

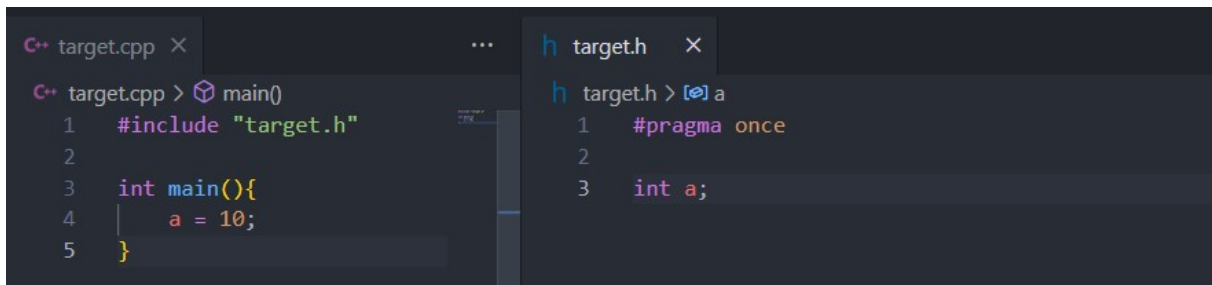
## **Zawartość**

- include
- makra define
- instrukcje warunkowe preprocesora

## Preprocesor

- Jest to pierwszy etap procesu, który nazywamy kompilacją
- Prawdę mówiąc preprocesor to nie jest kompilacja
- Preprocesor to etap poprzedzający właściwą kompilację

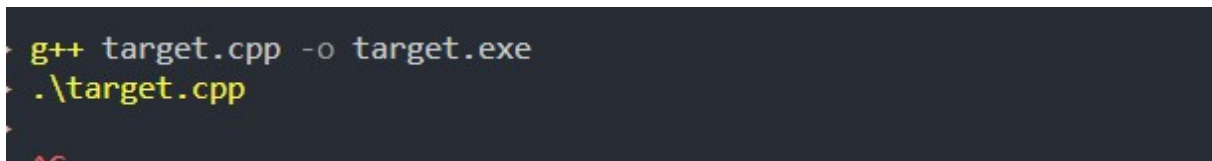
Zobaczmy co się stanie jeżeli przerwę proces kompilacji po wykonaniu preprocesora:



```
C++ target.cpp > main()
1 #include "target.h"
2
3 int main(){
4     a = 10;
5 }

h target.h > a
1 #pragma once
2
3 int a;
```

W pliku \*.cpp utworzyliśmy tylko maina. Do tego załączyliśmy plik \*.h z definicją zmiennej. Przerwijmy proces kompilacji po preprocesorze.

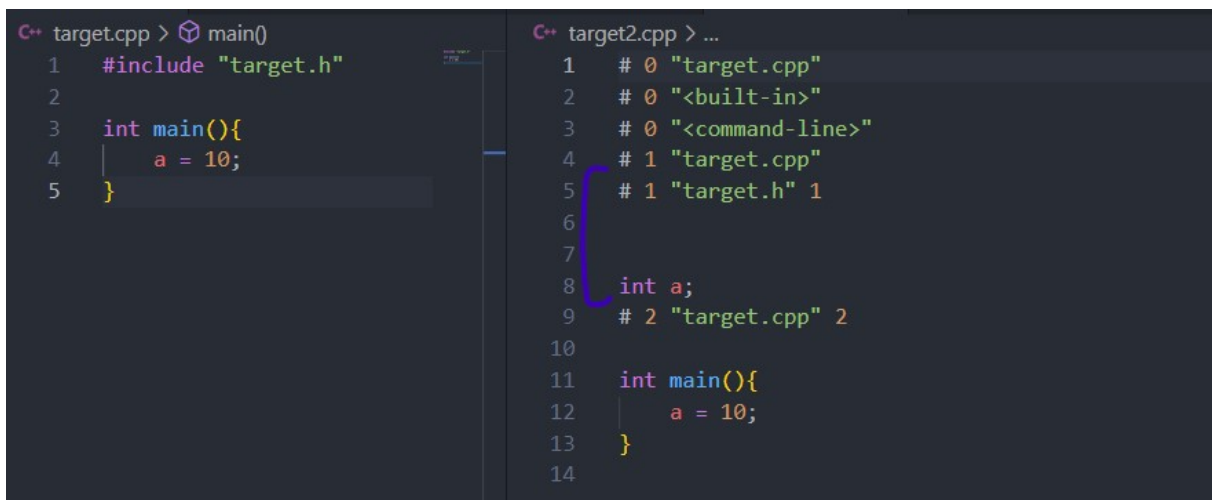


```
g++ target.cpp -o target.exe
.\target.cpp
```

Jeżeli kogoś interesuje jak kompiluje ten program za pomocą starego dobrego GCC. Flaga -o to skrót od output.

Dodanie flagi -E przerywa kompilację po procesorze.

```
g++ target.cpp -E -o target2.cpp
```



```
C++ target.cpp > main()
1 #include "target.h"
2
3 int main(){
4     a = 10;
5 }

C++ target2.cpp > ...
1 # 0 "target.cpp"
2 # 0 "<built-in>"
3 # 0 "<command-line>"
4 # 1 "target.cpp"
5 # 1 "target.h" 1
6
7
8 int a;
9 # 2 "target.cpp" 2
10
11 int main(){
12     a = 10;
13 }
14
```

## Include

Instrukcja include działa w bardzo prymitywny sposób. Powoduje ona po prostu wklejenie całego pliku w miejsce jej wystąpienia.

Dlatego ekstremalnie ważne jest by w plikach \*.h umieszczać tak zwane **include guard** w postaci:

```
1  #pragma once
2
3  int a;
```

Lub:

```
1  ✓ #ifndef TARGET_H
2    #define TARGET_H
3
4    int a;
5
6    #endif //TARGET_H
```

Dlaczego to takie ważne? Pisząc kod w C/C++ piszemy #include x prawdopodobnie wiele razy w jednym projekcie (pewnie w każdym pliku). Dobrze było by powiedzieć preprocesorowi, że ten gdy będzie składał kod w całość to niech sprawdza czy dany plik już dokleił.

Jeżeli tego nie dodamy to plik zostanie wklejony wiele razy co w dalszej fazie kompilacji spowoduje błąd.

## Warto wyjaśnić po co w C i C++ są pliki h?

Zawierają one **deklaracje** a nie definicje funkcji. Przykład:

```
3  int main(){
4      int a = add(1,2);
5  }
6
7  int add(int a, int b){
8      return a + b;
9  }
10
```

Ten kod nie zadziała, ponieważ w linii 4 funkcja add() jest niezdefiniowana. W C/C++ aby funkcja była możliwa do użycia musi być zadeklarowana/zdefiniowana wyżej w pliku. Czyli np. tak:

```

3  ✓ int add(int a, int b){
4      |     return a + b;
5      |
6      }
7  ✓ int main(){
8      |     int a = add(1,2);
9      |
10     }

```

Teraz program się skompiluje, ponieważ w linii 8 funkcja `add()` jest już znana. W innych językach nie ma tego problemu, ponieważ kompilatory najpierw się uczą funkcji, a później w drugim przebiegu ewaluują kod.

Inne rozwiązanie to rozdzielić deklarację od definicji:

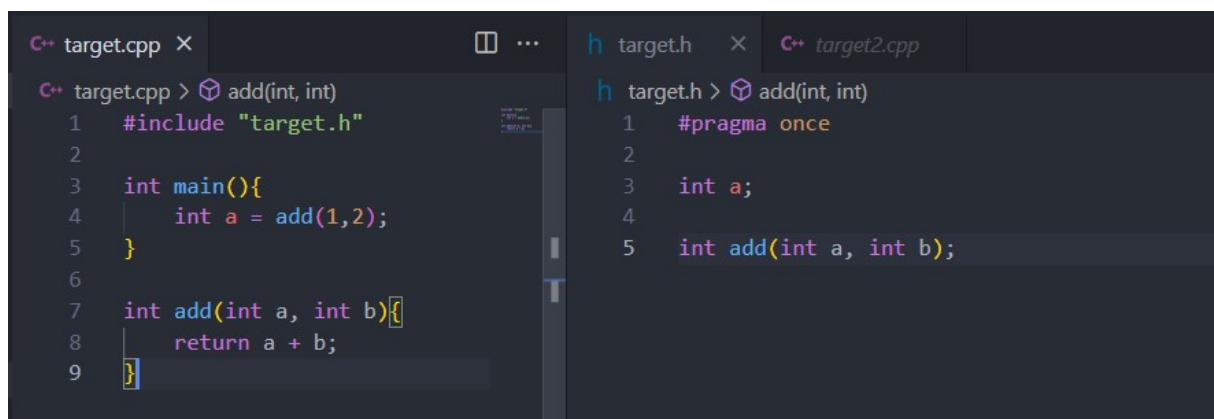
```

3  int add(int a, int b);
4
5  ✓ int main(){
6      |     int a = add(1,2);
7      |
8      }
9  ✓ int add(int a, int b){
10     |     return a + b;
11     |
12     }

```

Teraz mówimy za pomocą deklaracji w linii 3, że gdzieś będzie zdefiniowana funkcja `add()`.

**Działanie plików \*.h jest identyczne:**



```

C++ target.cpp > add(int, int)
1  #include "target.h"
2
3  int main(){
4      |     int a = add(1,2);
5      |
6      }
7  int add(int a, int b){
8      |     return a + b;
9      |
10     }

```

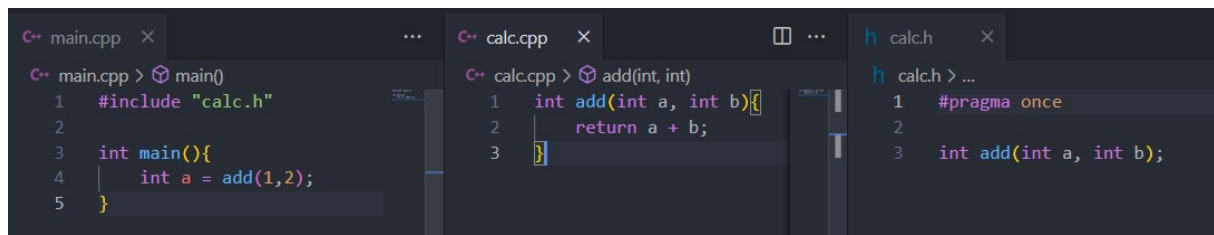
```

h target.h > add(int, int)
1  #pragma once
2
3  int a;
4
5  int add(int a, int b);

```

W miejscu `include` doklejona zostanie zawartość pliku `target.h` co w konsekwencji da nam kod identyczny jak obrazek wcześniej.

**Może się to wydawać w tym momencie bezsensu. Bo po co to wszystko. Ano to wszystko nabiera sensu jak używamy w kodzie wiele plików:**



```
main.cpp > main()
1 #include "calc.h"
2
3 int main(){
4     int a = add(1,2);
5 }

calc.cpp > add(int, int)
1 int add(int a, int b){
2     return a + b;
3 }

calc.h > ...
1 #pragma once
2
3 int add(int a, int b);
```

Mamy plik main.cpp z funkcją main(). Ta funkcja wykorzystuje funkcję add() zawartą w pliku calc.cpp.

Kompiluję to w ten sposób:

```
g++ main.cpp calc.cpp -o main.exe
```

Kompilator połączył w sobie znany sposób te dwa pliki. Nie mamy pewności, która funkcja będzie wyżej lub niżej. Na szczęście plik \*.h. wklejany przed main() gwarantuje, że deklaracja funkcji add() znajdzie się przed main() bez względu na to gdzie jest definicja.

Podsumowując instrukcja include wkleja całą zawartość wskazanego pliku w miejsce jej wystąpienia.

**Ważne by w plikach \*.h umieszczać DEKLARACJE a nie definicje bo to może też prowadzić do dziwnych błędów.**

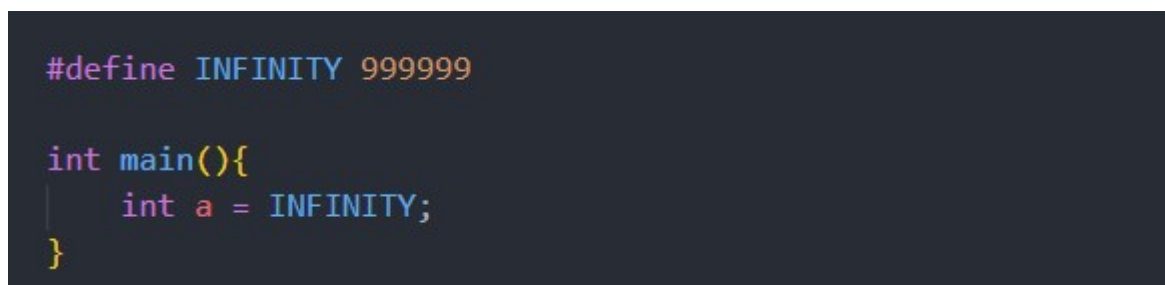
```
int a; //deklaracja
```

```
int a = 10; //definicja
```

**Nie robimy include na plikach \*.cpp!**

**define**

Drugi ważny ficzer preprocesora to marka. Makra na pierwszy rzut oka wyglądają jak stałe const, **ale to nie są stałe const.**



```
#define INFINITY 999999

int main(){
    int a = INFINITY;
}
```

Co tutaj się stanie? Preprocesor przed kompilacją podmieni wszystkie wystąpienia INFINITY na 999999. Dostaniemy:

```

14
15  int main(){
16      int a = 999999;
17  }
18  |

```

Makra można stosować oczywiście do przechowywania stałych, ale zachowują się one inaczej niż `const int x = 999999`

Zmienna typu `const` to zmienna zaalokowana w pamięci aplikacji. Wszystkie odwołania odwołują się do niej w trakcie działania programu. Makro to po prostu informacja dla preprocesora, że wartość X ma podmienić na wartość Y jeszcze przed kompilacją.

```

3  #define SIZE 10
4
5  int size = 10;
6
7  int main(){
8      int array[SIZE]; //OK
9      int array[size]; //error
10 }
11

```

Makra są ok do definiowania rozmiaru tablicy statycznej. Przypominam, że wartość podana w nawiasie kwadratowym musi być stałą czasu kompilacji a zwykła zmienna nią nie jest.

<pre> 16  int main(){ 17      int array[SIZE]; //OK 18 19      for(int i = 0; i &lt; SIZE; i++){ 20          array[i] = SIZE+1; 21      } 22  } </pre>	<pre> 17  int main(){ 18      int array[10]; 19 20      for(int i = 0; i &lt; 10; i++){ 21          array[i] = 10 + 1; 22      } 23  } 24 </pre>
--	--

Powyżej widać kod przed i po wykonaniu preprocesora (wciąż przed kompilacją).

```

2
3  #define SIZE 10;
4
5  int main(){
6      int array[SIZE]; //OK
7
8      for(int i = 0; i < SIZE; i++){
9          array[i] = SIZE+1;
10     }
11 }
12

```

Klasyczny błąd popełniany na makrach. Dodaliśmy błędnie (z automatu) średnik na końcu linii (10;) przez co preprocesor powstawił wszędzie w miejsce SIZE wartość 10; co jest błędem składniowym.

Inny mega popularny błąd to:

```

#define SUM(x,y) x+y

int main(){
    int a = 2*SUM(2,3);
}

```

Makra to podmiana tekstu na tekst. Błędem jest myślenie, że zmienna a przyjmie wartość 10. Zmienna ta przyjmie wartość 7. Dlaczego? Przed kompilacją SUM(2,3) zostanie podmienione na 2+3 więc kolejność wykonywania działań będzie inna. Wygląda to trochę jak funkcja, ale nią nie jest.

## ifdef/ifndef

Jeżeli zdefiniowane/Jeżeli nie zdefiniowane

Konstrukcja stosowana do warunkowego włączania/wyłączania fragmentów kodu.

```

2
3  #define DEBUG
4
5  int main(){
6      //some code
7
8      #ifdef DEBUG
9          printf("Jakaś informacja");
10     #endif
11
12     //some code
13 }
14

```

W czasie gdy makro `DEBUG` jest zdefiniowane kod umieszczony wewnątrz bloku `ifdef` zostanie włączony do kodu kompilowanego. Jeżeli usuniemy makro ten fragment kodu nie będzie kompilowany.

Po co takie coś? Tak jak na przykładzie wyżej. W czasie tworzenia aplikacji często używamy jakichś printów do debugowania i sprawdzania funkcjonalności kodu. Jeżeli decydujemy się wydać aplikację usuwamy makro i pozbywamy się w ten sposób zbędnego kodu z produkcyjnej wersji aplikacji.

Jeżeli kiedyś przyjdzie potrzeba debugować kod możemy makro dodać ponownie.

## If

W miarę zwykła instrukcja warunkowa. Działa jak `ifdef`

```
5  int main(){
6      //some code
7
8  #if DEBUG == 0
9      printf("Jakaś informacja");
10 #elif DEBUG == 1
11     //some code
12 #else
13     //some code
14 #endif
15
16     //some code
17 }
```

Zależnie od wartości makra do kodu wynikowego dodany zostanie odpowiedni fragment kodu.