Podział programu na pliki

Pliki nagłówkowe (.h) vs. Pliki źródłowe (.cpp)

Po co w ogóle dzielić kod?

- **Organizacja i czytelność:** Kod podzielony na logiczne moduły jest łatwiejszy do zrozumienia i zarządzania.
- **Skrócenie czasu kompilacji:** Zmiana w jednym pliku .cpp nie wymaga ponownej kompilacji całego projektu.
- **Oddzielenie interfejsu od implementacji:**
  - **Interfejs (.h):** Mówi, **CO** dany moduł robi.
  - **Implementacja (.cpp):** Mówi, **JAK** to robi.
- **Współpraca zespołowa:** Różni programiści mogą pracować nad różnymi modułami jednocześnie.
- **Wielokrotne użycie kodu:** Ten sam moduł (np. do obsługi wektorów) może być łatwo użyty w wielu różnych projektach.

Deklaracja vs. Definicja

- **Deklaracja (ang. *declaration*)**
  - Informuje kompilator o istnieniu czegoś (funkcji, zmiennej, klasy).
  - Przykład: int gcd(int a, int b);
  - **Może występować wiele razy.**
- **Definicja (ang. *definition*)**
  - Dostarcza pełną implementację (ciało funkcji)
    lub rezerwuje pamięć (dla zmiennej).
  - Przykład: int gcd(int a, int b) { /* ... kod ... */ }
  - **Zasada Jednej Definicji (ODR):**
    W całym programie może istnieć **tylko jeden raz!**

Przykład 1 (Proceduralnie)

```
// Plik: matematyka.h

#ifndef MATEMATYKA_H
#define MATEMATYKA_H

// Deklaracja funkcji obliczającej
największy wspólny dzielnik
int gcd(int a, int b);

// Deklaracja funkcji obliczającej
najmniejszą wspólną wielokrotność
int lcm(int a, int b);

#endif // MATEMATYKA_H
```

```
// Plik: matematyka.cpp

#include "matematyka.h" // Łączymy
implementację z jej deklaracją

// Definicja funkcji gcd
int gcd(int a, int b) {
    // ... ciało funkcji ...
}

// Definicja funkcji lcm
int lcm(int a, int b) {
    // ... ciało funkcji ...
}
```

Przykład 2 (Obiektowo)

```
// Plik: Rectangle.h

#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h); // Deklaracja konstruktora
    double getArea() const;        // Deklaracja metody
};

#endif
```

Przykład 2 (Obiektowo) – ciąg dalszy

```
// Plik: Rectangle.cpp

#include "Rectangle.h"

// Definicja konstruktora
Rectangle::Rectangle(double w, double h)
{
    width = w;
    height = h;
}

// Definicja metody
double Rectangle::getArea() const
{
    return width * height;
}
```

Jak to działa w praktyce?
Kompilacja z linii poleceń

- **Wariant 1: Kompilacja i linkowanie za jednym razem**

```
# Kompilator sam zajmie się kompilacją każdego .cpp,
# a następnie połączeniem (linkowaniem) wyników w jeden program.

g++ main.cpp matematyka.cpp -o moj_program
```

Jak to działa w praktyce? Kompilacja z linii poleceń

- **Wariant 2: Oddzielna kompilacja i linkowanie**

```
# Kompilujemy main.cpp -> powstaje main.o
g++ -c main.cpp -o main.o

# Kompilujemy matematyka.cpp -> powstaje matematyka.o
g++ -c matematyka.cpp -o matematyka.o


# Linker łączy main.o i matematyka.o w jeden program
g++ main.o matematyka.o -o moj_program
```

Automatyzacja Procesu: Plik Makefile

- Ręczne wpisywanie komend kompilacji jest uciążliwe i podatne na błędy. Narzędzie make automatyzuje ten proces. Czyta ono plik o nazwie Makefile, w którym opisane są zależności między plikami i komendy potrzebne do ich zbudowania.
- make jest inteligentny: przebudowuje tylko te pliki, które są przestarzałe (np. gdy plik .cpp został zmieniony po ostatniej kompilacji pliku .o).

Przykładowy plik Makefile

```
# Używane zmienne - ułatwiają zmiany
CXX = g++
CXXFLAGS = -Wall -std=c++17

# --- Reguła domyślna: co chcemy zbudować ---
# Mówi, że celem jest plik "moj_program".
# Zależy on od plików main.o i matematyka.o.
all: moj_program

# --- Reguła linkowania ---
# Mówi, jak stworzyć "moj_program" z plików obiektowych.
moj_program: main.o matematyka.o
	$(CXX) main.o matematyka.o -o moj_program

# --- Reguły kompilacji (jedna dla każdego pliku .cpp) ---

# Mówi, jak stworzyć plik main.o.
# Zależy on od main.cpp ORAZ od nagłówka matematyka.h!
main.o: main.cpp matematyka.h
	$(CXX) $(CXXFLAGS) -c main.cpp -o main.o

# Mówi, jak stworzyć plik matematyka.o.
# Zależy on od matematyka.cpp i matematyka.h.
matematyka.o: matematyka.cpp matematyka.h
	$(CXX) $(CXXFLAGS) -c matematyka.cpp -o matematyka.o
```

Krótka historia make:
Jak frustracja napędza innowację

- **Problem lat 70-tych: Ludzka pamięć**
- Przed powstaniem make, proces budowania programów był serią ręcznych, powtarzalnych komend. Programy stawały się coraz większe, składały się z dziesiątek plików. Największą zmorą programistów było pamiętanie, które pliki zostały zmienione i które w związku z tym trzeba ponownie skompilować.
- Często marnowano całe godziny na szukanie błędu, którego przyczyną był fakt, że programista zapomniał o ponownej kompilacji jednego z modułów po wprowadzeniu w nim poprawki.

Narodziny Automatyzacji

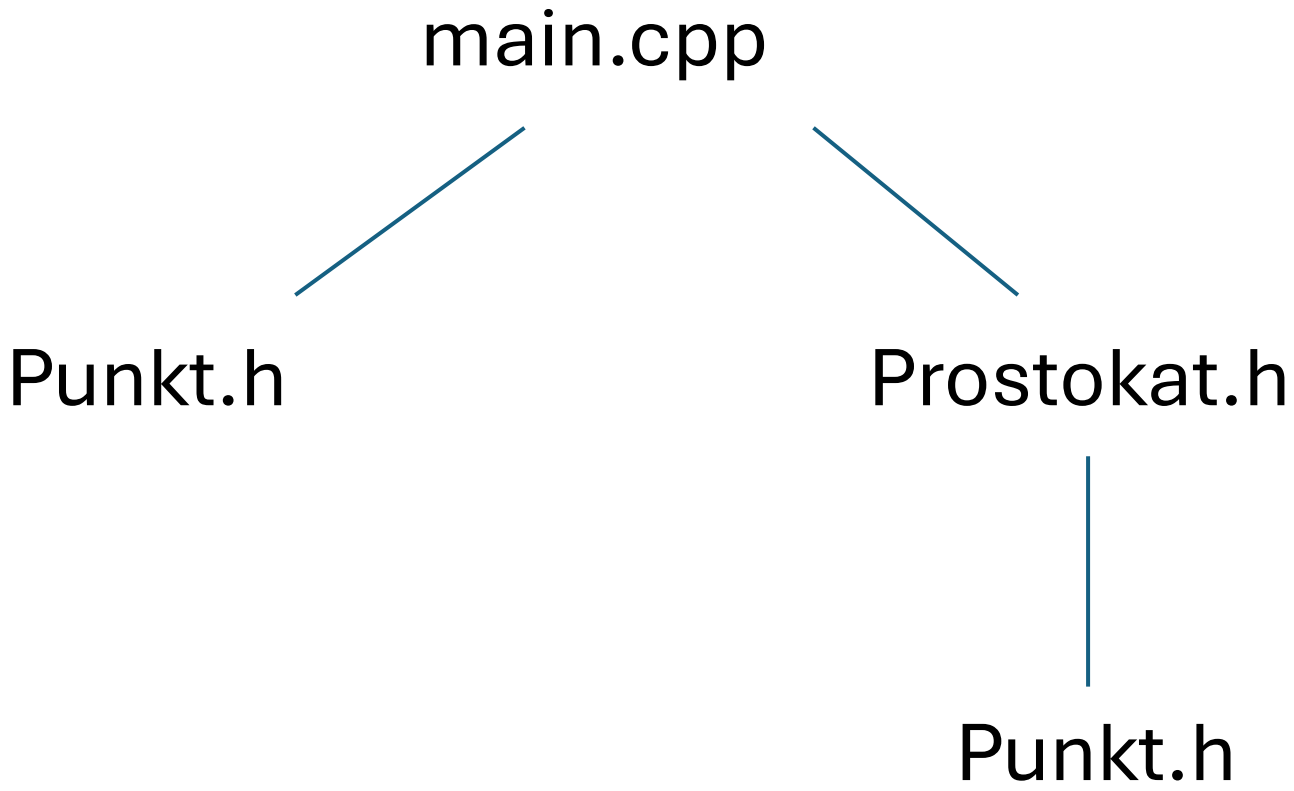
- **Kto?** Stuart Feldman.
- **Kiedy?** W 1976 roku, w legendarnych laboratoriach Bell Labs.

  „Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.”

make dzisiaj i rola IDE

- Narzędzie make (i jego nowocześni następcy jak CMake) wciąż jest fundamentem budowania oprogramowania, zwłaszcza w środowiskach Linux i w dużych, wieloplatformowych projektach.
- Jednak problem, który make rozwiązał, jest dzisiaj często niewidoczny dla programisty.
- **Nowoczesne środowiska programistyczne (IDE)**, takie jak Visual Studio, CLion czy VS Code, mają wbudowane systemy budowania. Gdy klikasz przycisk **"Build"** lub **"Run"**, IDE w tle wykonuje dokładnie taką samą analizę zależności, jaką robi make, i przekompilowuje tylko to, co jest absolutnie konieczne.

Problem: Co gdy dołączymy nagłówek dwa razy?



Preprocesor wklei zawartość Punkt.h do main.cpp **dwa razy**!

Rozwiązanie: Include Guard (Strażnik Dołączeń)

```
#ifndef NAZWA_STRAZNIKA  // "Jeśli to makro NIE jest zdefiniowane..."
#define NAZWA_STRAZNIKA  // "...to je zdefiniuj..."

// ... i wstaw całą zawartość pliku nagłówkowego ...

#endif // NAZWA_STRAZNIKA // "...koniec bloku warunkowego."
```

Przy drugiej próbie dołączenia, warunek #ifndef będzie fałszywy i preprocesor pominie całą zawartość pliku.
KAŻDY PLIK NAGŁÓWKOWY POWINIEN MIEĆ STRAŻNIKA!

Błędy Kompilacji vs. Błędy Linkowania

- **Błąd Kompilacji:**
  - Występuje, gdy kompilator analizuje **jeden plik .cpp** (jednostkę translacji).
  - Nie rozumie kodu: błędy składni, redefinicje w tym samym pliku.
  - **Include guard bezpośrednio chroni przed błędem kompilacji.**
- **Błąd Linkowania:**
  - Występuje, gdy linker próbuje połączyć **wiele plików obiektowych** (.obj, .o) w jeden program.
  - Nie może znaleźć definicji (np. undefined reference) lub znajduje ich za dużo (np. multiple definition).
  - **Najczęstsza przyczyna: umieszczenie definicji funkcji w pliku .h.**
  - **Inna częsta przyczyna: problem z użyciem biblioteki zewnętrznej**

Studium Przypadku: Błąd Kompilacji

Scenariusz: Punkt.h bez strażnika, dołączony bezpośrednio i pośrednio w main.cpp.

```
// main.cpp po działaniu preprocesora
struct Punkt { ... }; // z pierwszego include
struct Punkt { ... }; // z drugiego include (poprzez Prostokat.h)
```

Wynik: Błąd Kompilacji: error: redefinition of 'struct Punkt'.
Wniosek: Strażnik zapobiega wielokrotnemu wklejeniu kodu do JEDNEGO pliku .cpp, chroniąc przed błędem kompilatora.

Studium Przypadku: Błąd Linkowania

Scenariusz: Funkcja logMessage() zdefiniowana w Logger.h.
Plik ten jest dołączony do main.cpp i ModulA.cpp.

1) main.cpp kompiluje się poprawnie -> powstaje main.obj z definicją logMessage.
2) ModulA.cpp kompiluje się poprawnie -> powstaje ModulA.obj z definicją logMessage.
3) Linker próbuje połączyć main.obj i ModulA.obj.

Wynik: Błąd Linkera: multiple definition of 'logMessage'.
Wniosek: Umieszczanie definicji funkcji w nagłówkach łamie Zasadę Jednej Definicji.

Wyjątki od Reguły Jednej Definicji (ODR)

Pytanie: "Ale przecież definicja class Rectangle { ... }; jest w nagłówku. Czemu to nie powoduje błędu linkera?"

Odpowiedź: Ponieważ standard C++ przewiduje **wyjątki** od ODR dla bytów, które muszą być znane w wielu plikach.

Wyjątki (mogą być definiowane w plikach .h):

- **Definicje klas/struktur.**
- Funkcje inline.
- Szablony (templates).

Brak wyjątku (MUSZĄ być w .cpp):

- **Definicje zwykłych funkcji i metod.**
- **Definicje zmiennych globalnych i statycznych członków klas!**

Podsumowanie i Dobre Praktyki

- **ZAWSZE** używaj include guards w każdym pliku .h.
- **W plikach .h umieszczaj:**
  - Deklaracje funkcji (void func();).
  - Definicje klas/struktur (class X { ... };).
  - Deklaracje extern dla zmiennych globalnych (extern int x;).
- **W plikach .cpp umieszczaj:**
  - Definicje funkcji i metod (void func() { ... }).
  - Definicje zmiennych globalnych i statycznych (int x = 10;).
- Pamiętaj o różnicy między błędem kompilacji (problem w jednym pliku) a błędem linkowania (problem między plikami).

Patrząc w przyszłość: Moduły w C++20

- Standard **C++20** wprowadza zupełnie nowy paradygmat: **Moduły**.
- To największa zmiana w organizacji kodu od początku istnienia języka.
- Zamiast tekstowego wklejania plików, moduły opierają się na binarnej, skompilowanej informacji o interfejsie. Kluczowe słowa to import i export.

Patrząc w przyszłość: Moduły w C++20

```
// Plik: matematyka.cppm
export module matematyka; // Definiujemy moduł o nazwie "matematyka"

// Eksportujemy tylko to, co ma być publiczne
export int gcd(int a, int b) {
    // ... implementacja jest tutaj ...
}

export int lcm(int a, int b) {
    // ... implementacja jest tutaj ...
}

// Funkcja pomocnicza, bez 'export' - jest prywatna dla modułu
int internal_helper(int n) {
    return n * 2;
}
```

Patrząc w przyszłość: Moduły w C++20

```
// Plik: main.cpp
import matematyka; // Zamiast #include "matematyka.h"
import <iostream>;  // Standardowa biblioteka też staje się modułowa!

int main() {
    std::cout << "GCD = " << gcd(54, 24) << std::endl;
    // internal_helper(10); // BŁĄD KOMPILACJI! Nie zostało wyeksportowane.
    return 0;
}
```

Patrząc w przyszłość: Moduły w C++20

Główne Zalety Modułów:

- **Drastycznie szybsza kompilacja:** Koniec z wielokrotnym przetwarzaniem tych samych nagłówków.
- **Koniec z include guards:** Problem, który rozwiązują, po prostu znika.
- **Silna izolacja:** Makra i inne definicje nie "wyciekają" z modułów. Koniec z konfliktami nazw.
- **Jawne zależności:** import jasno pokazuje, z czego korzysta dany plik.

Patrząc w przyszłość: Moduły w C++20

- Moduły są częścią standardu C++20.
- Najnowsze wersje kompilatorów (MSVC, GCC, Clang) mają już w dużej mierze zaimplementowane wsparcie.
- Jednak największym wyzwaniem jest adaptacja ekosystemu – systemów budowania (CMake, Make), menedżerów pakietów i przyzwyczajeń programistów.
- To przyszłość, ale jej pełna adaptacja zajmie jeszcze kilka lat.