

C++

Kod w jednym pliku vs. w kilku plikach

Kod w jednym pliku:

- Wszystkie funkcje, zmienne, definicje klas są umieszczone w jednym pliku, np. `main.cpp`.
- **Zalety:** Proste do rozpoczęcia, łatwe w zarządzaniu przy małych projektach.
- **Wady:** Gdy projekt rośnie, plik staje się duży i trudny do zarządzania. Trudniej zlokalizować błędy i utrzymywać kod.

Podział na kilka plików:

- Kiedy projekt rośnie, kod zazwyczaj dzielony jest na pliki w celu lepszego zarządzania. W C++ mamy pliki nagłówkowe (`.h`) i pliki implementacyjne (`.cpp`).
- **Zalety:** Łatwiejsze zarządzanie dużymi projektami, możliwość wielokrotnego użycia kodu (np. poprzez importowanie nagłówków), szybsza kompilacja (tylko zmienione pliki są kompilowane).
- **Wady:** Początkowa konfiguracja jest bardziej skomplikowana i wymaga zrozumienia, jak deklaracje i definicje działają w C++.

Header-only vs. podział na plik `.h` i plik `.cpp`

Podejście "header-only":

- Wszystkie definicje funkcji, klas, itp. są umieszczone w pliku nagłówkowym (`.h`).
- **Zalety:** Prostsza organizacja, bo jest tylko jeden plik. Łatwo można dołączyć taki plik do innych projektów bez konieczności kompilowania osobnych plików.
- **Wady:** Może prowadzić do problemów z wydajnością, ponieważ każdy plik, który dołączy ten nagłówek, będzie kompilował całą zawartość nagłówka. Powoduje to dłuższe czasy kompilacji i potencjalne problemy z wielokrotnymi definicjami (chyba że użyjemy strażników nagłówka, np. `#pragma once`).

Podział na `.h` i `.cpp`:

- Plik `.h` zawiera tylko deklaracje funkcji, klas, itp., a ich definicje znajdują się w pliku `.cpp`.
- **Zalety:** Dłuższe czasy kompilacji są redukowane, ponieważ pliki nagłówkowe nie zawierają pełnych definicji. Definicje są w osobnych plikach `.cpp`, które są kompilowane tylko raz. Ułatwia to także modularność i ponowne wykorzystanie kodu.
- **Wady:** Wymaga dodatkowej struktury i zrozumienia, jak pliki `.h` i `.cpp` współpracują ze sobą.

Kod w jednym pliku	Podejście "header-only"	Podział na plik <code>.h</code> i <code>.cpp</code>
<p>Brak większych zmian w składni: Wszystkie deklaracje i definicje są po prostu umieszczone w jednym pliku. Nie ma tu żadnych różnic składniowych w stosunku do reszty podejść.</p>	<p>Deklaracje i definicje w pliku nagłówkowym (<code>.h</code>): W tym podejściu umieszczasz zarówno deklaracje, jak i definicje w pliku nagłówkowym.</p> <p>Zasadnicza różnica: Wszystkie definicje funkcji i metod klas są umieszczone w pliku <code>.h</code>. Aby zapobiec problemom z wielokrotnym dołączaniem plików nagłówkowych, musisz używać strażników nagłówka (<code>#ifndef</code>, <code>#define</code>) lub <code>#pragma once</code>.</p>	<p>Deklaracje w pliku <code>.h</code>, definicje w pliku <code>.cpp</code>: W tym podejściu deklaracje funkcji i klas znajdują się w pliku nagłówkowym <code>.h</code>, natomiast ich definicje w pliku źródłowym <code>.cpp</code>.</p> <p>Różnica składniowa: W pliku <code>.h</code> znajdują się tylko nagłówki funkcji/metod, bez ich implementacji. W pliku <code>.cpp</code> używasz składni <code>ClassName::methodName</code> do implementacji metod.</p>

Przykłady: Kod w jednym pliku

```
#include <iostream>
class Cat {
public:
    void meow() {
        std::cout << "Meow!" << std::endl;
    }
};

int main() {
    Cat myCat;
    myCat.meow();
    return 0;
}
```

Przykłady: Podejście "header-only"

Plik Cat.h:

```
#pragma once
#include <iostream>

class Cat {
public:
    void meow() {
        std::cout << "Meow!" << std::endl;
    }
};
```

Plik Main.cpp:

```
#include "Cat.h"

int main() {
    Cat myCat;
    myCat.meow();
    return 0;
}
```

Przykłady: Podział na plik .h i .cpp

Plik Cat.h:

```
#ifndef CAT_H
#define CAT_H

class Cat {
public:
    void meow();
};

#endif
```

Plik Cat.cpp:

```
#include "Cat.h"
#include <iostream>

void Cat::meow() {
    std::cout << "Meow!" << std::endl;
}
```

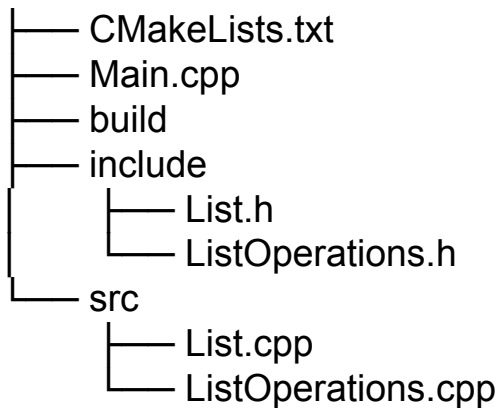
Plik Main.cpp:

```
#include "Cat.h"

int main() {
    Cat myCat;
    myCat.meow();
    return 0;
}
```

Kompilacja za pomocą CMake

Struktura projektu:



Jak kompilujemy?

Kiedy znajdujemy się w projekcie:

```
cd build  
cmake ..  
make
```

I gotowe.