumentów może być pusta ■
- **void** zamiast zwracanego typu: funkcja nic nie zwraca

```
int main(){  
    cout << "Enter the edge of the cube: ";  
    float edge;  
    cin >> edge;  
    cout << "The edge is ";  
    cout << edge << "\n";  
    cout << "The volume of the cube is ";  
    cout << cubeVolume(edge) << "\n";  
}
```

- Chociaż niektóre kompilatory na to pozwalają, użycie **void** jako typ zwracany przez funkcję **main** jest niezgodny ze standardem C / C ++ ■
- Wartość zwracana przez funkcję **main** jest używana przez system operacyjny do ustalenia czy wywołanie programu zakończyło się sukcesem (wartość zero) czy błędem (wówczas zwracany jest kod błędu). ■

**Exercise 0.6.** Napisz funkcję, która przyjmuje jako argument dwa boki prostokąta i zwraca pole prostokąta. Napisz, skompiluj i uruchom program główny, który demonstruje użycie tej funkcji.

- Math functions require the **cmath** library

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

int main(){
    cout << "Square root of 2 is " << sqrt(2) << "\n";
    cout << "Absolute value of 2 is " << abs(2) << "\n";
    cout << "Natural log of 2 is " << log(2) << "\n";
    cout << "Sinus of 2 is " << sin(2) << "\n";
    cout << "Cosinus of 2 is " << cos(2) << "\n";
    cout << "Tangens of 2 is " << tan(2) << "\n";
}
```

**Exercise 0.7.** Napisz funkcję **przekatna**, która przyjmuje jako argument dwa boki prostokąta i zwraca przekątną prostokąta. Napisz, skompiluj i uruchom program główny, który demonstruje użycie tej funkcji.

Pole trójkąta o bokach  $a, b, c$  można obliczyć według wzoru Herona

$$T = \sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = \frac{a + b + c}{2}.$$

**Exercise 0.8.** Napisz funkcję **poleTrojkata**, która przyjmuje jako argumenty trzy boki trójkąta i zwraca pole trójkąta. Napisz, skompiluj i uruchom program główny, który demonstruje użycie tej funkcji.

1

2.



## Instrukcje warunkowe

# Warunki<sub>3</sub>

- **warunek**: wyrażenie equiuowane do **true** albo **false** ■
- podstawowe przykłady:

$x < y$ ,  $x \leq y$ ,  $x > y$ ,  $x \geq y$ ,  $x == y$ ,  $x != y$

- warunki są używane w instrukcjach warunkowych ■
- każda niezerowa liczba traktowana jako warunek jest konwertowana na **true** ■
- zero traktowane jako warunek jest konwertowane na **false** ■

## Warunki<sub>4</sub>

- Automatyczna konwersja wartości liczbowych na warunki jest zachowana w C ++ dla kompatybilności wstecznej, ale może prowadzić do niebezpiecznych błędów
- dzieje się tak, gdy warunek  $x == y$  przez pomyłkę zapisano jako  $x=y$  ■
- W pierwszym przypadku mamy do czynienia z operatorem porównania który zwraca **true** wtedy i tylko wtedy gdy  $x$  jest równe  $y$ . ■
- W drugim przypadku mamy do czynienia z operatorem przypisania, który zwraca  $y$  skonwertowane do **true** gdy  $y$  jest niezerowe, a do **false** gdy  $y$  wynosi zero.

**Exercise 2.1.** Napisz program, który czyta ze standardowego wejścia liczby  $x$  i  $y$ , ewaluuje wyrażenia  $x == y$  i  $x = y$  i wypisuje wynik. Eksperymentuj z danymi wejściowymi, aby zobaczyć, jak różnią się dwa wyrażenia w zależności od wartości  $x$  i  $y$

**Exercise 2.2.** Rozszerz powyższy program, aby również ewaluował i drukował wyrażenia  $x < y$ ,  $x > y$ ,  $x \leq y$ ,  $x \geq y$  i  $x \neq y$ .

# Instrukcja **if**<sub>6</sub>

```
#include <iostream>
int main(){
    std::cout << "Enter a positive number:";
    int number;
    std::cin >> number;
    if(number <= 0) std::cout << ". Wrong number.\n";
}
```

# Instrukcja **else**<sub>7</sub>

- wariant z **else**

```
if (number <= 0){  
    std::cout << ". This is a wrong number.\n";  
    std::cout << "Try again: ";  
    std::cin >> number;  
}else{  
    std::cout << "This is a positive number.\n";  
    std::cout << "Thank you\n";  
}
```

# Instrukcje i bloki<sub>8</sub>

- **Instrukcja** - pojedyncza instrukcja zakończona średnikiem
- **Blok** - sekwencja instrukcji ujęta w nawiasy klamrowe { ... }

```
{  
    std::cout << "Wrong number.\n";  
    std::cout << "Try again: ";  
    std::cin >> number;  
}
```

- blok nazywany jest również **instrukcją złożoną** ■

Nie umieszczamy średnika po zamknięciu nawiasu }!!!

# Bloki 9

- Blok nie jest potrzebny w przypadku wystąpienia pojedynczej instrukcji **if** albo **else**

```
if (number < 0) std::cout << "Negative number.\n";  
else    std::cout << "Non-negative number.\n";
```





**Exercise 2.3.** Zmodyfikuj program, obliczający pole prostokąta, aby sprawdzał, czy wysokość i szerokość podane przez użytkownika są prawidłowe. Wydrukuj wiadomość informującą o złych danych, jeśli podane zostaną nieprawidłowe dane.■

**Exercise 2.4.** Wzór Herona na pole trójkąta może się nie powieść, jeśli podane liczby nie są bokami trójkąta. Zmodyfikuj program obliczający pole trójkąta tak, aby wykrywał, czy dostarczone dane są poprawne przed obliczeniem pola. Wydrukuj wiadomość informującą o złych danych, jeśli podane zostaną nieprawidłowe dane.■

# Bloki zagnieżdżone <sup>11</sup>

- Blok może być zagnieżdżony w innym bloku

```
if (a != 0){  
    if (a < 0){  
        std::cout << "a is negative.\n";  
    } else {  
        std::cout << "a is positive.\n";  
    }  
} else {  
    std::cout << "a is zero.\n";  
}
```

- 
- Zwiększamy czytelność poprzez głębsze wcięcie wewnętrznych bloków.

**Exercise 2.5.** Napisz program, który wypisze rozwiązanie równania kwadratowego  $ax^2 + bx + c = 0$  dla parametrów  $a, b, c$  odczytanych ze standardowego wejścia. Przeanalizuj przypadki, w których istnieją dwa, jedno lub brak rozwiązania.

**Exercise 2.6.** Napisz program, który wypisze rozwiązanie układu równań liniowych

$$ax + by = c$$

$$dx + ey = f$$

dla parametrów  $a, b, c, d, e, f$  odczytanych ze standardowego wejścia. Przeanalizuj przypadki, gdy istnieje dokładnie jedno rozwiązanie, brak rozwiązania lub nieskończenie wiele rozwiązań.

**Exercise 2.7.** Napisz program, który wypisze rozwiązanie nierówności

$$|x + a| > |3x + b|$$

dla parametrów  $a, b$  odczytanych ze standardowego wejścia. Przeanalizuj liczbę rozwiązań.

## operator wyrażenia warunkowego <sup>13</sup>

- używany w wyrażeniach: zamiast instrukcji warunkowej możemy użyć wyrażenia warunkowego ?: ■
- Instrukcja

```
if (u > 0) x=u; else x=0;
```

jest równoważne ewaluacji wyrażenia

```
x= (u > 0 ? u : 0);
```


**Exercise 2.8.** Napisz program, który wykorzystuje operator wyrażenia warunkowego do drukowania wartości bezwzględnej liczby odczytanej ze standardowego wejścia.■

**Exercise 2.9.** Napisz program, który wykorzystuje operator wyrażenia warunkowego do wypisania maksimum i minimum dwóch liczb odczytanych ze standardowego wejścia.■

- instrukcja **switch**

```
switch(integerExpression) {  
    case e_1: statementList_1;  
    case e_2: statementList_2;  
  
    .....  
    case e_n: statementList_n;  
    default: statementList;  
}
```



- *e\_1, e\_2 ... e\_n* muszą być liczbami całkowitymi znanymi w czasie kompilacji 

- The most common form

```
switch(integerExpression) {  
    case e_1: statementList_1; break;  
    case e_2: statementList_2; break;
```

```
    .....  
    case e_n: statementList_n; break;  
    default: statementList;  
}
```



**Exercise 2.10.** Napisz program, który wczyta ze standardowego wejścia numer dnia w tygodniu i użyje instrukcji wyboru do wydrukowania nazwy tego dnia. Wykorzystaj instrukcję **default** do poinformowania, że podany numer jest niepoprawny . ■

**Exercise 2.11.** Napisz program, który wczyta ze standardowego wejścia numer miesiąca i użyje instrukcji wyboru do wydrukowania nazwy tego miesiąca. Wykorzystaj instrukcję **default** do poinformowania, że podany numer jest niepoprawny . ■

**Exercise 2.12.** Napisz program, który wczyta ze standardowego wejścia numer miesiąca i użyje instrukcji wyboru do wydrukowania liczby dni tego miesiąca. Wykorzystaj instrukcję **default** do poinformowania, że podany numer jest niepoprawny . ■



## Pętle i tablice

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

int main(){
    for(int i=0;i<5;i++){
        cout << "Square of " << i;
        cout << " is " << i*i << "\n";
    }
}
```

## Drukowanie ciągu liczb 20

**Exercise 2.13.** Napisz program, który drukuje pierwiastki kwadratowe liczb całkowitych od 1 do 10.■

**Exercise 2.14.** Napisz program, który drukuje kwadraty liczb zmiennoprzecinkowych od  $a = 1.0$  do  $b = 2.0$  w krokach co  $d = 0.1$ .■

**Exercise 2.15.** Zmodyfikuj powyższy program, aby użytkownik mógł podać  $a$ ,  $b$  i  $d$ .

**Exercise 2.16.** Napisz program, który wypisuje elementy sekwencji  $\frac{1}{n^2+1}$  dla  $n = 1, 2, \dots, 7$ .■

## Suma liczb naturalnych<sub>21</sub>

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

int main(){
    int s=0;
    int n=1000;
    for(int i=1;i<=n;i++) s+=i;
    cout << "Sum of squares of integers from 1 to "
    cout << n << " is " << s << "\n";
}
```

## Drukowanie sumy ciągu liczb <sup>22</sup>

**Exercise 2.17.** Napisz program, który wypisze sumę kwadratów liczb całkowitych od 1 do 1000.■

**Exercise 2.18.** Napisz program, który wypisze sumę pierwiastków kwadratowych liczb całkowitych od 1 do 1000.■

**Exercise 2.19.** Napisz program, który wypisze iloczyn liczb całkowitych od 1 do 10.■

**Exercise 2.20.** Napisz funkcję o nazwie *silnia*, która zwraca iloczyn liczb całkowitych od 1 do  $n$ , gdzie liczba  $n$  jest argumentem funkcji. Napisz program główny, który odczyta liczbę  $n$  ze standardowego wejście i użyje funkcji *silnia* do drukowania iloczynu liczb całkowitych od 1 do  $n$ .

# Tablice <sup>23</sup>

- **Tablice** - zmienne, w których można przechowywać więcej niż jedną liczbę
- Aby zadeklarować tablicę piszemy:

```
int age[5];
```


- Aby wykorzystać elementy tablicy:

```
age[0]=2; age[1]=5; age[2]=7; age[3]=10; age[4]=14;
```

Numeracja obiektów w tablicy zaczyna się od zera!

## Czytanie tablic<sup>24</sup>

```
float wages[12];  
for(int i=0; i<12; i++){  
    cout << "Wages in the " << i+1 << "th month: ";  
    cin >> wages[i];  
}
```



```
float s=0;  
for(int i=0; i<12; i++) s+=wages[i];  
cout << "Your total income is " << s << "\n";
```

**Exercise 2.21.** Napisz program, który odczyta ciąg 10 liczb i wypisze średnią wartość oraz odchylenie standardowe liczb w tym ciągu.■

**Exercise 2.22.** Zmodyfikuj program w poprzednim ćwiczeniu, aby mógł przetwarzać zbiór liczb o dowolnej wielkości. W tym celu najpierw przeczytaj liczbę liczb w kolekcji.■



```
int main(){  
    const int length=30;  
    char text[length];  
    cout << "Enter a word: ";  
    gets(text);  
    cout << "You wrote: " << text << "\n";  
}
```



```
int main(){
    const int length=30;
    char text[length];
    cout << "Enter a word: ";
    int i;
    for(i=0;i<length-1;++i){
        char c;
        c=getchar();
        if(c <= ' ') break;
        text[i]=c;
    }
    text[i]=0;
    cout << "You wrote: " << text << "\n";
    cout << "Inverted: ";
    for(int j=i-1;j>=0;--j) cout << text[j];
    cout << "\n";
}
```



**Exercise 2.23.** Napisz program, który odczytuje wiersz tekstu ze standardowego wejścia i wypisuje liczbę cyfr w tym tekście.

**Exercise 2.24.** Napisz program, który odczytuje wiersz tekstu ze standardowego wejścia i drukuje liczbę oddzielnych słów w tekście. Przez słowo rozumiemy ciąg znaków inny niż spacja.

```
float x;  
cout << "Enter a  number: ";  
cin >> x;  
while(x>=1) —x;  
cout << "Fractional part of your number is: ";  
cout << x << "\n";
```

**Exercise 2.25.** Napisz program, który oblicza ułamkową część odczytanej liczby zmiennoprzecinkowej ze standardowego wejścia. Użyj pętli **while**.

## Instrukcja **do-while** 30

```
float x;  
do{  
    cout << "Give me a positive number: ";  
    cin >> x;  
}while(x<=0);  
cout << "Your positive number is: ";  
cout << x << "\n";
```

**Exercise 2.26.** Napisz program, który odczytuje liczbę zmiennoprzecinkową  $x$  ze standardowego wejścia taką, że  $0 < x < 1$ . Użyj pętli do-while.

**Exercise 2.27.** Napisz program, który odczytuje dodatnią liczbę zmiennoprzecinkową i oblicza pierwiastek kwadratowy z dobrą dokładnością. Użyj ciągu

$$\begin{cases} x_1 := 1, \\ x_{n+1} := \frac{x_n^2 + a}{2x_n} \end{cases}$$

który zbiega do  $\sqrt{a}$ . ■

## Instrukcja **break** 32

```
float x;  
while(true){  
    cout << "Give me a positive number: ";  
    cin >> x;  
    if(x>0) break;  
    cout << "This is not a positive number.\n";  
};  
cout << "Your positive number is: ";  
cout << x << "\n";
```

**Exercise 2.28.** Napisz program, który odczyta ciąg 10 liczb naturalnych i wydrukuje pozycję pierwszej liczby w ciągu, która dzieli ostatnią liczbę w ciągu. ■



## Instrukcja **goto** 34

```
for(int i=0; i<12; i++)  
for(int j=i+1; j<12; j++){  
    if(wages[i]==wages[j]){  
        cout << "Wages are the same in months ";  
        cout << i+1 << " and " << j+1 << "\n" ;  
        goto done;  
    }  
}  
cout << "Each month the wages are different\n";  
done;;
```

**Exercise 2.29.** Napisz program, który odczytuje ciąg 10 liczb i odpowiada na pytanie, czy istnieje para różnych liczb w ciągu, taka że jedna liczba pary dzieli drugą.■

**Exercise 2.30.** Napisz program, który odczytuje ciąg 10 liczb i odpowiada na pytanie, czy istnieje para różnych liczb w ciągu, taka że ich suma wynosi 10.■

## Instrukcja **continue**<sup>36</sup>

```
float limit=1500;
int cnt=0;
for(int i=0; i<12; i++){
    if(wages[i]>limit) continue;
    ++cnt;
}
cout << "Your wages did not exceed the limit";
cout << " in " << cnt << " months.\n";
```

1

3.

## C/C++ - typy danych

# Podstawowe typy danych<sub>3</sub>

- **char** — Jednocześnie działa jako typ znakowy oraz liczbowy typ dla małych liczb całkowitych ■
- **int** — podstawowy typ dla liczb całkowitych ■
- **float** — podstawowy typ liczb zmiennoprzecinkowych ■
- **double** — typ liczb zmiennoprzecinkowe z podwójną precyzją ■
- **bool** — typ zmiennych logicznych (Boolowskich) ■

# Typ zmiennych logicznych<sup>4</sup>

- type **bool**: dwie wartości: **true** albo **false** ■
- dostępny tylko w C++ ■
- w C: fałsz reprezentowany jako zero a prawda jako wartość różna od zera ■
- dla kompatybilności z C w C++ są stosowane automatyczne konwersje (niestety - może to prowadzić do błędów) ■

# Typ zmiennych logicznych<sup>5</sup>

- wyrażenie w nawiasach po **if** lub **while**: **wyrażenie boolowskie** ■
  - wyrażenia boolowskie budowane są przy pomocy słowa kluczowego **bool** i operatorów logicznych:
    - **and**, również pisane **&&** ■
    - **or**, również pisane **||** ■
    - **not**, również pisane **!** ■
- 
- operatory porównania **<**, **>**, **<=**, **>=**, **==**, **!=**: zwracają wartość typu **bool** ■



## Typ zmiennych logicznych<sub>6</sub>

```
int main(){
    bool b=false;
    cout << "b=" << b << endl;
    cout << "!b=" << !b << endl;

    int j=2,k=-1;
    b= (j>0) && (k!=0);
    cout << "b=" << b << "\n";
    if(b) cout << "b is true\n";
    else cout << "b is false\n";

    b= (j<0) || (k==0);
    cout << "b=" << b << endl;
    if(b) cout << "b is true\n";
    else cout << "b is false\n";
}
```

# Liczby pierwsze<sub>7</sub>

**Exercise 3.1.** Napisz funkcję z jednym argumentem całkowitym bez znaku, która zwraca **true**, jeśli argument jest liczbą pierwszą, a w przeciwnym razie **false**.

# Typ char<sub>8</sub>

- **char**: służy do przechowywania 8-bitowych liczb całkowitych
- interpretowane jako kody znaków w standardzie ASCII ■
- literały znakowe pisane zamknięty apostrofami

```
char x='A' ;
```

■

- na wydruku pokazuje znak, a nie jego kod ■

# Specyfikatory<sup>9</sup>

Niektóre typy danych C / C++ mogą być modyfikowane za pomocą tzw. specyfikatorów.

**Specyfikatory:** **short**, **long**, **signed** oraz **unsigned**. ■

## Specyfikatory **short** oraz **long**

- sugerują pojemność (maksymalny zakres liczb) odpowiednio niższą lub wyższą niż typ podstawowy. ■
- Rzeczywista pojemność zależy od kompilatora ■
- C/C++ wymaga jedynie by
  - typ **char** miał minimalną pojemność 8 bitów, ■
  - typ **short int** miał minimalną pojemność 16 bitów, ■
  - typ **long int** miał minimalną pojemność 32 bitów, ■

# Specyfikatory **signed** oraz **unsigned** 10

- określają, czy jeden bit ma być używany jako znak, czy nie. Wpływa to na pojemność typu. ■
- mogą być używane z typami **char** i **int**. ■

Rzeczywistą pojemność (w bajtach) dla każdego typu można sprawdzić, wywołując operator **sizeof**, przy czym argumentem jest nazwa typu lub zmienna danego typu.

# Typ wyliczeniowy <sup>11</sup>

- Pisząc

```
enum weekday {Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

definiujemy własny typ, tak zwany **typ wyliczeniowy**. ■

- Możemy teraz deklarować zmienne typu **weekday**:

```
weekday payDay=pia;
```

■

- Kompilator kojarzy liczby całkowite z elementami typu wyliczeniowego, zaczynając od zera. Można to zmodyfikować poprzez jawne skojarzenie określonego typu elementu z inną liczbą:

```
enum controlKeys {tab=8, enter=13 }
```

■

## Typ wyliczeniowy <sup>12</sup>

- Nie możemy bezpośrednio przypisać wartości całkowitej do zmiennej typu wyliczeniowego. Konieczna jest jawna konwersja. ■
- Zatem instrukcja

```
payDay=4; // brak konwersji  
spowoduje błąd. ■
```

- Możemy napisać

```
payDay=weekday(7);
```

choć nie przypisuje to żadnego dnia z listy, ponieważ sobota to dzień tygodnia (6), a niedziela to dzień tygodnia (0).

```
enum weekDay{Sun, Mon, Tue, Wed, Thu, Fri, Sat};  
int main(int argc, char* argv[]){  
    using namespace std;  
  
    weekDay payDay=Fri;  
  
    //payDay=4; // lack of conversion  
    //payDay=weekDay(-3); // non-existing value  
    //payDay=weekDay(8); // non-existing value  
    std::cout << payDay << endl;  
}
```



## Typ `void` <sup>14</sup>

- Typ `void` jest używany w C/C++ do zaznaczenia typu pustego, to jest typu, który nie przyjmuje żadnych wartości. ■
- Dla wtajemniczonych: Jest on używany ze wskaźnikami do konstruowania wskaźników do obiektów nieznanego typu:

```
void * p;
```



- Jest używany również z funkcjami, aby wskazać, że funkcja nie zwraca żadnej wartości:

```
void f ();
```

## Słowo kluczowe **typedef** <sup>15</sup>

- Słowo kluczowe **typedef** nie definiuje nowego typu.
- Służy do nadania nowej nazwy istniejącemu typowi. ■
- Nowa nazwa może być krótsza i bardziej opisowa ■

```
typedef unsigned long int Ulong;  
typedef int* IntPtr; //Dla wtajemniczonych  
Ulong, j;  
IntPtr p, q;
```

# Kontrola typów w C/C++ <sup>16</sup>

**Kontrola typów** to działanie kompilatora, którego celem jest zminimalizowanie błędów związanych z niezgodnością typów przy podstawieniu lub wywołaniu funkcji. Dzieli się na **statyczną** t.j. wykonywaną w czasie kompilacji oraz **dynamiczną**, wykonywaną przez kod wygenerowany przez kompilator.■

- C ma stosunkowo słabą kontrolę typów. ■
- C++ charakteryzuje się stosunkowo silną, ale tylko statyczną kontrolą typów ■
- Jednak w celu zagwarantowania zgodności z C sprawdzanie typu w C++ nie jest pełne. ■
- Brak dynamicznego sprawdzania typu wynika z orientacji C++ na wydajność kodu. ■

- **literał**: wartość zmiennej wprowadzonej bezpośrednio do programu ■
- Literały są używane do inicjalizacji zmiennych oraz jako argument funkcji i operatorów ■

# Literały całkowite <sup>18</sup>

- reprezentują liczby całkowite ■
- – Dziesiętne: 2015, −1 ■
  - ósemkowe, rozpoczynają się zerem: 00, 02, 0123 ■
  - szesnastkowe, rozpoczynają się 0x, używają dodatkowych cyfr a,b,c,d,e,f: 0x1a7d ■
- 
- przyrostek U: typ **unsigned int**, na przykład, 7U ■
- przyrostek L: type **long int** na przykład 1L ■

# Literały zmiennoprzecinkowe <sup>19</sup>

- reprezentują liczby zmiennoprzecinkowe
  - pisane z kropką dziesiętną: 3.14, 1.0 ■
  - lub z kropką dziesiętną i wykładnikiem: 1.23e - 45, -1.17e10 ■
- domyślnie są typu **double** ■
- pisane z przyrostkiem f są typu **float** eg. 3.14f ■

Są tylko dwa: **false** i **true**

- pisane jako znak w apostrofach ■
- Reprezentują wartość liczbową kodu znakowego w standardzie ASCII ■
- Istnieją również znaki specjalne poprzedzone odwrotnym ukośnikiem: `'\n'` (znak zmiany linii), `'\t'` (tabulator) ■
- Mogą być zapisane kodem ósemkowym poprzedzonym odwrotnym ukośnikiem i zerem, n.p. `'\012'` ■
- Mogą być zapisane kodem szesnastkowym poprzedzonym odwrotnym ukośnikiem i znakiem x, n.p. `'\xff'` ■



- **Literał tekstowy**: ciąg znaków ujęty w cudzysłów

```
"This is a text literal \n"
```



- używane do wypisywania tekstu na standardowym wyjściu
- mogą być używane do inicjalizacji zmiennych typu string

```
cout << "Tests of character and string literals:\n";
char c='?', d='\t' e='\x21';
cout << "c=" << c << "d=" << d << "e=" << e << "\n";
c='\\';
d='\\';
e='\042';
cout << "c=" << c << "d=" << d << "e=" << e << "\n";
cout << "\nTests of integer literals:\n";
int i=1, j=020, k=0xff;
cout << "i=" << i << "j=" << j << "k=" << k << "\n";
cout << "\nTests of floating literals:\n";
float x=7.14, y=6e3, z=11.3e-4;
cout << "x=" << x << "y=" << y << "z=" << z << "\n";
```

# Deklarowanie zmiennych <sup>24</sup>

- Deklarujemy zmienne, wpisując typ, a następnie listę nazw zmiennych

```
int counter,number;
```

W C/C++ wszystkie zmienne muszą być zadeklarowane przed ich użyciem.

# Zmienne lokalne i globalne <sup>25</sup>

- **zmienna lokalna** : deklarowana wewnątrz bloku ■
- **zmienna globalna**: deklarowana na zewnątrz bloku ■

## Zakres ważności zmiennej <sup>26</sup>

- Zakres ważności zmiennej globalnej jest od deklaracji do końca pliku
- Zakres ważności zmiennej lokalnej jest od deklaracji do końca najbardziej wewnętrznego bloku zawierającego deklarację

## Czas życia zmiennej<sup>27</sup>

- Czas życia zmiennej globalnej jest od samego początku wykonywania do samego końca wykonywania programu
- Czas życia zmiennej lokalnej jest od momentu realizacji deklaracji do momentu, w którym wykonanie programu przechodzi przez koniec najbardziej wewnętrznego bloku zawierającego deklarację.

## Zaśnianie zmiennych 28

```
/* Declaration of a global variable */
char c = 'G' ;

int main(){
    cout << "Execution begins\n";
    cout << "Global variable c contains " << c << "\n";
    {
        cout << "Entering a block\n";
        cout << "Global variable c is " << c << "\n";
        /* Declaration of a local variable */
        char c = 'L' ;
        cout << "Local variable c is " << c << "\n" ;
        cout << "Global variable c is " << ::c << "\n" ;
        cout << "Leaving the block\n";
    }
    cout << "Global variable c is " << c << "\n";
}
```

## Zaślanianie zmiennych lokalnych <sup>29</sup>

```
int main(){
    cout << "Execution begins\n";
    char c = 'G' ;
    cout << "Local variable c is " << c << "\n" ;
    {
        cout << "Entering internal block\n";
        cout << "External c is " << c << "\n" ;
        /* Declaration of an internal local variable */
        char c = 'L' ;
        cout << "Internal c  is " << c << "\n" ;
        cout << "External c is inaccessible\n" ;
        cout << "Leaving internal block\n";
    }
    cout << "External c is " << c << "\n" ;
}
```



# Statyczne zmienne lokalne <sup>30</sup>

- **Statyczna zmienna lokalna**: zmienna z zakresem ważności jak zmienna lokalna i czasem życia zmiennej globalnej ■

- 

```
static int i;
```

■

- statyczna zmienna lokalna jest inicjowana tylko raz podczas ładowania programu do pamięci!

## Statyczne zmienne lokalne <sup>31</sup>

```
int main(){
    int n=3;

    for(int i=0;i<n;i++){
        int cnt=0;
        static int staticCnt=0;
        cnt++;
        staticCnt++;
        cout << "cnt=" << cnt << "\n";
        cout << "staticCnt=" << staticCnt << "\n";
    }
    // staticCnt is invisible here
    // cout << "staticCnt==" << staticCnt << "\n";
}
```

## Obiekty stałe <sup>32</sup>

Mamy trzy metody używania stałych obiektów:

- poprzez literał

```
field=3.14 * r * r;
```



- przez dyrektywę preprocesora

```
#define PI 3.14  
....  
field=PI * r * r;
```



- przez modyfikator **const** (najlepsze rozwiązanie)

```
const float pi=3.14;  
....  
field=pi * r * r;
```



# Odnosi

# Typ odnośnikowy i odnośniki <sup>2</sup>

- **typ odnośnikowy** pochodny względem typu **T**: typ, którego obiekty służą do pokazywania na obiekty typu **T** ■
- **odnośniki**: obiekty typu odnośnikowego. ■
- inaczej: obiekty, które zawierają jedynie adresy innych obiektów ■
- odnośniki zazwyczaj przechowywane są w zmiennych ■
- obiekty wskazywane przez odnośniki: tak zmienne jak i obiekty anonimowe ■

W przeciwieństwie do nazw, odnośniki w trakcie kompilacji nie znikają. W skompilowanym programie odnośniki pozostają obiektami, które pokazują na inne obiekty.

# Interpretacja typów odnośnikowych<sub>3</sub>

- dwie interpretacje: bliska, daleka ■
- bliska: przy użyciu odnośnika interpretujemy go dosłownie jako adres obiektu, na który odnośnik pokazuje ■
- daleka: przy użyciu odnośnika interpretujemy go jako obiekt, na który odnośnik pokazuje ■
- interpretacja bliska zawsze przy inicjalizacji odnośnika ■
- w pozostałych sytuacjach: tak interpretacja bliska jak i daleka, w zależności od konkretnego rodzaju odnośnika i kontekstu ■

# Referencje <sup>4</sup>

- **referencja**: odnośnik, dla których zawsze stosowana jest interpretacja daleka ■
- C++:

```
int n=1;  
int& r = n;  
r=2;
```

■

# Wskaźniki<sub>5</sub>

- **wskaźnik**: odnośnik, dla których zawsze stosowana jest interpretacja bli-ska ■
- C++:

```
int n;  
int* p=&n;
```

■

- odniesienie do obiektu, na który wskaźnik pokazuje wymaga wykonania jawnie operatora \*, t.zw. **operatora wyłuskania** ■
- C++:i

```
*p=5;
```

■



Odnośników używa się

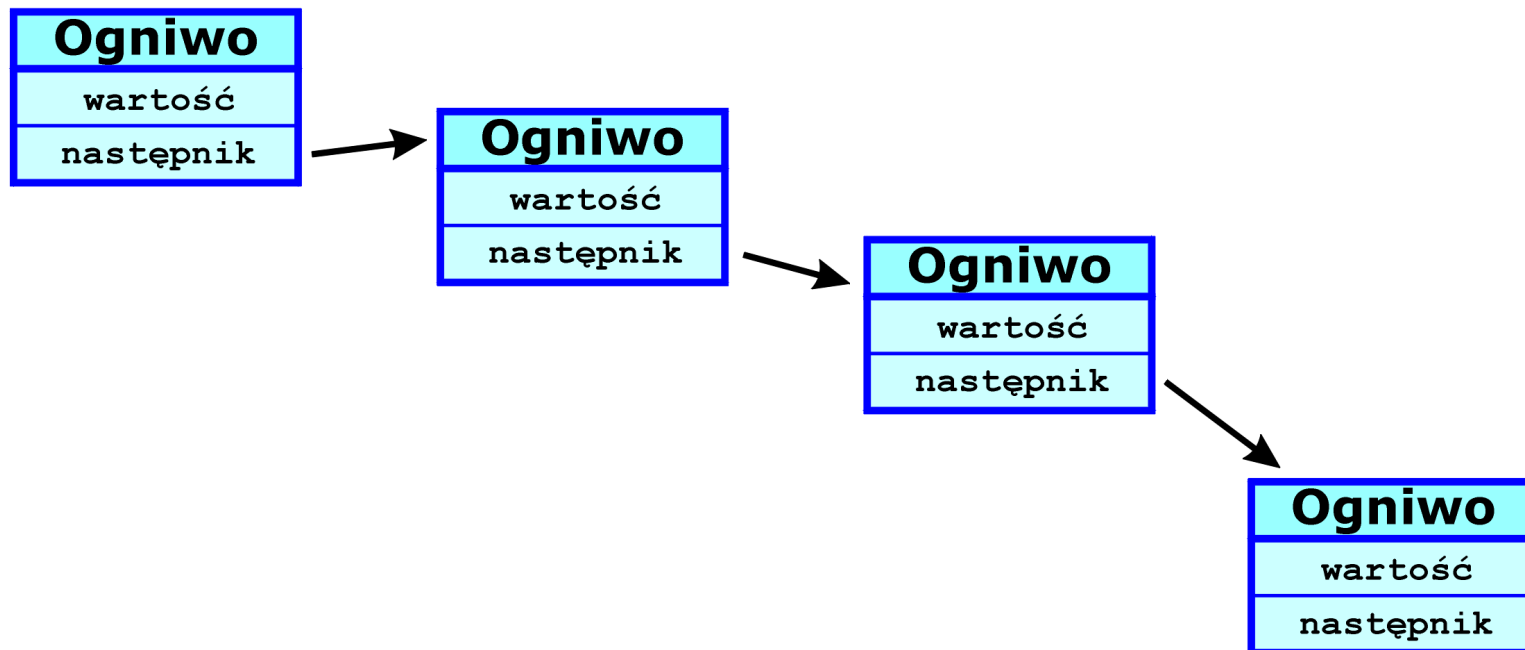
- przy tworzeniu obiektów na stercie. ■
- do tworzenia wiązanych struktur danych takich jak lista czy drzewo ■
- w celu umożliwienia funkcji zmiany wartości przekazanych jej argumentów. ■

■

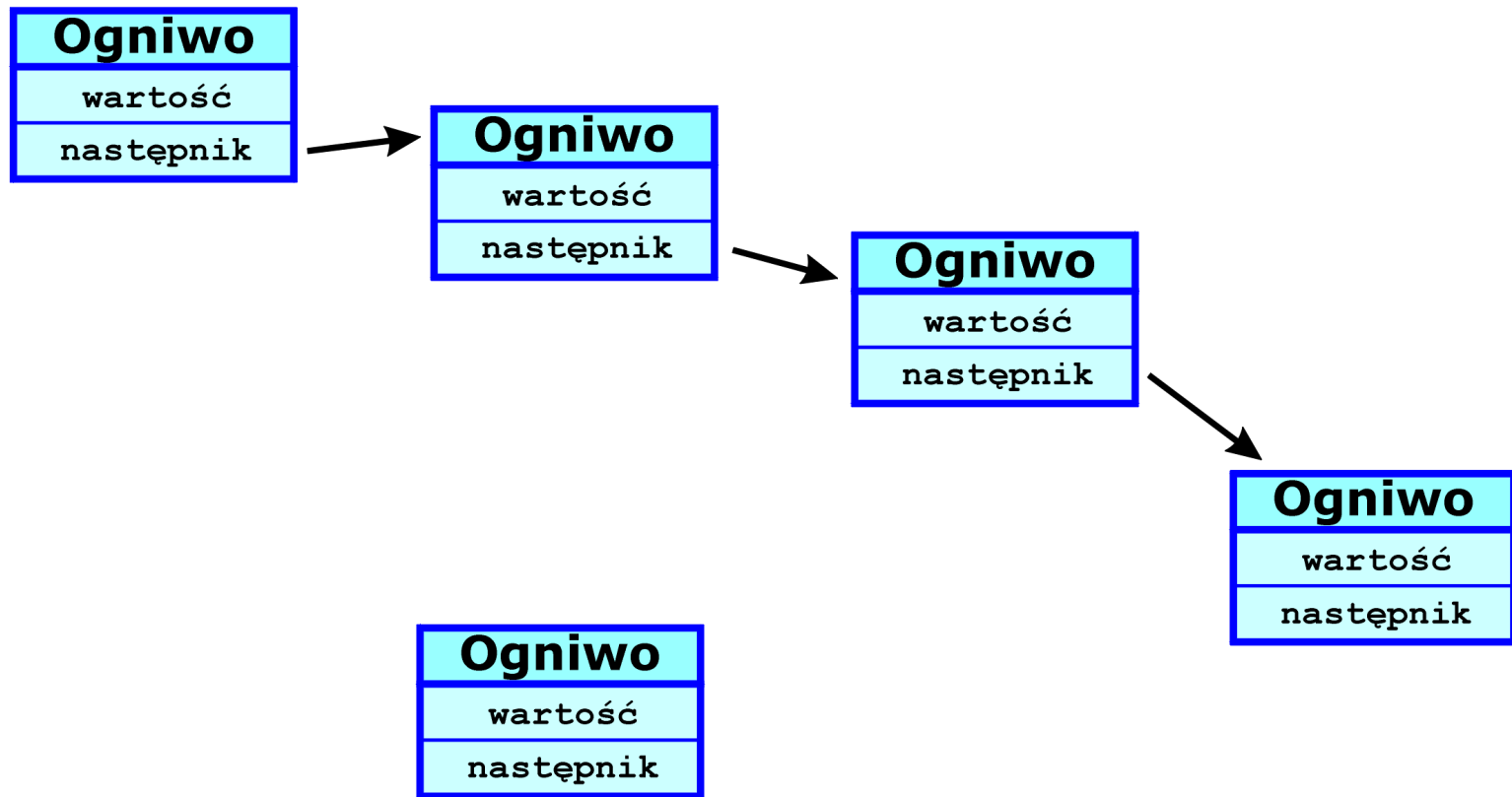
Ponadto wskaźniki umożliwiają:

- bezpośredni dostęp do pamięci komputera ■
- szybszy dostęp do elementów tablic ■

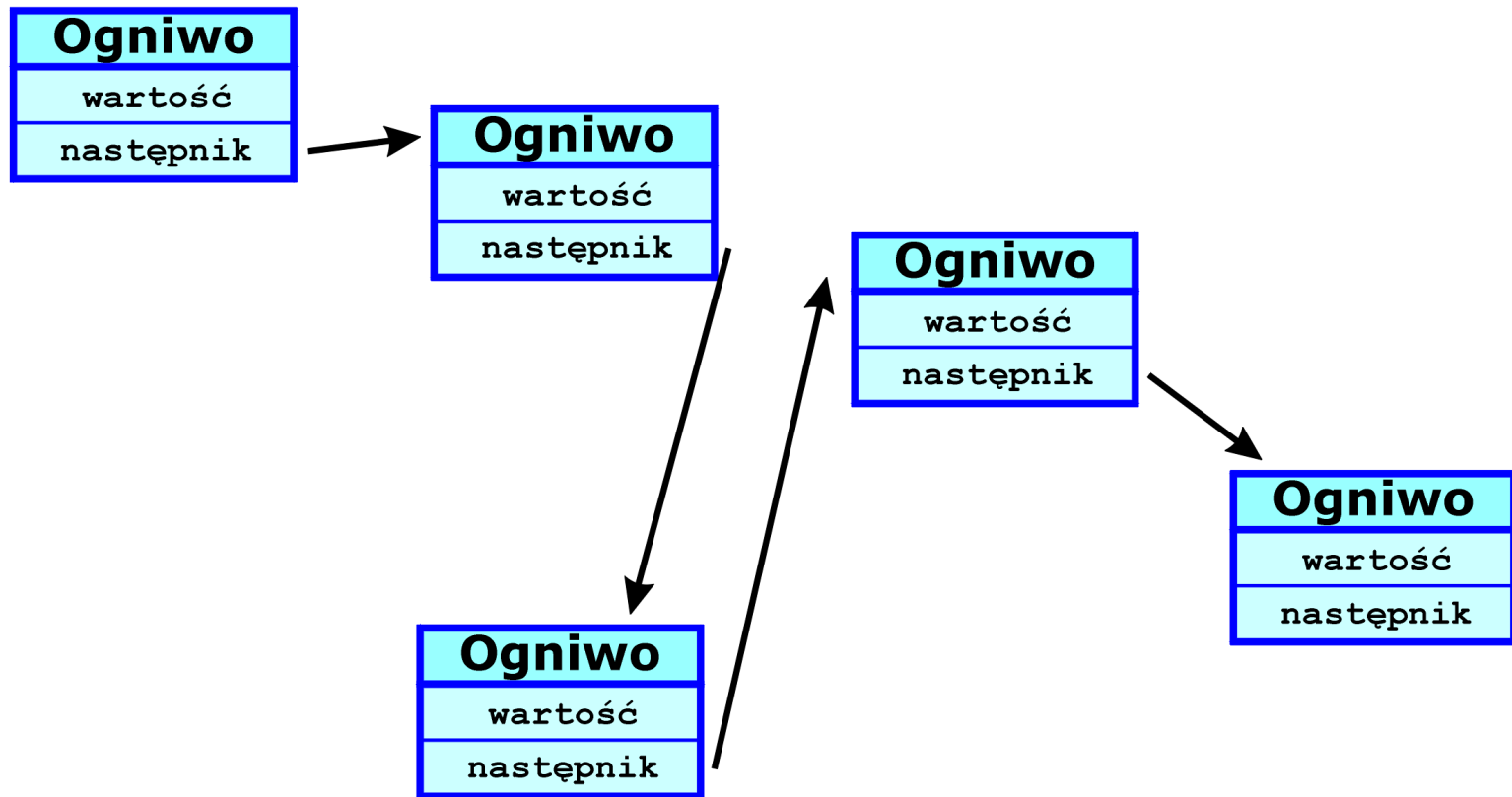
# Wiązane struktury danych 7



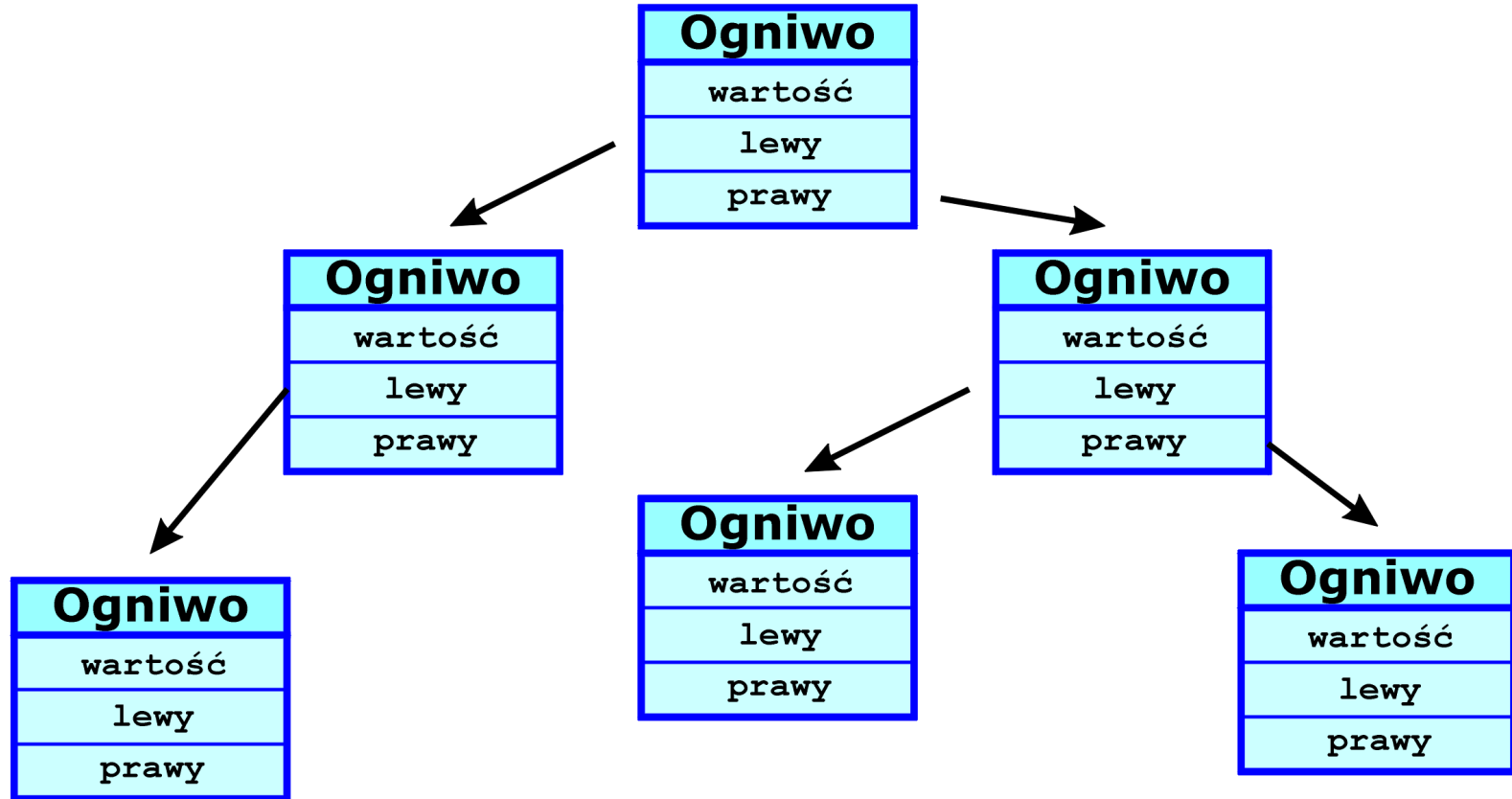
# Wiązane struktury danych 8



# Wiązane struktury danych 9



# Wiązane struktury danych 10



# Arytmetyka wskaźników <sup>11</sup>

- **arytmetyka wskaźników**: operacje dodawania do wskaźnika liczby, odejmowania i porównywania wskaźników ■
- **pożytek**: bezpośrednie operacje na adresie, co pozwala na tworzenie niskopoziomowego kodu ■
- **kompilator nie jest w stanie sprawdzić poprawności**: arytmetyka wskaźników niebezpieczna ■

- W kodzie generowanym przez kompilator referencja jest traktowana jako zmienna, w której przechowuje się adresy obiektów. ■
- Kod maszynowy wygenerowany w przypadku referencji jest taki sam jak w przypadku użytego w jej miejsce wskaźnika ■
- Na poziomie aseblera nie ma różnicy między referencją, a wskaźnikiem. ■
- Różnica pojawia się przy tłumaczeniu programu przez kompilator i sprowadza się do zastosowania interpretacji bliskiej w odniesieniu do wskaźników, a interpretacji dalekiej w odniesieniu do referencji. ■

|        | Referencja                                                                                                                                                  | Wskaźnik                                                                                                                                            |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Zalety | <ul style="list-style-type: none"><li>• nie wymaga operatora wyłuskania ■</li><li>• nie grozi zapisaniem niewłaściwego miejsca w pamięci ■</li></ul>        | <ul style="list-style-type: none"><li>• umożliwia arytmetykę wskaźników ■</li><li>• pozwala na wskazywanie na dowolne miejsce w pamięci ■</li></ul> |
| Wady   | <ul style="list-style-type: none"><li>• nie umożliwia arytmetyki wskaźników ■</li><li>• nie pozwala na wskazywanie na dowolne miejsce w pamięci ■</li></ul> | <ul style="list-style-type: none"><li>• wymaga operatora wyłuskania ■</li><li>• grozi zapisaniem niewłaściwego miejsca w pamięci ■</li></ul>        |



# Wskaźniki w miejsce referencji <sup>14</sup>

- referencję zawsze można zastąpić wskaźnikiem ■
- lepiej tego nie robić:
  - arytmetyka wskaźników niepotrzebna, ale przez pomyłkę może zostać użyta i to w sposób niebezpieczny ■
  - konieczny operator wyłuskania ■

Wskaźniki należy stosować tylko wtedy gdy jest to absolutnie konieczne do osiągnięcia wysokiej wydajności kodu, bo używanie wskaźników grozi wygenerowaniem trudnych do zlokalizowania błędów.

# Typy referencyjne <sup>15</sup>

- odnośniki w formie pośredniej — w zależności od kontekstu stosowana jest albo interpretacja bliska lub daleka ■
- terminologia: wskaźniki lub referencje, w zależności od tego, która interpretacja jest stosowana częściej. ■
- przykład: typy referencyjne ■
- typ referencyjny: odnośnik jest jedynym sposobem dostępu do obiektu ■
- zmienna typu referencyjnego: w rzeczywistości odnośnik, któremu zazwyczaj najbliżej do referencji ■
- Java:

```
Samochod pierwszy=new Samochod();
```

■

- dwa rodzaje spojrzenia:
  - niskopoziomowe: obiekt traktujemy jako odnośnik ■
  - wysokopoziomowe: obiekt, będący w istocie odnośnikiem, utożsamiamy z obiektem, na który ten odnośnik pokazuje ■
- 
- na codzień spojrzenie wysokopoziomowe ■
- subtelności w sposobie postępowania z typami referencyjnym łatwiej zrozumieć poprzez spojrzenie niskopoziomowe: zwrot **traktowany jako odnośnik**. ■

## Wartość null<sup>17</sup>

- wartość specjalna: odnośnik nie pokazuje na żaden obiekt ■
- w C++ jest to 0, w Javie i C# **null** ■
- Java i C#: wartość domyślna, gdy tworzymy obiekt typu referencyjnego i nie inicjalizujemy go ■
- Java, C#: instrukcja nie tworzy obiektu!

Samochod drugi;

■

# Porównanie interpretacji typów odnośnikowych w C++, C# i Java<sub>18</sub>

Podstawowe sytuacje, w których występują różnice w interpretacji odnośników zebrane są w poniższej tabeli.

|               | Wsk. | Wsk. | Typ ref.                     | Typ ref. | Ref. |
|---------------|------|------|------------------------------|----------|------|
| Język         | C++  | C#   | Java                         | C#       | C++  |
| inicjalizacja | B    | B    | B                            | B        | B    |
| =             | B    | B    | B                            | B        | D    |
| ==, !=        | B    | B    | B                            | D        | D    |
| <, >, <=, >=  | B    | B    | —                            | D        | D    |
| +, -, ++, --  | B    | B    | — (D dla + w <b>String</b> ) | D        | D    |
| .             | —    | —    | D                            | D        | D    |
| —>            | D    | D    | —                            | —        | —    |

B — interpretacja bliska, D — interpretacja daleka ■

# Programowanie: sterta

- zmienne lokalne, globalne i tablice to za mało:
  - wielkość zwykłej tablicy musi być znana w momencie kompilacji ■
  - czas życia obiektu nie pasuje ani do zmiennych globalnych ani do lokalnych ■
- potrzebne obiekty, które mogą być powoływać do życia i likwidowane w dowolnym czasie ■
- problem: jak przydzielać pamięć
  - zmienne globalne: źle, czas życia zmiennej to czas życia programu ■
  - zmienne lokalne: źle, czas życia od wejścia do bloku do wyjścia z bloku ■

**Sterna**: obszar pamięci przeznaczony dla zmiennych dowolnego typu, które mogą być w dowolnym momencie tworzone i likwidowane ■



# Tworzenie i likwidacja obiektów na stercie 4

- bezpośrednio przed utworzeniem obiektu przydzielenie pamięci ■
- przydział w miarę wolnego miejsca ■
- obiekt likwidowany zwalnia pamięć ■
- brak likwidacji prowadzi do przepełnienia sterty ■
- **wyciek pamięci**: błąd w programie polegający na stopniowym zapełnianiu, a wreszcie przepełnieniu pamięci sterty na skutek braku likwidacji niepotrzebnych obiektów ■
- **poszatkovanie sterty**: wolne miejsce sumarycznie duże, składa się z dużej ilości małych porcji ■

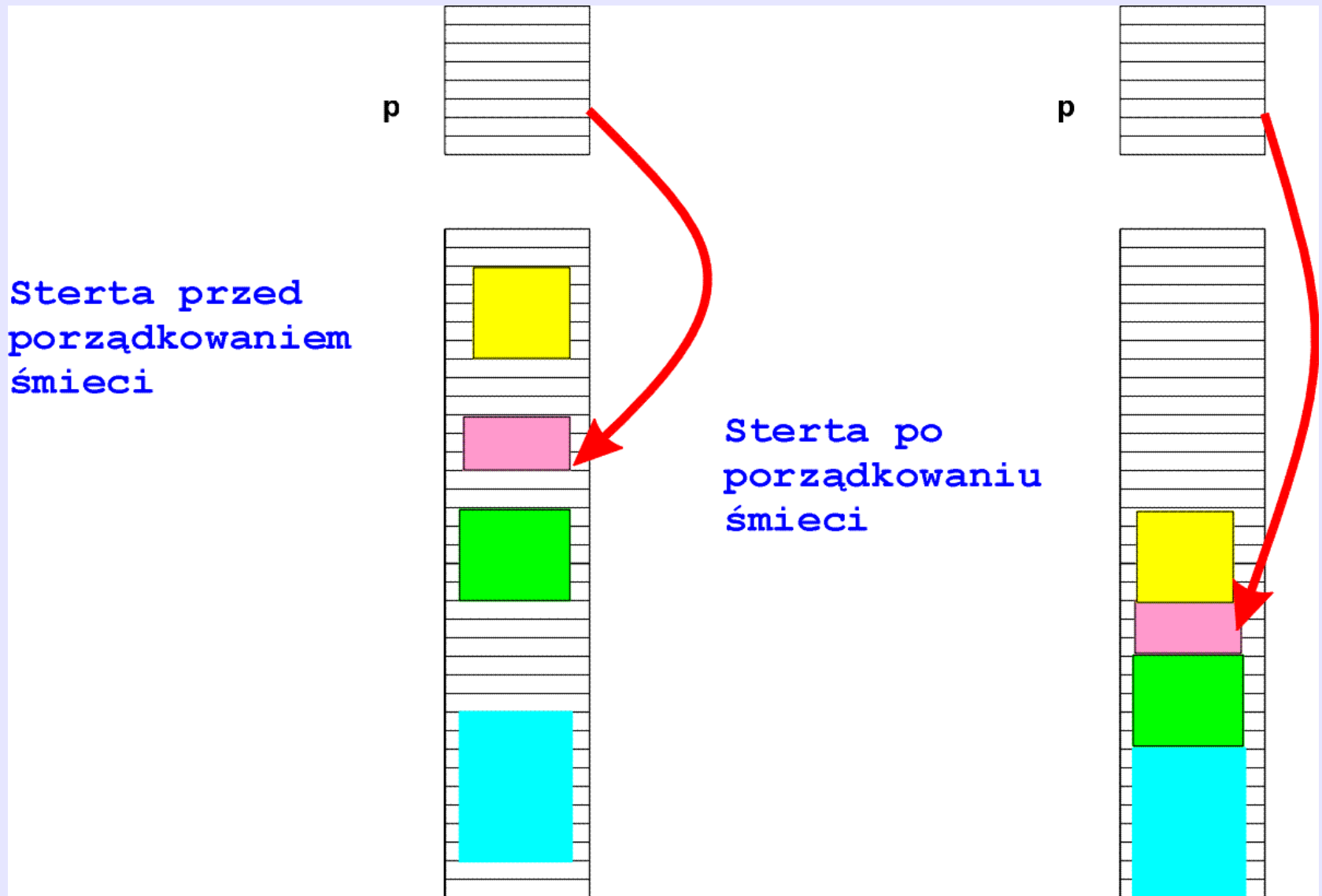
# Tworzenie i likwidacja obiektów na stercie 5

- Obiektom tworzonym na stercie nie nadaje się nazwy ■
- Dostęp do takich obiektów jest tylko poprzez odnośniki ■

# Sterna z automatycznym odśmiecaniem 6

- obiekt istnieje co najmniej tak długo jak długo istnieje choć jedna referencja do niego ■
- mechanizm obsługi steru sam decyduje kiedy należy zwolnić pamięć ■
- w sytuacji poszatkowania realokacja obiektów ■
- w konsekwencji konieczność modyfikacji wszystkich odnośników pokazujących na obiekty na steru ■
- proces określany mianem **garbage collection** (odśmiecanie) ■
- cena: obniżenie wydajności ■
- dostępna w C#, Javie, językach skryptowych, ale nie w C/C++ ■

# Sterta z automatycznym odśmiecaniem 7



# Jak może działać automatyczny odśmiecacz? <sup>8</sup>

Odśmiecacz potrzebuje pomysłu na

- odszukiwanie niepotrzebnych obiektów ■
- przeadresowywanie referencji ■

W tym celu stosowane są różne metody

- Zapamiętywanie referencji wskazujących na obiekt
  - jest powolna ■
  - nie wykrywa pętli w referencjach ■

■

- Analiza ścieżek dostępu wychodzących ze stosu i pamięci statycznej ■

6.

## C/C++ wskaźniki i referencje

# Wskaźniki<sub>3</sub>

- **Wskaźnik** to zmienna, która pokazuje na inną zmienną (przechowuje adres zmiennej)

- 

```
int* p;
```



- Wskaźnikom przypisujemy wartości poprzez operator &

```
int n;  
p=&n;
```





# Wskaźniki<sub>4</sub>

Wykorzystanie poprzez operator wyłuskania:

```
int n;  
p=&n;  
*p=12;  
cout << n;
```

- Wyrażenie postaci **\*p**, gdzie **p** jest wskaźnikiem pokazującym na zmienną **x** może być wykorzystane wszędzie tam gdzie można użyć **x**. ■
- W szczególności, **\*p** można użyć po lewej stronie operatora **=**.

**Ćwiczenie.** Napisz krótki program, który deklaruje dwa wskaźniki: jeden do zmiennej typu int, drugi do zmiennej typu float, a następnie przypisuje wskaźnikom wartości i wypisuje ich wartości (adresy) oraz wartości zmiennych, na które wskaźniki pokazują.

## Wskaźniki<sub>6</sub>

```
int main(){  
    int n=3;  
    int* p=&n;  
  
    (*p)=5;  
    cout << "n=" << n << ", *p=" << *p;  
    cout << ", p=" << p << "\n";  
  
    (*p)--;  
    cout << "n=" << n << ", *p=" << *p;  
    cout << ", p=" << p << "\n";  
}
```

n=5, \*p=5, p=0x22ff1c

n=4, \*p=4, p=0x22ff1c

# Deklaracja wskaźnika <sup>7</sup>

- formy równoważne

```
int* p;  
int * p;  
int *p;
```



- Pierwsza konwencja: uwypukla nowy typ
- Niestety nie działa, gdy chcemy zadeklarować więcej wskaźników w jednej deklaracji.

```
int* p, q;
```



- Trzeba napisać:

```
int *p, *q;
```

# Deklaracja wskaźnika <sup>8</sup>

- Zapis **int\*** ma jednak swoje zalety ■
- Najlepiej pisać **int\*** i używać oddzielnej deklaracji dla każdego wskaźnika

```
int* p;  
int* q;
```



- Można też użyć skrótu:

```
typedef int* IntPtr;
```



- Używając skrótu można zadeklarować kilka wskaźników w jednej deklaracji.

```
IntPtr p, q;
```

# Arytmetyka wskaźników<sup>9</sup>

```
int n;  
p= &n;  
p++;
```

```
int n[20], *p, *q;  
p=&n[3];  
q=&n[7];  
cout << q - p;
```

```
int main(){
    char c[]="programming";
    cout << "-";
    for(char* q=&c[0];q<=&c[10];++q){
        cout << *q << "-";
        *q-=32;
    }
    /* We test modification effect*/
    cout << "\n\n";
    for(char* q=&c[0];q<=&c[10];++q){
        cout << *q << "-";
    }
}
```



**Ćwiczenie.** Napisz program, który wczytuje ze standardowego wejścia 10 liczb całkowitych i zapisuje je w tablicy, a następnie przegląda tablicę wykorzystując wskaźniki w celu znalezienia największej liczby w tablicy.

```
int main(){
    /* We cannot assume, that the variables
       declared jointly share nearby places */

    int k=1,n=3,m=7;
    int* p=&n;

    cout << "k=" << k << "\n";
    cout << "n=" << n << "\n";
    cout << "m=" << m << "\n";
    cout << "*(p-1)=" << *(p-1) << "\n";
    cout << "*p=" << *p << "\n";
    cout << "*(p+1)=" << *(p+1) << "\n";
}
```

- Kod generowany przez kompilator nie sprawdza, czy wskaźnik rzeczywiście pokazuje na obiekt właściwego typu oraz czy wpisywanie w to miejsce jest sensowne i bezpieczne. ■
- Wskaźniki w C/C++ dają programiście pełną kontrolę nad całą pamięcią. Pozwala to na pisanie bardzo szybkich programów. Jednak znacząco zwiększa to ryzyko pomyłek skutkujących trudnymi do zlokalizowania błędami. ■

# Porównywanie wskaźników <sup>15</sup>

- Wskaźniki mogą być porównywane przy użyciu operatorów `==`, `!=`, `<`, `>`, `<=`, `>=` ■
- – wskaźniki są równe, gdy pokazują na ten sam obiekt ■
  - wskaźniki są różne, gdy pokazują na różne obiekty ■
  - Jeśli dwa wskaźniki pokazują na obiekty tej samej tablicy, to wskaźnik o mniejszej wartości pokazuje na element tablicy o mniejszym indeksie. ■
- konwencja: żaden obiekt nie jest przechowywany pod adresem zero. ■
- wskaźnik ustawiony na zero oznacza, że w danym momencie wskaźnik nie pokazuje na żaden obiekt. ■

# Tablica jako wskaźnik <sup>16</sup>

- Nazwa tablicy jest synonimem adresu elementu tablicy o indeksie zero
- 

```
int t[]={0,1,2,3,4,5};  
cout << *(t+3),
```



- Nazwa tablicy może być traktowana jako stały wskaźnik: jego wartość nie może być zmieniona
- notacja tablicowa jest możliwa przy użyciu dowolnego wskaźnika

```
int t[]={0,1,2,3,4,5};  
int* p=t+1;  
cout << p[2];
```



## Tablica jako wskaźnik 17

**Ćwiczenie.** Napisz program, który deklaruje tablicę **a** o pojemności 20 liczb typu **int** i przechowuje  $2i^2 + 1$  w elemencie tablicy o indeksie  $i$ . Wykorzystaj wskaźniki do przypisywania wartości elementom tablicy. Następnie zadeklaruj inny wskaźnik pokazujący na  $a[10]$ . Użyj go jako nazwy tablicy i, wykorzystując składnię tablicy napisz pętlę, która drukuje elementy  $a[10], a[11], \dots, a[14]$ .

## Tablica jako wskaźnik <sup>18</sup>

```
int main(){
    int t[20];

    // Use an array as a pointer
    for(int i=0; i<20; i++) *(t+i)=i*i;

    int* p=&t[0];

    // Use a pointer as an array
    for(int i=0; i<20; i++){
        cout << i << "^2=" << p[i] << "\n";
    }
}
```

# C/C++ Referencje <sup>19</sup>

- **Referencja** w C++ jest czymś w rodzaju wskaźnika, który udaje, że jest zmienną ■
- uproszczony wygląd: na referencję można patrzeć jak na dodatkową nazwę zmiennej.■
- Ale: czasami jest to jedyna nazwa obiektu (obiekty na stercie) ■
- 

```
int n;  
int& r=n;
```

■

- możliwa interpretacja **int&**: nowy typ, pochodny od typu **int** ■
- te same ograniczenia co przy wskaźnikach ■



- Referencja musi być zainicjalizowana w miejscu deklaracji i nie może być później zmieniona. ■
- Referencja może być użyta wszędzie tam gdzie zmienna. ■

**Ćwiczenie.** Napisz program, który deklaruje dwie referencje: jedną dla zmiennej typu **int**, a drugą dla typu **double**, przypisuje im wartości początkowe, a następnie wypisuje wartości zmiennych wskazywanych przez referencje. (via the references)

```
int main(){  
    int n=3;  
    int& r=n;  
  
    ++r;  
    cout << n << "\\n";  
  
}
```

# Wskaźniki i referencje, a modyfikator **const** 22

- **const** przed deklaracją:
  - wskaźnik tylko do odczytu:

```
float x;  
const float* wsk=&x;
```

■

- referencja tylko do odczytu:

```
float x;  
const float& ref=x;
```

■

■

# Wskaźniki i referencje, a modyfikator **const** 23

- modifier **const** po znaku \* ale przed nazwą wskaźnika:

```
float x;  
float* const wsk=&x;
```



- **stały wskaźnik** - zawsze pokazuje na ten sam obiekt ■
- efekt jest zbliżony do użycia referencji, ale w przeciwieństwie do referencji konieczny jest operator \* (operator wyłuskania) ■

Definition wskaźnika tylko do odczytu syntaktycznie różni się od stałego wskaźnika jedynie lokalizacją znaku \* w odniesieniu do **const**.

# Użycie wskaźników i referencji do tworzenia obiektów na stercie <sup>24</sup>

- operator **new**: do tworzenia obiektów na stercie ■

- 

```
double* q=new double;
```

■

- Aby użyć obiektu potrzebujemy operatora wyłuskania

```
*q=1.5;
```

■

- Pozbycie się obiektu na stercie, gdy nie jest już potrzebny:

```
delete q;
```

■

# Użycie wskaźników i referencji do tworzenia obiektów na stacku<sup>25</sup>

- To samo z referencjami:

```
double& q=*new double;  
q=1.5;  
....  
delete &q;
```




# Użycie wskaźników i referencji do tworzenia obiektów na stercie<sup>26</sup>

- stworzenie tablicy na stercie:

```
double* q=new double[n]  
q[5]=1.12;
```



- Wielkość tablicy na stercie nie musi być znana w chwili kompilacji! 
- Zwolnienie pamięci

```
delete[] q;
```



# Brak wolnej pamięci na sterencie <sup>27</sup>

- jeśli zabraknie wolnego miejsca na sterencie ...
  - C: zwracany adres zero ■
  - C++: rzucony tzw. wyjątek ■

■

**Ćwiczenie.** Napisz program, który metodą prób i błędów sprawdza jakie jest największe  $n$ , takie, że sarta pomieści tablicę typu **double** o  $2^n$  elementach. Pamiętaj o zwolnieniu pamięci przed ponowną jej rezerwacją.



5.

## C/C++ - funkcje

# Funkcje rekurencyjne <sub>3</sub>

- funkcja może wywoływać sama siebie: **wywołanie rekurencyjne** ■■
- wywołanie musi być warunkowe, w przeciwnym razie wywołania zapętliłyby się ■
- typowe zastosowanie: ciągi definiowane rekurencyjnie

$$0! := 1$$

$$n! := (n - 1)! * n \text{ dla } n \geq 1$$



# Silnia rekurencyjnie <sup>4</sup>

```
int factorial(int n){
    if(n<=1) return 1;
    else return factorial(n-1)*n;
}

int main(){
    for(int i=1;i<=10;++i){
        std::cout << "f[" << i << "]= " << factorial(i)
                    << std::endl;
    }
}
```

# Funkcje rekurencyjne <sub>5</sub>

**Ćwiczenie.** Napisz funkcję rekurencyjną, która oblicza  $n$ ty wyraz ciągu zdefiniowanego rekurencyjnie wzorami

$$\begin{aligned}c_0 &:= 1 \\ c_n &:= (1 + c_{n-1})^2 \text{ dla } n \geq 1\end{aligned}$$

# Funkcje rekurencyjne <sub>6</sub>

- Ciąg Fibonacciego

$$\begin{aligned} f_0 &:= 0, \quad f_1 := 1 \\ f_n &:= f_{n-1} + f_{n-2} \text{ dla } n > 1 \end{aligned}$$

# Fibonacci rekurencyjnie<sub>7</sub>

```
int fib(int n){  
    std::cout << "Evaluating fib for n=" << n  
                << std::endl;  
    if (n<=0) return 0;  
    if (n==1) return 1;  
    return fib(n-2)+fib(n-1);  
}
```

## Fibonacci iteracyjnie<sub>8</sub>

```
int fib(int n){  
    if(n<=0) return 0;  
    if(n==1) return 1;  
    int f=1,fp=0;  
    for(int i=2;i<=n;++i){  
        int t=fp+f;  
        fp=f;  
        f=t;  
    }  
    return f;  
}
```



# Zalety i wady wywołania rekurencyjnego 9

- Zaleta: prostota, łatwość analizy kodu ■
- Wada: jest wolniejsze i wymaga więcej pamięci ■

**Ćwiczenie.** Napisz funkcję, która przyjmuje jako argument liczbę naturalną  $n$  i drukuje wszystkie sposoby posortowania elementów zbioru  $\{1, 2, \dots, n\}$ .

# Argumenty funkcji `main` <sup>11</sup>

- Funkcja **main** może być wywołana z następującymi argumentami

```
int main(int argc, char* argv[])
```



- **argc**: liczba łańcuchów znakowych w linii wywołania programu z wliczeniem nazwy programu ■
- **argv**: tablica wskaźników do łańcuchów znakowych przeczytanych z linii komend w kolejności pojawienia się na tej liście ■

## Argumenty funkcji main<sub>12</sub>

```
int main(int argc, char* argv[]){
    cout << "\nLiczba argumentow linii komend to ";
    cout << argc << "\n\n";
    for(int i=0;i<argc;i++){
        cout << "Slowo " << i << " to: \n ";
        cout << argv[i] << "\n";
    }
}
```

## Argumenty funkcji main<sup>13</sup>

**Ćwiczenie.** Napisz program, który wypisuje wszystkie argumenty z linii komend, które rozpoczynają się od cyfry.

# Przekazywanie argumentów do to funkcji przez wartość <sup>14</sup>

```
int a_function(int i, double f){  
    i++;  
    f--;  
    cout << "Inside: i=" << i << ", f=" << f << "\n";  
    return 1;  
}
```

```
int main(){  
    int j=7;  
    double g=1.5;  
    cout << "Before: j=" << j << ", g=" << g << "\n";  
    a_function(j,g);  
    cout << "After: j=" << j << ", g=" << g << "\n";  
}
```

# Przekazywanie argumentów do to funkcji przez wartość <sup>15</sup>

- Wykonywane przez skopiowanie argumentu do zmiennej lokalnej ■
- Zaleta: chroni przed niezamierzoną modyfikacją argumentu ■
- Wada: kopiowanie dużych argumentów jest kosztowne czasowo i pamięciowo

# Przekazywanie argumentów do to funkcji przez referencję <sup>16</sup>

```
int a_function(const int& i, const double& f){  
    i++;  
    f--;  
    cout << "Wewnatrz: i=" << i << ", f=" << f << "\n";  
    return 1;  
}  
  
int main(int argc, char* argv[]){  
    int j=7;  
    double g=1.5;  
    cout << "Przed: j=" << j << ", g=" << g << "\n";  
    a_function(j,g);  
    cout << "Po: j=" << j << ", g=" << g << "\n";  
    return 0;  
}
```



# Przekazywanie argumentów do to funkcji przez referencję <sup>17</sup>

- Wykonywane przez skopiowanie adresu argumentu ■
- Zaleta: oszczędza czas i pamięć w przypadku dużych argumentów, bo kopiowany jest tylko adres.■
- Wada: nie chroni przed przypadkową modyfikacją chyba, że użyto **const**

# Przekazywanie argumentów do to funkcji przez referencję <sup>18</sup>

**Ćwiczenie.** Napisz funkcję, która przyjmuje dwa argumenty i wzajemnie zamienia ich wartości. Wytłumacz co stałoby się przy przekazywaniu argumentów przez wartość

# Przekazywanie argumentów do to funkcji przez wskaźnik <sup>19</sup>

```
int a_function(int* i, double* f){
    (*i)++;
    (*f)--;
    cout << "Inside: *i=" << *i << ", *f=" << *f << "\n";
    return 1;
}

int main()
{
    int j=7;
    double g=1.5;
    cout << "Before: j=" << j << ", g=" << g << "\n";
    a_function(&j,&g);
    cout << "After: j=" << j << ", g=" << g << "\n";
    return 0;
}
```

# Przekazywanie argumentów do to funkcji przez wskaźnik <sup>20</sup>

- Podobne do przekazywania argumentu przez referencję.■
- Wada: wymaga jawnego użycia operatora wyłuskania.

## Przekazywanie tablic do funkcji <sup>21</sup>

```
void f(int t[]) {  
    for(int i=0; i<5; i++){  
        t[i]++;  
        cout << "Element[" << i << "]= " << t[i] << endl;  
    }  
}  
  
int main(){  
    int x[5]={3,2,1,2,3};  
  
    f(x);  
    for(int i=0; i<5; i++){  
        cout << "x[" << i << "]= " << x[i] << endl;  
    }  
}
```

# Przekazywanie tablic do funkcji <sup>22</sup>

- Nie da się przekazać przez wartość ■
- Nagłówek funkcji

```
void f(int t[5])
```

uważany jest za identyczny z nagłówkiem

```
void f(int* t)
```

■

- W obu przypadkach do funkcji przekazywany jest adres początkowego elementu tablicy. ■
- Tak czy owak wielkość tablicy jest pomijana, więc trzeba sobie radzić samemu.

```
void f(int t[])
```

# Przekazywanie tablic do funkcji <sup>23</sup>

- Liczbę elementów przekazujemy jako oddzielny argument.

## Modyfikator **const** przy argumentach funkcji <sup>24</sup>

**const** użyty przy argumentach przekazanych przez wskaźnik lub referencję powoduje, że ten argument nie może być modyfikowany. ■

```
void f1(const int& i, const int* j) {  
    i++; // wrong, the compiler will report an error  
    *j+=5; // wrong, the compiler will report an error  
}
```



## Modyfikator **const** przy argumentach funkcji <sup>25</sup>

```
void f1(const int& i){  
    // wrong, the compiler will report an error:  
    // i++;  
    cout << "Argument value inside " << i << "\n";  
}  
  
void f2(const int* ptr){  
    // wrong, the compiler will report an error:  
    // (*ptr)*=2;  
    cout << "Argument value inside " << *ptr << "\n";  
}
```

# Modyfikator **const** przy argumentach funkcji <sup>26</sup>

- Użycie **const** musi być konsekwentne. ■
- W przeciwnym razie będzie problem z wywoływanymi funkcjami, które nie zmieniają argumentów, ale tego nie obiecują, co wywołuje konieczność kaskadowych zmian. ■

## Przetwarzanie funkcji <sup>27</sup>

```
void write(int i){  
    cout << "This is an int: " << i << "\n";  
}  
  
void write(float f){  
    cout << "This is a float: " << f << "\n";  
}  
  
void write(float* f){  
    cout << "This is a float: " << *f;  
    cout << " transfered by a pointer\n";  
}
```

# Pewne formalnie różne typy traktowane są jako jednakowe <sup>28</sup>

Identification:

- **T\*** oraz **T[]** ■
- **T** oraz **const T** ■
- **T** oraz **T&** ■

```
void f(int* x)
void f(int x[]);
```

```
void f(int x);
void f(const int x);
```

```
void f(int x);
void f(int& x);
```

Stosowane są następujące zasady:

- (1) Pełna zgodność typów argumentów lub zgodność na zasadzie utożsamiania typów (np. **int** i **const int**) ■
- (2) zgodność uzyskana przy użyciu konwersji typów zwiększających pojemność np. **bool** na **int** albo **float** na **double** ■
- (3) zgodność uzyskana przy użyciu innych standardowych konwersji np. **int** na **double** ■
- (4) zgodność uzyskana przy użyciu konwersji użytkownika ■

W przypadku gdy dwie lub więcej funkcji daje się dopasować na tym samym poziomie kompilator zgłasza błąd.

**Ćwiczenie.** Napisz trzy wersje funkcji max obliczającej maksimum z trzech liczb oddzielnie dla typów: **int**, **float** i **double**.

# Zwracanie referencji i wskaźników przez funkcję <sup>31</sup>



```
int& h(int& x) {  
    ...  
    return x;  
}
```

- W ten sposób zwrócić można wskaźniki i referencje otrzymane na liście argumentów lub wskaźniki/referencje obiektów utworzonych przez funkcję na stacku. ■

Nie wolno zwracać wskaźników i referencji do obiektów lokalnych!!!

## Zwracanie referencji i wskaźników przez funkcję <sup>32</sup>

**Ćwiczenie.** Napisz funkcję, która przyjmuje przez referencję dwa argumenty typu **int** i zwraca przez referencję argument, który ma większą wartość. Zdemostruj, że funkcji tej można użyć po lewej stronie argumentu przypisania. Wyjaśnij co dokładnie się wtedy dzieje.



## Argumenty domyślne <sup>33</sup>

```
int volume(int l, int w=2, int h=1);
```

Wywołanie

```
int obj=volume(7);
```

jest równoważne wywołaniu

```
int obj=volume(7,2,1);
```

# Argumenty domyślne <sup>34</sup>

- Efekt jest równoważny zdefiniowaniu kolekcji przeładowanych funkcji ■
- Definicje tych funkcji są tworzone automatycznie przez kompilator poprzez dodanie stosownych instrukcji inicjalizujących. ■

- Jeśli podaliśmy domyślną wartość argumentu, musimy też podać domyślną wartość dla wszystkich kolejnych argumentów. ■
- Jeśli pominiemy argument przy wywołaniu funkcji, musimy też pominąć kolejne argumenty.

## Funkcje **inline** 35

Poprzedzając nagłówek funkcji słowem kluczowym **inline** wyrażamy życzenie, by kompilator przepisał ciało funkcji w miejscu wywołania. ■

```
inline double Fahrenheit2Celsius(double f){  
    return (f - 32)/9 * 5;  
}
```

- jest to tylko sugestia dla kompilatora ■
- użyteczne tylko dla krótkich funkcji

## Wskaźniki do funkcji <sup>36</sup>

- Kompilator identyfikuje nazwę funkcji z jej adresem w pamięci. ■
- Używając tylko nazwy funkcji bez nawiasów i argumentów, odnosimy się do adresu funkcji ■
- Aby wywołać funkcję potrzebujemy nawiasów() ■
- Adres można przechowywać w specjalnym wskaźniku ■

Wskaźnik do funkcji deklarujemy pisząc

```
int (*fptr)(int);
```

Do wskaźnika do funkcji możemy przypisać tylko nazwę funkcji której lista argumentów i typ zwracanego wyniku są zgodne z typem wskaźnika.

```
int f(int i){  
    cout << "This is function f\n";  
    return i+1;  
}  
int g(int i){  
    cout << "This is function g\n";  
    return i+2;  
}
```

## Wskaźniki do funkcji <sup>39</sup>

```
int main(){
    int (*fptr)(int)=f;
    cout << (unsigned int)f << endl;
    cout << (unsigned int)fptr << endl;
    cout << (unsigned int)*fptr << endl;
    int i=3;
    // We call the function indicated by the pointer,
    // that is function f
    cout << fptr(i) << endl;
    cout << (*fptr)(i) << endl;
    // We set the pointer on function g
    // and we repeat the call
    fptr=g;
    cout << (unsigned int)fptr << endl;
    cout << fptr(i) << endl;
}
```

# Deklarowanie funkcji <sup>40</sup>

Jeśli chcemy umieścić funkcję w osobnym pliku, aby skompilować ją oddzielnie od programu głównego, musimy powiedzieć programowi głównemu, jaki jest nagłówek funkcji. Robimy to, pisząc deklarację funkcji. Składa się ona z nagłówka funkcji (typ zwracany, nazwa i lista argumentów ujęte w nawiasy), a następnie średnika zamiast treści funkcji.

Nagłówkiem funkcji

```
double square(double x){  
    return x*x;  
}
```

jest

```
double square(double x);
```



1

4.

## C/C++ preprocesor

# Preprocesor<sub>3</sub>

**Preprocesor**: służy do wstępnego przetwarzania pliku źródłowego programu ■

Dyrektywy preprocesora są przetwarzane przed uruchomieniem kompilatora.

Składnia:

*#dyrektywa argumenty*

- brak średnika! ■
- ukośnik potrzebny do kontynuacji linii ■
- szeroko używany w C ■
- w C++ jego rola jest ograniczona. Wiele konstrukcji używanych w C w C++ jest traktowanych jako zaszłość ■

# Makrodefinicje <sup>4</sup>

syntax:

```
#define nazwa definicja
```

```
#define nazwa (argumenty_formalne) definicja
```

- name: dowolny identyfikator
- zwyczajowo: wielkie litery
- semantyka: wywołanie zostaje zastąpione definicją
- w przypadku argumentów: każde wystąpienie nazwy argumentu w definicji jest zastępowane tekstem podanym w wywołaniu makrodefinicji

# Makrodefinicje <sub>5</sub>

- W C++ w większości przypadków makra mogą i powinny być zastąpione przez bardziej sprytne **szablony**. ■
- Szablony są podobne do makr, ale są przetwarzane bezpośrednio przez kompilator, a nie przez preprocesor. ■
- W ten sposób można uniknąć wielu błędów spowodowanych brakiem właściwej współpracy między preprocesorem a kompilatorem.

# Zastąpienie argumentu łańcuchem znaków <sub>6</sub>

- literał łańcuchowy jest wstawiany w miejsce argumentu poprzedzonego # ■
- makro

```
#define SHOW(x) cout << #x << "=" << x << "\n" ;  
int i=7;  
SHOW(i)
```

efekt na standardowym wyjściu

```
i=7
```



**Ćwiczenie.** Napisz makro, którego można użyć do zadeklarowania zmiennej i jednocześnie wydrukować informację na standardowym wyjściu, że zmienna jest dekladowana.

# Klejenie identyfikatorów 7

- Podwójny krzyżyk `##` jest konwertowany na pusty tekst ■
- zastosowanie: tworzenie identyfikatorów poprzez klejenie nazw ■
- przykład: zadeklarowanie zestawu powiązanych zmiennych

```
#define TEXT(a) char *a##_pointer; int a##_length;  
TEXT (alpha);  
TEXT (beta);  
TEXT (gamma);
```

# Klejenie identyfikatorów<sub>8</sub>

- preprocesor nie sprawdza, czy argumenty są sensowne:

```
char* "alpha"_pointer; int "alpha"_length
```



**Ćwiczenie.** Napisz makro, które deklaruje dwie powiązane zmienne: tablicę i liczbę całkowitą do przechowywania liczby elementów w tablicy.



# Zastosowanie makrodefinicji w C++<sup>9</sup>

- aby zdefiniować puste nazwy do sterowania kompilatorem

```
#define INTEL
```

- dyrektywy kompilacji warunkowej

```
#if warunek  
    // pomija kompilację tego fragmentu, jeżeli warunek  
    nie jest spełniony  
#endif
```

- warunek: decyzja musi być możliwa w czasie kompilacji ■

# Dyrektywy kompilacji warunkowej<sup>10</sup>

wariant z **else**

```
#if warunek  
    // linie kompilowane gdy warunek zachodzi  
#else  
    // linie kompilowane gdy warunek nie zachodzi  
#endif
```

# Testowanie obecności i usuwanie nazw makrodefinicji <sup>11</sup>

Można sprawdzić, czy makrodefinicja jest zdefiniowana przez

```
defined(nazwa)
```



Makrodefinicja może zostać usunięta przez

```
#undef nazwa
```

# Testowanie obecności i usuwanie nazw makrodefinicji <sup>12</sup>

Czasami chcemy skompilować część programu tylko wtedy, gdy nazwa nie jest zdefiniowana. Następnie możemy użyć dyrektywy **#if** w formularzu

```
#if !defined (nazwa)  
    . . . .  
#endif
```

albo możemy użyć dyrektywy **#ifndef**

```
#ifndef nazwa  
    . . . .  
#endif
```

# Przerwanie kompilacji <sup>13</sup>

Poniższa dyrektywa może być użyta do przerywania kompilacji i wyświetlenia komunikatu o błędzie

```
#error tekst
```

**Ćwiczenie.** Napisz program, który przerywa kompilację, jeśli zdefiniowane jest makro o nazwie BORLAND

```
//#define _BORLAND
int main(){
    int i=7,j=14,k=3;

    #if defined(_BORLAND)
        #error keyword bitand does not work with Borland C++
        k=i & j;
    #else
        k=i bitand j;
    #endif
    cout << "k=" << k << "\n";
}
```

# Dyrektywa **include** 15

Dyrektywa **#include** pozwala na włączenie do kompilowanego pliku zawartości innego pliku. ■

syntax

```
#include <file_name>
```

```
#include "file_name"
```

■

- te dwie formy różnią się kolejnością przeszukiwania pliku przez foldery ■
- Konkretna kolejność zależy od kompilatora ■

# Jednokrotne włączenie pliku <sup>16</sup>

- złożony zestaw relacji między dołączonymi plikami: jeden plik może zostać błędnie dołączony więcej niż raz podczas jednej kompilacji ■
- to może prowadzić do błędów ■
- trik

```
#if !defined(_FILE_)  
#define _FILE_  
    // zawartość pliku  
#endif
```



## Predefiniowane nazwy <sup>17</sup>

- `_ _ LINE _ _` — Numer linii, przy której aktualnie działa preprocesor ■
  - `_ _ NAME _ _` — nazwa aktualnie kompilowanego pliku ■
  - `_ _ DATE _ _` — data kompilacji ■
  - `_ _ TIME _ _` — czas kompilacji ■
- 

**Ćwiczenie.** Napisz program, który wypisze na standardowym wejściu datę i godzinę jego skompilowania.

**Ćwiczenie.** Napisz makro, które deklaruje zmienną i wypisuje linię, w której zmienna została zadeklarowana.