

PROGRAMOWANIE I

Podstawy programowania w C++

Marek Brancewicz

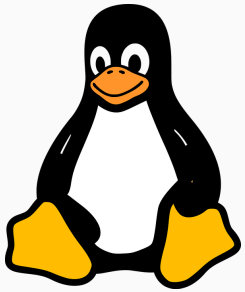
30 października 2026

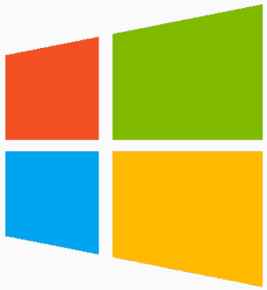
Wydział Fizyki
Uniwersytet w Białymstoku

Wykład 1

Wprowadzenie

1. Organizacja zajęć.
2. Narzędzia, materiały i literatura.
3. Zasady korzystania z AI.
4. Zasady oceniania.
5. Podstawowe pojęcia.
6. Oprogramowanie (instalacja i konfiguracja środowiska).
7. Praca w terminalu Bash (Linux) lub PowerShell (Windows).







Organizacja zajęć

1. Wprowadzenie. Organizacja zajęć. Narzędzia, materiały i literatura. Zasady korzystania z AI. Zasady oceniania. Podstawowe pojęcia. Oprogramowanie. Instalacja i konfiguracja środowiska. Praca w terminalu Linux'a (lub wierszu poleceń Windows).
2. Pierwszy program. Podstawowe typy zmiennych. Wczytywanie danych z klawiatury i wypisywanie ich na ekranie (strumień wejścia-wyjścia). Instrukcja warunkowa „if".
3. Pętle (for, while, do ... while). Liczby pseudolosowe. Instrukcja wielokrotnego wyboru „switch-case". Pomiar czasu wykonywania programu.
4. Tablice. Zapisywanie do i czytanie z pliku tekstowego. Złożone typy danych 1 (string).
5. Funkcje (definicja, struktura, przykłady, przekazywanie tablic do funkcji, wartości domyślne parametrów funkcji, przeciążanie funkcji, sufiksy danych, przesłanianie zmiennych). Rekurencja (rekursja).
6. Wskaźniki. Pojęcie stosu i sterty (stack i heap). Dynamiczna alokacja pamięci. Tablice alokowane dynamicznie. Arytmetyka wskaźników (inkrementacja i dekrementacja). Dynamiczna alokacja tablic wielowymiarowych.
7. Złożone typy danych 2 (struktura).
8. Ćwiczenia programistyczne.
9. Zaliczenie

Narzędzia wspomagające, materiały, literatura

[1] Asystenci AI oparci na LLM (Claude, Copilot, ChatGPT, Gemini...)

[2] Kursy on-line, np.: https://miroslawzelent.pl/kurs-c++/

[3] W. Porębski, *Język C++: wprowadzenie do programowania*, Komputerowa Oficyna Wydawnicza "Help", Warszawa 1999, Wyd. 2

[4] J. Grębosz, *Symfonia C++ standard: programowanie w języku C++ orientowane obiektowo*, Wydawnictwo "Edition 2000", Oficyna Kallimach, Kraków 2005

[5] S. Prata, *Język C++*, Wydawnictwo Helion, Gliwice 2006, Wyd. 5

[6] A. Koenig and B.E. Moo, *Accelerated C++: Practical Programming by Example*, Addison-Wesley, Boston 2013, 22nd printing

Zasady korzystania z AI









- **Korzystanie z AI jest dozwolone** podczas pracy na zajęciach oraz w domu.
- **AI jest pomocą w nauce, a nie wykonawcą zadań.** Używaj go do:
  - wyjaśniania pojęć i konstrukcji języka C++,
  - tłumaczenia komunikatów kompilatora i pomocy w znalezieniu błędu,
  - uzyskania wskazówki, gdy utkniesz (np. *„Daj mi podpowiedź, ale nie podawaj rozwiązania”*),
  - omówienia Twojego kodu i optymalizacji (np. *„Dlaczego to działa?”*, *„Co można tu napisać prościej?”*).
- **Najpierw spróbuj samodzielnie.** Zanim zapytasz AI, przemyśl problem, napisz choćby częściowe rozwiązanie i sprawdź, co dokładnie nie działa.

Zasady korzystania z AI

- **Musisz rozumieć każdą linię oddawanego kodu.** Prowadzący może poprosić o wyjaśnienie lub zmodyfikowanie dowolnego fragmentu. Brak zrozumienia oznacza, że rozwiązanie nie jest Twoje.
- **Nie kopiuj bezrefleksyjnie.** Jeśli AI pokazało Ci rozwiązanie, zamknij je i napisz kod od nowa samodzielnie.
- **Weryfikuj odpowiedzi AI.** Modele językowe popełniają błędy i potrafią podawać je z dużą pewnością. Kompiluj, testuj na własnych przykładach, sprawdzaj w dokumentacji (`np.cppreference.com`).
- **Trzymaj się materiału z zajęć.** AI często proponuje konstrukcje, których jeszcze nie omawialiśmy. Jeśli ich nie rozumiesz, poproś o rozwiązanie z użyciem poznanych elementów języka.
- **Bądź transparentny.** Jeśli AI istotnie pomogło Ci w zadaniu, zaznacz to krótkim komentarzem w kodzie (w czym pomogło).

Zasady korzystania z AI

- **Nie udostępniaj danych wrażliwych.** Nie wklejaj do narzędzi AI danych osobowych, haseł ani materiałów, których nie wolno rozpowszechniać.
- **Na zaliczeniu korzystanie z AI jest zabronione.** Dlatego regularnie ćwicz pisanie programów bez pomocy AI - umiejętność, której nie wytrenujesz, nie pojawi się nagle na zaliczeniu.

Podstawowe pojęcia

Podstawowe pojęcia

Algorytm skończony ciąg jednoznacznie określonych kroków, który prowadzi od danych wejściowych do wyniku.„Jednoznacznie” oznacza, że wykonawca (człowiek lub komputer) nie ma wątpliwości, co zrobić w danym kroku. Przykłady z życia: przepis kulinarny, dzielenie pisemne.

Pseudokod i schemat blokowy sposoby zapisu algorytmu niezależne od języka programowania. Pseudokod opisuje kroki „prawie po ludzku”, a schemat blokowy przedstawia je jako rysunek z bloków połączonych strzałkami.

Podstawowe pojęcia

Język programowania sztuczny, ściśle określony język do zapisywania instrukcji dla komputera. Składa się z: *słów kluczowych i symboli* (np. `int`, `if`, `;`), *składni*, czyli reguł poprawnego łączenia ich w zapisy („gramatyka"), oraz *semantyki*, czyli znaczenia zapisu: co komputer ma zrobić.

Język wysokiego i niskiego poziomu język wysokiego poziomu jest bliski sposobowi myślenia człowieka (np. C++, Python). Język niskiego poziomu jest bliski temu, jak działa procesor (asembler, kod maszynowy). C++ jest językiem wysokiego poziomu, ale pozwala też na operacje bliskie sprzętowi.

Podstawowe pojęcia

Kod źródłowy tekst programu napisany przez programistę w języku programowania i zapisany w zwykłym pliku tekstowym (w C++ zwykle `.cpp`, a pliki nagłówkowe `.h` lub `.hpp`). Jest czytelny dla człowieka, ale procesor nie potrafi go wykonać bezpośrednio.

Program ciąg instrukcji wykonywanych przez komputer w celu realizacji określonego zadania. Program zwykle pobiera dane wejściowe, przetwarza je i wyprowadza wynik. Słowem „program" nazywa się zarówno kod źródłowy, jak i jego gotową do uruchomienia postać (plik wykonywalny).

Podstawowe pojęcia

Język maszynowy (kod maszynowy) instrukcje zapisane jako liczby binarne, które procesor wykonuje bezpośrednio. Każda rodzina procesorów (np. x86-64, ARM) ma własny język maszynowy. Dlatego program zbudowany dla jednego typu procesora i systemu operacyjnego zwykle nie działa na innym.

Polecenie tekst wpisany w terminalu i zatwierdzony klawiszem Enter. Składa się z nazwy programu do uruchomienia oraz argumentów, np. w `g++ main.cpp -o main` nazwą jest `g++`, a `main.cpp`, `-o` i `main` to argumenty. Argumenty zaczynające się od myślnika nazywa się opcjami (flagami).

Podstawowe pojęcia

Katalog bieżący i ścieżka terminal zawsze „znajduje się” w jakimś katalogu (folderze), tzw. katalogu bieżącym. Polecenia szukają plików właśnie tam. Ścieżka to adres pliku lub katalogu, np. `/home/student/zad1/main.cpp`. Podstawowe polecenia: `cd` (zmiana katalogu), `ls` (Linux/macOS) lub `dir` (Windows) do wyświetlenia zawartości katalogu.

Budowanie programu (ang. *build*) zamiana kodu źródłowego w plik wykonywalny. W C++ składa się z trzech etapów: preprocesor $\rightarrow$ kompilator $\rightarrow$ linker. W terminalu wszystkie etapy wykonuje jedno polecenie:

```
g++ -Wall -Wextra -std=c++17 main.cpp -o main
```

Podstawowe pojęcia

Opcja `-o main` nadaje nazwę plikowi wykonywalnemu, `-Wall -Wextra` włączają ostrzeżenia, a `-std=c++17` wybiera wersję języka.

Preprocesor pierwszy etap budowania. Wykonuje dyrektywy zaczynające się od znaku #. Na przykład `#include <iostream>` wstawia w tym miejscu zawartość pliku nagłówkowego `iostream`.

Kompilator program, który tłumaczy kod źródłowy na kod maszynowy, zapisywany w tzw. pliku obiektowym. Przy okazji sprawdza poprawność kodu. Jeśli znajdzie błąd, wypisuje komunikat i nie tworzy wyniku. Popularne kompilatory C++: GCC (g++), Clang, MSVC.

Podstawowe pojęcia

Linker (konsolidator) program, który łączy pliki obiektowe i potrzebne biblioteki w jeden plik wykonywalny.

Plik wykonywalny wynik budowania: plik z kodem maszynowym, który można uruchomić. Z terminala uruchamiamy go poleceniem `./main` (Linux/macOS) lub `main.exe` (Windows).

Biblioteka standardowa zbiór gotowych narzędzi dostarczanych razem z każdym kompilatorem C++, np. do wypisywania tekstu na ekran (`std::cout`) czy pracy z napisami. Dzięki niej nie trzeba pisać wszystkiego od zera.

Podstawowe pojęcia

Interpreter program, który czyta kod źródłowy i od razu go wykonuje, instrukcja po instrukcji, bez tworzenia pliku wykonywalnego (np. Python). Zaletą jest możliwość natychmiastowego sprawdzenia zmian, wadą zwykle wolniejsze działanie programu. C++ jest językiem kompilowanym. (W praktyce wiele języków łączy oba podejścia.)

Rodzaje błędów *błąd kompilacji* to naruszenie reguł języka, np. brak średnika: program w ogóle się nie zbuduje; *błąd wykonania* pojawia się, gdy program się uruchamia, ale w trakcie działania kończy się awarią, np. przy dzieleniu liczby całkowitej przez zero; *błąd logiczny* występuje, gdy program działa, ale daje zły wynik. Błędy logiczne są najtrudniejsze, bo komputer ich nie zgłosi.

Podstawowe pojęcia

Ostrzeżenie (ang. *warning*) komunikat kompilatora, że kod jest poprawny, ale podejrzany. Program się zbuduje, ale ostrzeżeń nie należy ignorować, bo często wskazują na błąd logiczny, np. `if (x = 5)` zamiast `if (x == 5)`.

Debugger narzędzie do uruchamiania programu „pod kontrolą”. Pozwala zatrzymać program w wybranym miejscu (punkt przerwania, ang. *breakpoint*), wykonywać go krok po kroku i podglądać wartości zmiennych. Służy do wyszukiwania błędów (ang. *bugs*), a sam proces nazywa się debugowaniem.

Podstawowe pojęcia

Zintegrowane środowisko programistyczne (IDE, ang. *integrated development environment*) program, który zbiera w jednym miejscu narzędzia programisty: edytor kodu z podświetlaniem składni i podpowiedziami, budowanie i uruchamianie programu jednym przyciskiem oraz debugger. IDE zwykle *korzysta* z osobno zainstalowanego kompilatora i samo nim nie jest. Przykłady: Visual Studio, CLion, Code::Blocks. Na zajęciach pracujemy w terminalu, żeby widzieć, co IDE robi za nas „pod spodem".

Podstawowe pojęcia

Terminal (konsola) program (okno) do tekstowej komunikacji z komputerem: wyświetla tekst i przekazuje polecenia wpisywane z klawiatury powłoce, która je wykonuje (np. Windows Terminal, w Linuksie GNOME Terminal lub Konsole). Programy, które napiszemy na początku kursu, również działają w terminalu: tam wypisują wyniki i tam wczytują dane.

Powłoka program interpretujący i wykonujący polecenia wpisywane w terminalu (np. PowerShell w Windows, Bash w Linuksie).

Oprogramowanie











MinGW-w64 zestaw narzędzi GNU dla Windows (kompilator GCC, debugger GDB, Make); sam nie jest edytorem, potrzebuje edytora lub IDE

Notepad++ darmowy, lekki edytor tekstu dla Windows z podświetlaniem składni; nie kompiluje kodu, wymaga osobnego kompilatora (np. MinGW-w64)

Code::Blocks darmowe, otwartoźródłowe IDE dla C/C++; wersja instalacyjna dla Windows może zawierać MinGW

CLion zaawansowane IDE firmy JetBrains, oparte na CMake; bezpłatne do użytku niekomercyjnego i edukacyjnego

Visual Studio rozbudowane IDE firmy Microsoft z własnym kompilatorem MSVC; wersja Community jest bezpłatna

Praca w terminalu Linux - powłoka Bash

Praca w terminalu Linux - powłoka Bash

Bash (*Bourne Again SHell*) powłoka, także język skryptowy: ciąg poleceń można zapisać w pliku i uruchamiać jak program

Podstawowe polecenia

`pwd`	wyświetla ścieżkę bieżącego katalogu
`ls`	wyświetla zawartość bieżącego katalogu
`cd katalog`	przejście do podkatalogu `katalog`
`cd ..`	przejście do katalogu nadrzędnego
`cd`	powrót do katalogu domowego
`g++ --version`	wersja kompilatora (czy jest zainstalowany)

Linux rozróżnia wielkość liter: `Program.cpp` $\neq$ `program.cpp`

Czy mamy kompilator g++?

`g++ --version`	wyświetla wersję - kompilator jest zainstalowany
`which g++`	pokazuje, gdzie znajduje się program

Komunikat o braku polecenia (np. `command not found`) - trzeba go zainstalować:

Ubuntu, Debian	`sudo apt update`
	`sudo apt install build-essential`
Fedora	`sudo dnf install gcc-c++`
Arch	`sudo pacman -S gcc`

`sudo` - wykonanie polecenia z uprawnieniami administratora (wymaga hasła)

Kompilacja i uruchamianie programu

`g++ program.cpp`	tworzy plik `a.out`
`./a.out`	uruchamia go
`g++ program.cpp -o program`	tworzy plik `program`
`./program`	uruchamia go

Kompilacja z ostrzeżeniami:

```
g++ -Wall -Wextra program.cpp -o program
```

`./` oznacza bieżący katalog – powłoka nie szuka w nim programów automatycznie, dlatego sama nazwa (`program`) nie wystarczy.

Skrypt Basha

Na przykład plik o nazwie: `compandrun.sh`:

```
#!/bin/bash
echo "--- Compiling program ---"
if g++ main.cpp -o main; then
 echo "--- Compilation successful ---"
 echo "--- Running program ---"
./main
else
 echo "--- Compilation failed! ---" >&2
 exit 1
fi
```

Skrypt Basha

`if g++ main.cpp -o main; then` sprawdza, czy kompilacja zakończyła się sukcesem, jeśli kompilator zwróci kod 0, skrypt przechodzi do uruchomienia programu.

`else` jeśli kompilacja się nie powiedzie, wyświetla komunikat o błędzie.

`exit 1` kończy skrypt z kodem błędu 1.

`>&2` kieruje komunikat o błędzie na standardowe wyjście diagnostyczne (`stderr`).

Uruchamianie skryptu Basha

`#!/bin/bash`	*shebang* – wskazuje interpreter skryptu
`set -e`	przerywa skrypt przy pierwszym błędzie
`chmod +x compandrun.sh`	nadaje prawo wykonywania (`+x`)
`./compandrunrun.sh`	uruchamia skrypt
`bash compandrun.sh`	uruchamia skrypt bez `chmod`

Praca w Terminalu Windows - powłoka PowerShell

Terminal i powłoki w Windows

Terminal	w Windows 11 domyślnie **Windows Terminal**; w kartach można otwierać różne powłoki
cmd.exe	Wiersz polecenia - najstarsza i najprostsza powłoka; skrypty `.bat`, `.cmd`
PowerShell 5.1	`powershell.exe` – wbudowany w Windows, utrzymywany, ale już nierozwijany
PowerShell 7	`pwsh` - nowsza wersja, instalowana osobno, działa też w Linuksie i macOS

PowerShell przekazuje między poleceniami **obiekty**, a nie sam tekst jak Bash; skrypty mają rozszerzenie `.ps1`.

Bash w Windows

Git Bash instalowany razem z Gitem dla Windows; Bash z podstawowymi narzędziami uniksowymi (`ls`, `grep`, ...)

WSL *Windows Subsystem for Linux* - pełny system Linux uruchamiany w Windows, z prawdziwym Bashem

W WSL `g++` tworzy programy linuksowe, a nie pliki `.exe` dla Windows.

Podstawowe polecenia

`pwd`	wyświetla ścieżkę bieżącego katalogu
`ls`, `dir`	wyświetla zawartość bieżącego katalogu
`cd katalog`	przejście do podkatalogu `katalog`
`cd ..`	przejście do katalogu nadrzędnego
`cd ~`	przejście do katalogu domowego
`g++ --version`	wersja kompilatora (czy jest dostępny)

`pwd`, `ls` to aliasy poleceń PowerShella (`Get-Location`, `Get-ChildItem`) - nazwy jak w Bashu, ale opcje inne (np. `ls -la` nie zadziała).

Windows nie rozróżnia wielkości liter:
`Program.cpp` = `program.cpp`

Czy mamy kompilator g++?

`g++ --version` – jeśli PowerShell nie rozpoznaje polecenia `g++`, kompilatora nie ma lub nie ma go w zmiennej `PATH`.

Instalacja **MinGW-w64** (WinLibs, bez instalatora):

1. pobrać archiwum ZIP z `https://winlibs.com` (UCRT, 64-bit)
2. rozpakować do `C:\` – powstanie `C:\mingw64`
3. dodać `C:\mingw64\bin` do `PATH`: Start → „zmienne środowiskowe" → `Path` → Nowy
4. zamknąć i ponownie otworzyć terminal, sprawdzić `g++ --version`

Alternatywa: **MSYS2** (`https://www.msys2.org`) – instalator z menedżerem pakietów.

Kompilacja i uruchamianie programu

(Wymaga kompilatora **MinGW-w64** dodanego do zmiennej `PATH`.)

`g++ program.cpp`	tworzy plik `a.exe`
`.\a.exe`	uruchamia go
`g++ program.cpp -o program`	tworzy plik `program.exe`
`.\program`	uruchamia go

Kompilacja z ostrzeżeniami:

```
g++ -Wall -Wextra program.cpp -o program
```

`.\` oznacza bieżący katalog – jak `./` w Linuksie.

Skrypt PowerShella

Na przykład plik: `compandrun.ps1`:

```
Write-Host "--- Compiling program ---"
g++ main.cpp -o main.exe
if ($LASTEXITCODE -ne 0) {
 Write-Host "--- Compilation failed! ---" -ForegroundColor Red
 exit 1
}
Write-Host "--- Compilation successful ---"
Write-Host "--- Running program ---"
.\main.exe
if ($LASTEXITCODE -ne 0) {
 Write-Host "--- Program terminated with an error! ---" -ForegroundColor Red
 exit $LASTEXITCODE
}
```

`$LASTEXITCODE`	kod zakończenia programu (0 = sukces)
`if (...) { exit 1 }`	przerywa skrypt po błędzie kompilacji

Uruchamianie skryptu PowerShella

```
.\compandrun.ps1
```

Zasady wykonywania skryptów (Execution Policy)

W Windows 11 skrypty `.ps1` są domyślnie zablokowane ze względów bezpieczeństwa!

Sprawdzenie bieżących ustawień:

```
Get-ExecutionPolicy -List
```

Trwale, dla bieżącego użytkownika (dozwolone własne skrypty):

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Tylko w bieżącej sesji (do zamknięcia okna):

```
Set-ExecutionPolicy Bypass -Scope Process
```

Powrót do ustawień domyślnych:

```
Set-ExecutionPolicy Undefined -Scope CurrentUser
```

Zalety pracy w terminalu

Dlaczego warto znać terminal, mając IDE?

Zrozumienie widać każdy etap i każdą opcję kompilacji – IDE wykonuje je w tle

Automatyzacja polecenia łatwo zapisać w skrypcie lub `Makefile` i powtarzać, niezależnie od IDE

Praca zdalna na serwerach i klastrach obliczeniowych (SSH) terminal jest podstawowym narzędziem

Lekkość działa na słabym sprzęcie i bez środowiska graficznego

IDE i terminal się uzupełniają – większość IDE ma wbudowany terminal.

Podsumowanie

- Kod źródłowy trzeba zbudować: preprocesor → kompilator → linker tworzą plik wykonywalny.
- Kompilator g++ (MinGW-w64) uruchamiamy w terminalu; opcje `-Wall -Wextra` włączają ostrzeżenia.
- Terminal wyświetla tekst, a powłoka (PowerShell, Bash) wykonuje wpisane polecenia.
- Powtarzane polecenia (kompilacja, uruchomienie) można zapisać w skrypcie `.ps1` lub `.sh`.
- Rozróżniamy błędy kompilacji, wykonania i logiczne; ostrzeżeń kompilatora nie ignorujemy.
- AI pomaga w nauce, ale oddawany kod trzeba rozumieć; na zaliczeniu AI jest zabronione.

Pytania kontrolne

1. Czym różni się kod źródłowy od pliku wykonywalnego?
2. Jakie są etapy budowania programu w C++ i co robi każdy z nich?
3. Czym różni się terminal od powłoki?
4. Co oznacza `.\` w poleceniu `.\main`?
5. Co robi polecenie `g++ -Wall -Wextra main.cpp -o main`?
6. Czym różni się błąd kompilacji od błędu logicznego?
7. Dlaczego skrypt `.ps1` może się nie uruchomić i jak to naprawić?
8. Po co uczyć się pracy w terminalu, skoro istnieją IDE?