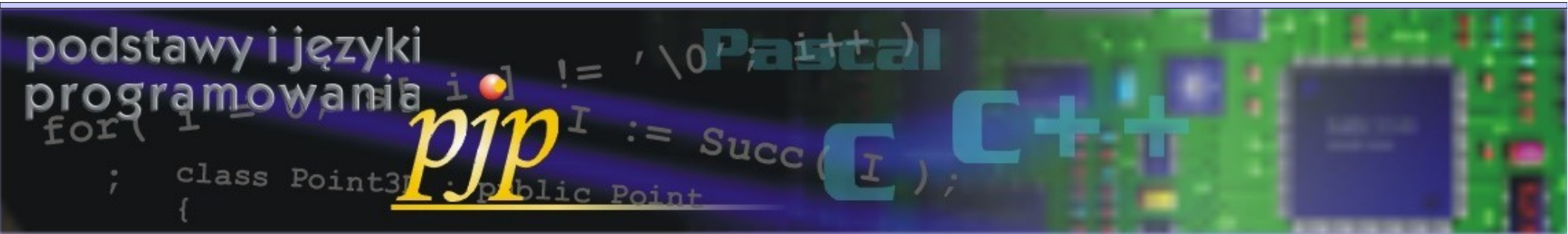
Języki programowania obiektowego
Nieobiektowe elementy języka C++

Roman Simiński
roman.siminski@us.edu.pl
www.programowanie.siminskionline.pl

Klasy pamięci, programy wielomodułowe



Deklaracje zmiennych a struktura programu — klasa pamięci auto

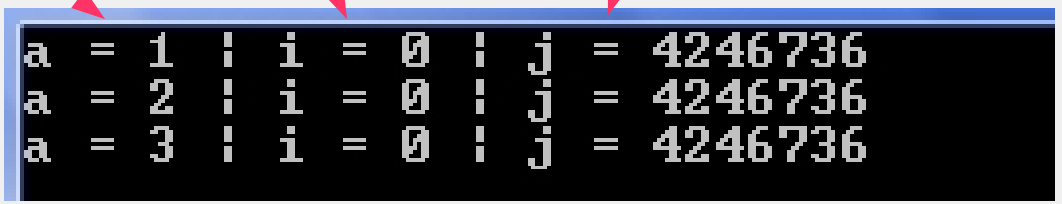
```
void fun( float a )
{
  int   i = 0;
  char  c = 'A';
  float f;

  if( i == 0  )
  {
    float i = 100.0;
    int   k;
    . . .
  }
}
```

Przesłanianie identyfikatorów – w obrębie tego bloku instrukcji **`if`** nazwa **`i`** oznacza, lokalną w tym bloku, zmienną typu **`float`**.

- Zmienne klasy *auto* mogą być definiowane na początku każdego bloku w C89, w standardzie C99 i w języku C++ każdym miejscu dozwolonym syntaktyką języka.
- Zmienne klasy *auto pojawiają* się wraz z wejściem sterowania do bloku w którym są zadeklarowane i *znikają* wraz z wyjściem sterowania z tego bloku.
- Zmienne deklarowane wewnątrz bloku są *automatycznymi*, jeżeli *nie podano* klasy pamięci albo *jawnie użyto* specyfikatora *auto*.

Deklaracje zmiennych a struktura programu — klasa pamięci auto

```
void fun( float a )
{
  int i = 0;
  int j;
  cout << "a = " << a << " | i = " << i << " | j = " << j << endl;
}

int main()
{
  fun( 1.0 );
  fun( 2.0 );
  fun( 3.0 );

  return  EXIT_SUCCESS;
}
```



Zmienne klasy auto:

- *Nie zachowują* swoich wartości pomiędzy swoimi kolejnymi kreacjami.
- O ile nie zostaną *zainicjalizowane*, maja wartości *przypadkowe*.
- Parametry formalne funkcji też są klasy *auto*.

Deklaracje zmiennych a struktura programu — klasa pamięci auto

- Zmienna klasy *auto* tworzone są automatycznie i lokowane są na *stosie*.
- *Stos* to element procesu, służący do przechowywania *danych chwilowych*.
- Na stosie lokowane są *zmienne auto*, w tym *argumenty funkcji*, oraz *adresy powrotu* dla wywoływanych podprogramów.
- Stos ma *ustalony* i *ograniczony* rozmiar – należy sprawdzić ustalenia rozmiaru stosu w opcjach *kompilatora* (lub *konsolidatora*).
- Może się zdarzyć, że stos ma rozmiar rzędu kilku kilobajtów.
- Wobec powyższego, niebezpieczna może być poniższa definicja dużej tablicy:

```
void fun( void )
{
  float tab[ 10000 ];
  . . .
}
```

Deklaracje zmiennych automatycznych w obrębie instrukcji

- W C++ można definiować zmienne w obrębie ograniczonym zasięgiem instrukcji:

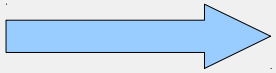
```
for( int i = 10; i > 0; i--)
  cout << endl << i << "...";

i = 0; // Błędna odwołanie poza zasięgiem
```

```
switch( char c = getchar() )
{
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
  case 'y': cout << "Samogłoska " << c;
            break;
  . . .
}

c = getchar(); // Błędna odwołanie poza zasięgiem
```

Uwaga, przedstawione wyżej odwołania do zmiennych definiowanych w obrębie boku dozwolone są w starych kompilatorach języka C++.

Deklaracje zmiennych a struktura programu — klasa pamięci static

```
void fun_auto()
{
  int i = 1;
  cout << endl << "Auto   " << i++;
}

void fun_static()
{
  static int i = 1;
  cout << endl << "Static " << i++;
}

int main()
{
  fun_auto();
  fun_static();

  fun_auto();
  fun_static();

  fun_auto();
  fun_static();
  . . .
 }
```



```
Auto   1
Static 1
Auto   1
Static 2
Auto   1
Static 3
```

Deklaracje zmiennych a struktura programu — klasa pamięci static

- Zmienne *statyczne* mogą być *lokalne w bloku* lub *zewnętrzne* dla wszystkich bloków.
- Jeżeli zmienna wewnątrz bloku zostanie zadeklarowana ze specyfikatorem *static*, to:
  - jest *raz inicjowana wartością inicjalizatora* lub *nadawana jest jej wartość zerowa* odpowiednio do typu.
  - *przechowuje wartość* po opuszczeniu i ponownym wejściu do bloku.
- Statyczne zmienne lokalne stanowią prywatną, nieulotną pamięć danej funkcji czy bloku.

Deklaracje zmiennych a struktura programu — klasa pamięci static

Przykład wykorzystania zmiennej statycznej — funkcja z limitem wywołań w obrębie pojedynczego wykonania programu:

```
void funkcja_z_limitem_wywolan( void )
{
  static int licznik_wywolan = 0;

  if( licznik_wywolan < 10 )
  {
    licznik_wywolan++;

    // Tutaj odpowiednie instrukcje

  }
  else
    cout << "Wersja demo -- limit wywolan wyczerpany";
}
```

Deklaracje zmiennych a struktura programu — klasa pamięci register

- Deklaracja zmiennej jako *register* jest równoważna z deklaracją *auto*, ale wskazuje że deklarowany obiekt będzie intensywnie wykorzystywany, i w miarę możliwości będzie umieszczony w rejestrze procesora.
- Jeżeli *nie jest możliwe* umieszczenie zmiennej w rejestrze, pozostaje ona w *pamięci*.
- Zmienne rejestrowe pozwalają *zredukować zajętość pamięci* i *poprawić szybkość wykonania operacji* takie zmienne wykorzystujących.
- Jednak większość współczesnych kompilatorów wykorzystuje optymalizację rejestrową, zatem wiele zmiennych i tak przechowywanych jest w rejestrach, mimo braku jawnej specyfikacji jako register.

```
register int i;
for( i = 0; i < 10; i++ )
{
  . . .
}
```

```
int very_time_critical_fun( register int i )
{
  . . .
}
```

Deklaracje zmiennych a struktura programu — klasa pamięci extern

```
double dystans, paliwo;

void czytaj_dane()
{
  . . .
  cin >> dystans;
  . . .
  cin >> paliwo;
}

void pisz_wyniki()
{
  if( dystans == 0 )
    cout << "Nie policze spalania dla zerowego dystansu" );
  else
    cout << "Spalanie " << ( paliwo * 100 ) / dystans << "l na 100 km" ;
}

int main()
{
  . . .
  czytaj_dane();
  pisz_wyniki();
  return EXIT_SUCCESS;
}
```

Deklaracje zmiennych a struktura programu — klasa pamięci extern

- Zmienne zewnętrzne deklarowane są na *zewnątrz wszystkich funkcji*. Zasięg zmiennej zewnętrznej rozciąga się *od miejsca deklaracji do końca pliku*.
- Zmienne zewnętrzne istnieją *stale*, nie pojawiają się i nie znikają, zachowują swoje wartości i są dostępne dla wszystkich funkcji programy występujących w zakresie danej zmiennej.
- Zmienna zewnętrzna jest *raz inicjowana wartością inicjalizatora* lub *nadawana jest jej wartość zerowa* odpowiednio do typu.
- Jeżeli dla zmiennej zewnętrznej użyjemy specyfikacji *static*, to oznacza to uprywatnienie (ograniczenie dostępu) w obrębie danego pliku źródłowego.
- Zmienne zewnętrzne oraz ich właściwe definiowanie i deklarowanie mają istotne znaczenie przy organizacji programów wielomodułowych.

Definicje zmiennych o ustalonej wartości

- Słowo kluczowe *const* oznacza modyfikator często używany w C++, dostępny jest również — choć rzadziej używany — w języku C.
- Modyfikator *const* użyty w definicji zmiennej lub parametru oznacza, że wartość takiego elementu *nie może być zmieniana* po zainicjowaniu.

```
const int i = 1;
...
i = 5; // Niedozwolone
```

- Aby wnętrze funkcji nie mogło zmodyfikować parametru aktualnego:

```
void printInt( const int & i )
{
  cout << i;
}
```

```
void inc( const int & i )
{
  ++i; // Nie wolno!
}
```

Mimo, że modyfikator *const* występuje w C i C++ występują pewne różnice w możliwości jego wykorzystania, co zostanie omówione później.

Definicje zmiennych o nieprzewidywalnie zmiennej wartości

- Słowo kluczowe *volatile* to modyfikator oznaczający obiekt o wartościach zmieniających się w nieprzewidywalny sposób, inaczej *obiekty ulotne*.
- Obiektami takimi mogą być zmienne odnoszące się do obiektów zewnętrznych w stosunku do programu — zmiennych nałożonych na porty We/Wy, odnoszących się do obszarów BIOS'a, systemu operacyjnego.
- Takie zmienne nie powinny być poddawane optymalizacjom — szczególnie optymalizacji rejestrowej. Specyfikacja *volatile* zapobiega wszystkim potencjalnym optymalizacjom.

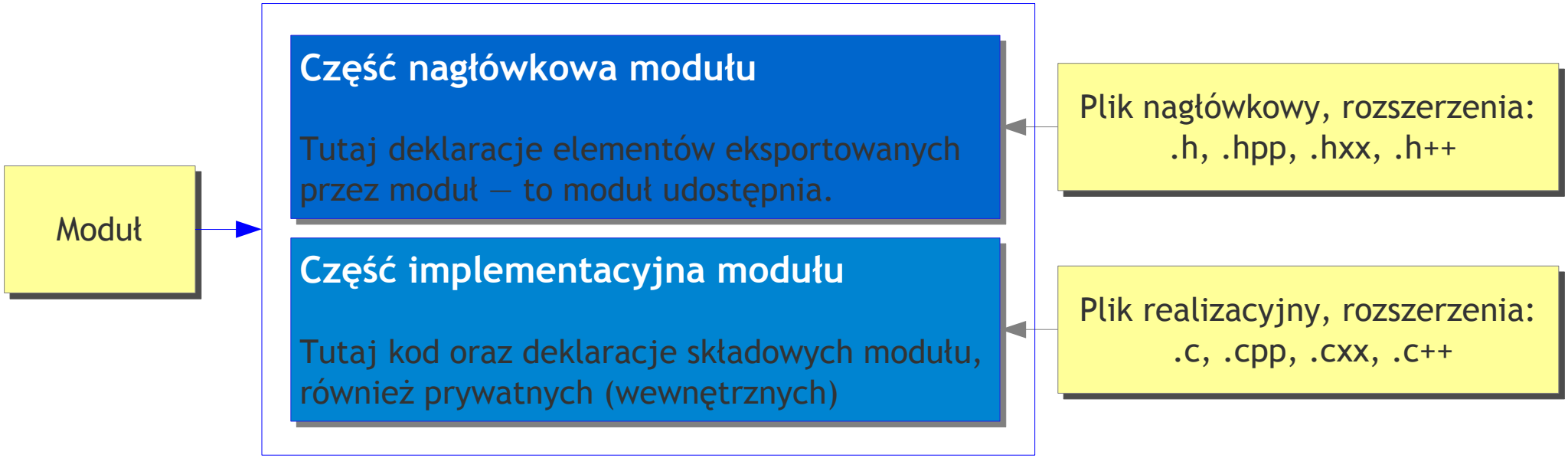
```
volatile unsigned char kbdState = ... ; // Zmienna nałożona na dane BIOS

while( kbdState & LR_SHIFT )
{
  if( kbdState & L_SHIFT )
    // coś tam. . .
  if( kbdState & R_SHIFT )
    // coś tam. . .
  . . .
}
```

Ponieważ zmienna *kbdState* jest z iteracji *while* często wykorzystywana, kompilator może „przenieść” ją do rejestru procesora.

Sprawi to, że ten fragment programu nie będzie działał poprawnie.

Funkcje w osobnym module

- Funkcje o podobnym przeznaczeniu mogą być grupowane w *moduły*.
- Języki C i C++ nie oferują zdefiniowanej *syntaktycznie* modularyzacji.
- Jednak program może się składać z kompilowanych oddzielnie części — zwanych właśnie *modułami* — łączonych potem przez *konsolidator* (wraz z bibliotekami) w plik wykonywalny.
- Przyjęta konwencja budowania modułów zakłada oddzielenie części *publicznej* (*nagłówkowej*) *modułu* od części *implementacyjnej* (*realizacyjnej*).



Część publiczna a cześć implementacyjna

- *Część publiczna* modułu zawiera opis elementów dostępnych do użytku w innych modułach programu.
- W językach C i C++ *część publiczna* modułu to osobny plik — *plik nagłówkowy* (rozszerzenie *h*, *hpp*, *hxx*).
- Plik nagłówkowy jest plikiem tekstowym, udostępniany jest i dystrybuowany w wersji tekstu jawnego.
- *Część implementacyjna* modułu zawiera wszystko to, co potrzebne jest dla działania elementów udostępnianych przez moduł.
- W językach C i C++ *część implementacyjna* modułu to osobny plik — zwykły *plik programu* (rozszerzenie *c*, *cpp*, *cxx*).
- Część implementacyjna może być udostępniana i dystrybuowana w postaci źródłowej lub skompilowanej (pliki *o*, *obj*, *lib*).

Funkcje w osobnym module — przykład

- Załóżmy, że chcemy stworzyć moduł o nawie *fun*, w którym będą zdefiniowane funkcje obliczające pola i obwody figur płaskich.
- Zaczniemy od modułu udostępniającego następujące funkcje:
  - `double pole_kwadratu( double bok );`
  - `double obwod_kwadratu( double bok );`
  - `double pole_kola( double promien );`
  - `double obwod_kola( double promien );`
- Inne moduły programu będą mogły wykorzystywać funkcje zdefiniowane w module *fun*.

Funkcje w osobnym module — koncepcja

Program główny: *main.cpp*

```
int main()
{
  . . .
  cout << pole_kwadratu( 10 ) << endl;
  cout << obwod_kwadratu( 10 ) << endl;
  cout << pole_kola( 10 ) << endl;
  cout << obwod_kola( 10 ) << endl;
  . . .
  return EXIT_SUCCESS;
}
```

Wywołania funkcji udostępnianych przez moduł *fun*

Moduł z funkcjami: *fun*

```
double pole_kwadratu( double bok )
{
   return bok * bok;
}

double obwod_kwadratu( double bok )
{
  return 4 * bok;
}

double pole_kola( double promien )
{
  return 3.14 * promien * promien;
}

double obwod_kola( double promien )
{
  return 2 * 3.14 * promien;
}
```

Definicje funkcji udostępnianych przez moduł *fun*

Funkcje w osobnym module — podział modułu na części

Moduł z funkcjami: *fun*

```
double pole_kwadratu( double bok )
{
   return bok * bok;
}

double obwod_kwadratu( double bok )
{
  return 4 * bok;
}

double pole_kola( double promien )
{
  return M_PI * promien * promien;
}

double obwod_kola( double promien )
{
  return 2 * M_PI * promien;
}
```

Definicje funkcji udostępnianych przez moduł *fun*

Część publiczna modułu: *fun.hpp*

```
double pole_kwadratu( double bok );
double obwod_kwadratu( double bok );
double pole_kola( double promien );
double obwod_kola( double promien );
```

Część implementacyjna : *fun.cpp*

```
double pole_kwadratu( double bok )
{
   return bok * bok;
}

double obwod_kwadratu( double bok )
{
  return 4 * bok;
}

double pole_kola( double promien )
{
  return 3.14 * promien * promien;
}

double obwod_kola( double promien )
{
  return 2 * 3.14 * promien;
}
```

Funkcje w osobnym module — wykorzystanie pliku nagłówkowego

Program główny: *main.cpp*

```
#include <iostream>
#include <cstdlib>
int main()
{
  . . .
  cout << pole_kwadratu( 10 ) << endl;
  cout << obwod_kwadratu( 10 ) << endl;
  cout << pole_kola( 10 ) << endl;
  cout << obwod_kola( 10 ) << endl;
  . . .
  return EXIT_SUCCESS;
}
```

Ostrzeżenie lub błąd kompilacji — definicje (prototypy) tych funkcji nie są znane!

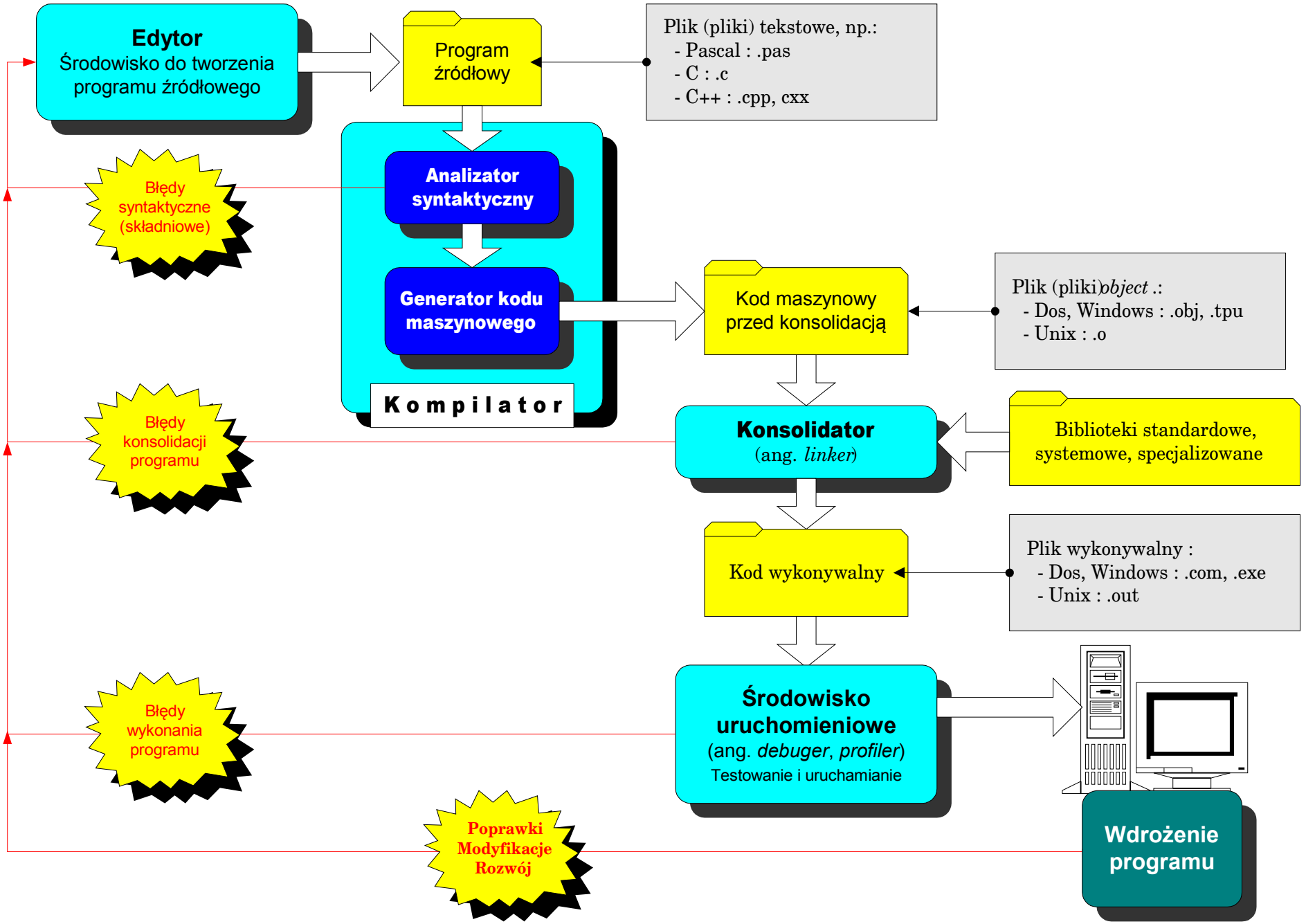
Program główny: *main.cpp*

```
#include <iostream>
#include <cstdlib>
#include "fun.hpp"
int main()
{
  . . .
  cout << pole_kwadratu( 10 ) << endl;
  cout << obwod_kwadratu( 10 ) << endl;
  cout << pole_kola( 10 ) << endl;
  cout << obwod_kola( 10 ) << endl;
  . . .
  return EXIT_SUCCESS;
}
```

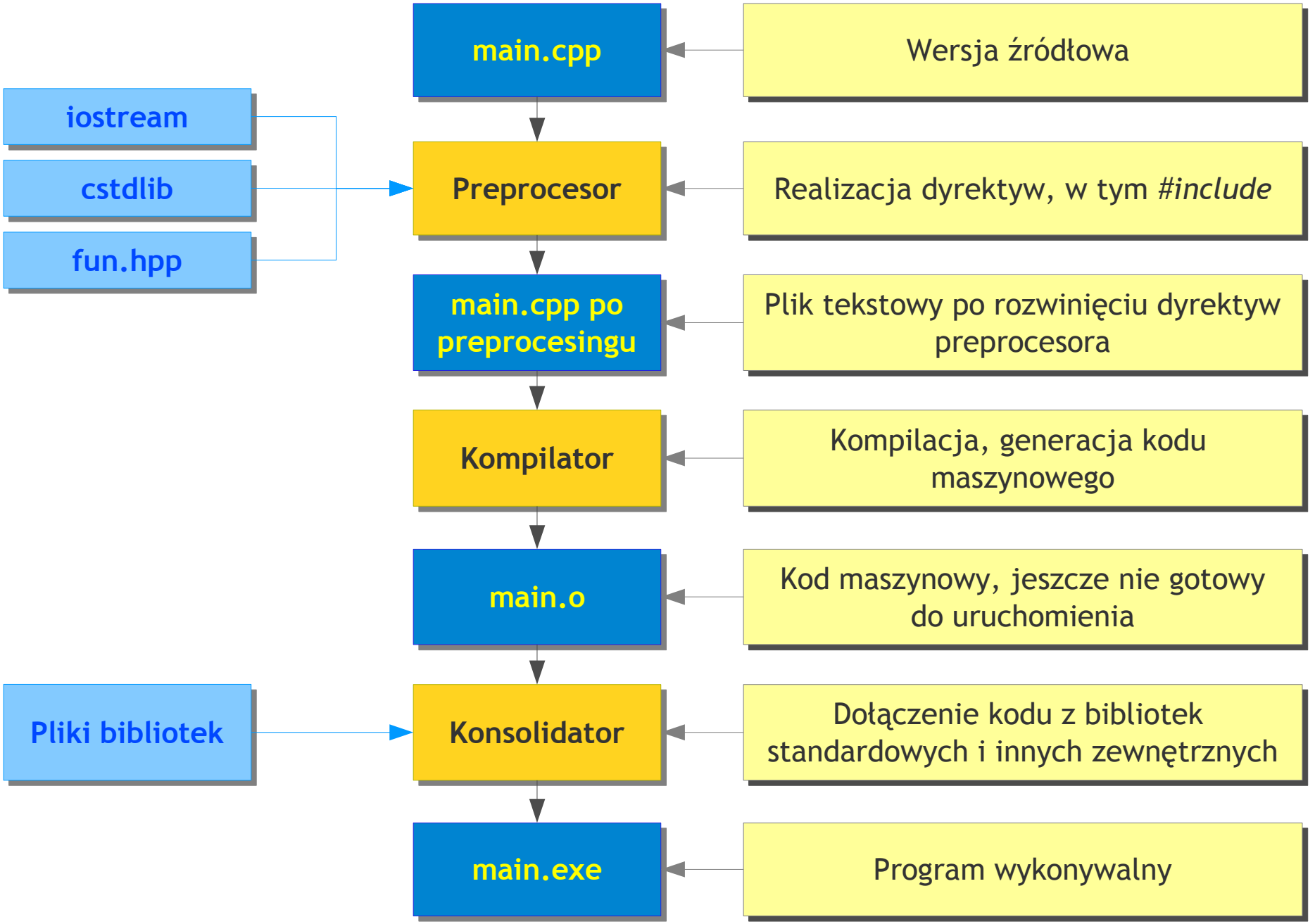
Po włączeniu pliku nagłówkowego *fun.hpp* deklaracje tych funkcji są znane kompilatorowi

Program w włączonym plikiem nagłówkowym skompiluje się poprawnie, ale nie powstanie wersja wykonywalna. Dlaczego?

Kompilator, konsolidator... przypomnienie

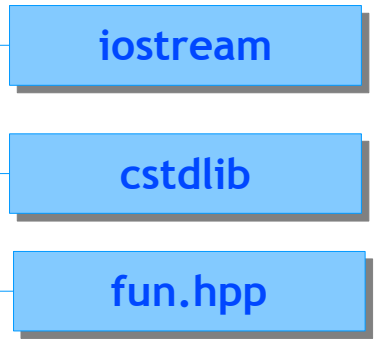


Kompilacja + konsolidacja dla pojedynczego modułu

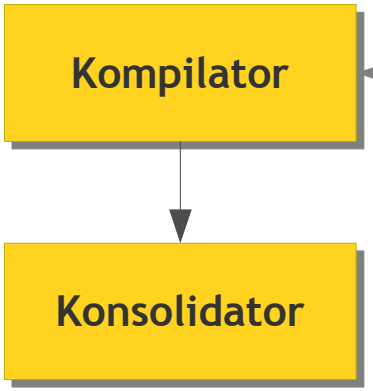


Uwaga — pliki nagłówkowe są „niewidoczne” dla kompilatora!

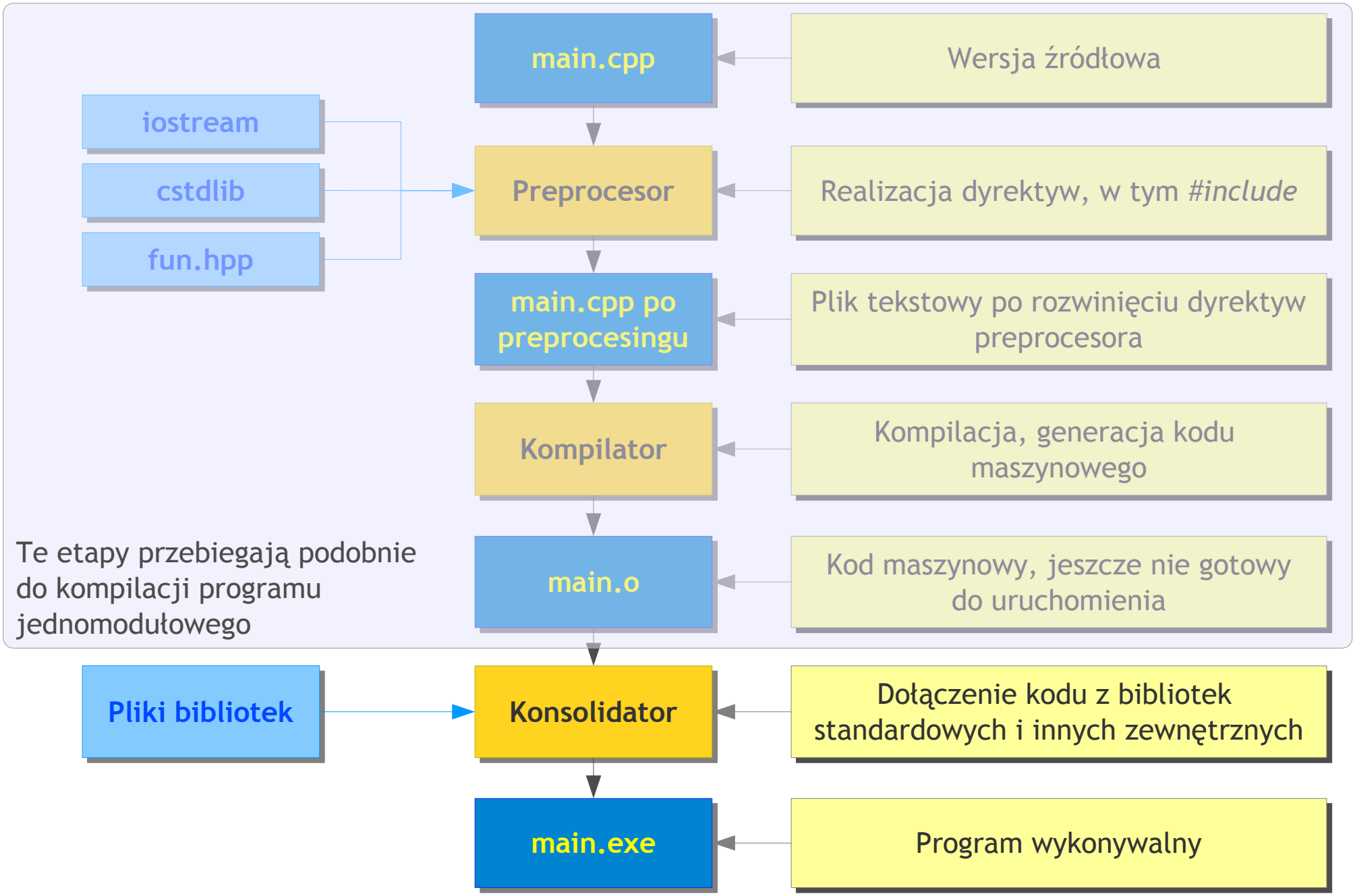
```
. . .
#include "fun.hpp"

int main()
{
  . . .
  cout << pole_kwadratu( 10 ) << endl;
  cout << obwod_kwadratu( 10 ) << endl;
  cout << pole_kola( 10 ) << endl;
  cout << obwod_kola( 10 ) << endl;
  . . .
  return EXIT_SUCCESS;
}
```



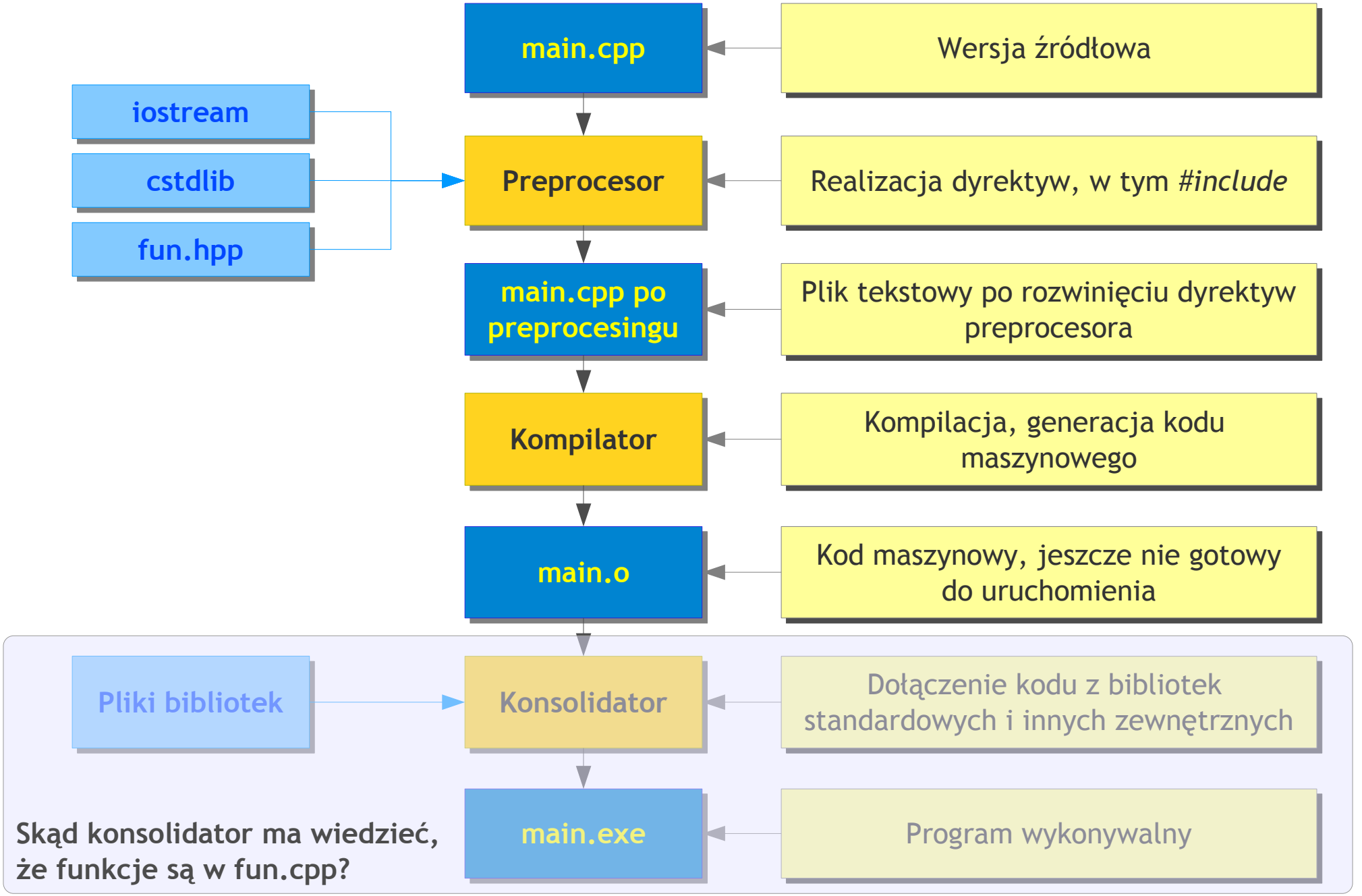

```
. . .
double pole_kwadratu( double bok );
double obwod_kwadratu( double bok );
double pole_kola( double promien );
double obwod_kola( double promien );

int main()
{
  . . .
  cout << pole_kwadratu( 10 ) << endl;
  cout << obwod_kwadratu( 10 ) << endl;
  cout << pole_kola( 10 ) << endl;
  cout << obwod_kola( 10 ) << endl;
  . . .
  return EXIT_SUCCESS;
}
```



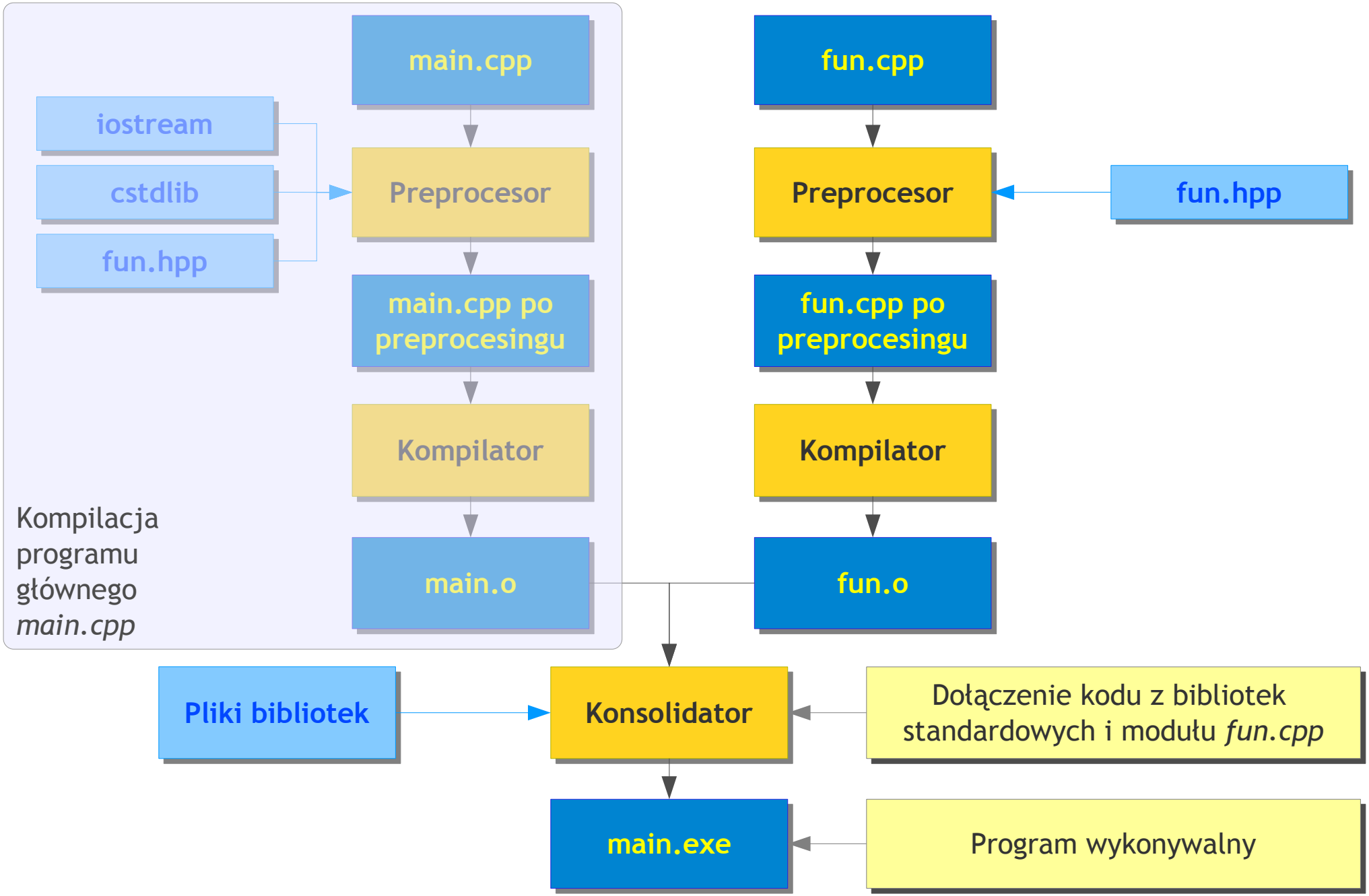
Kompilacja + konsolidacja dla programu wielomodułowego — problem



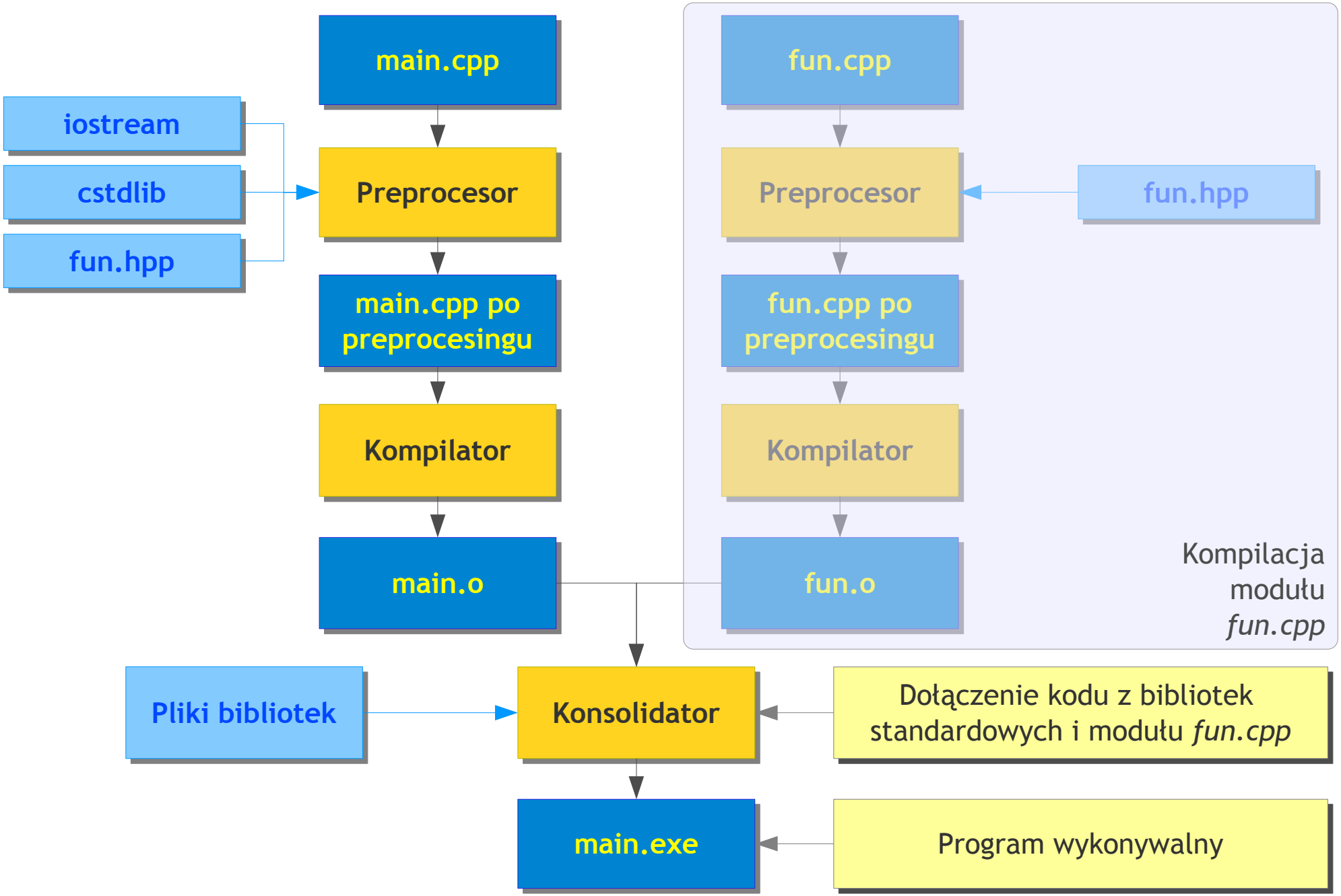
Kompilacja + konsolidacja dla programu wielomodułowego — problem



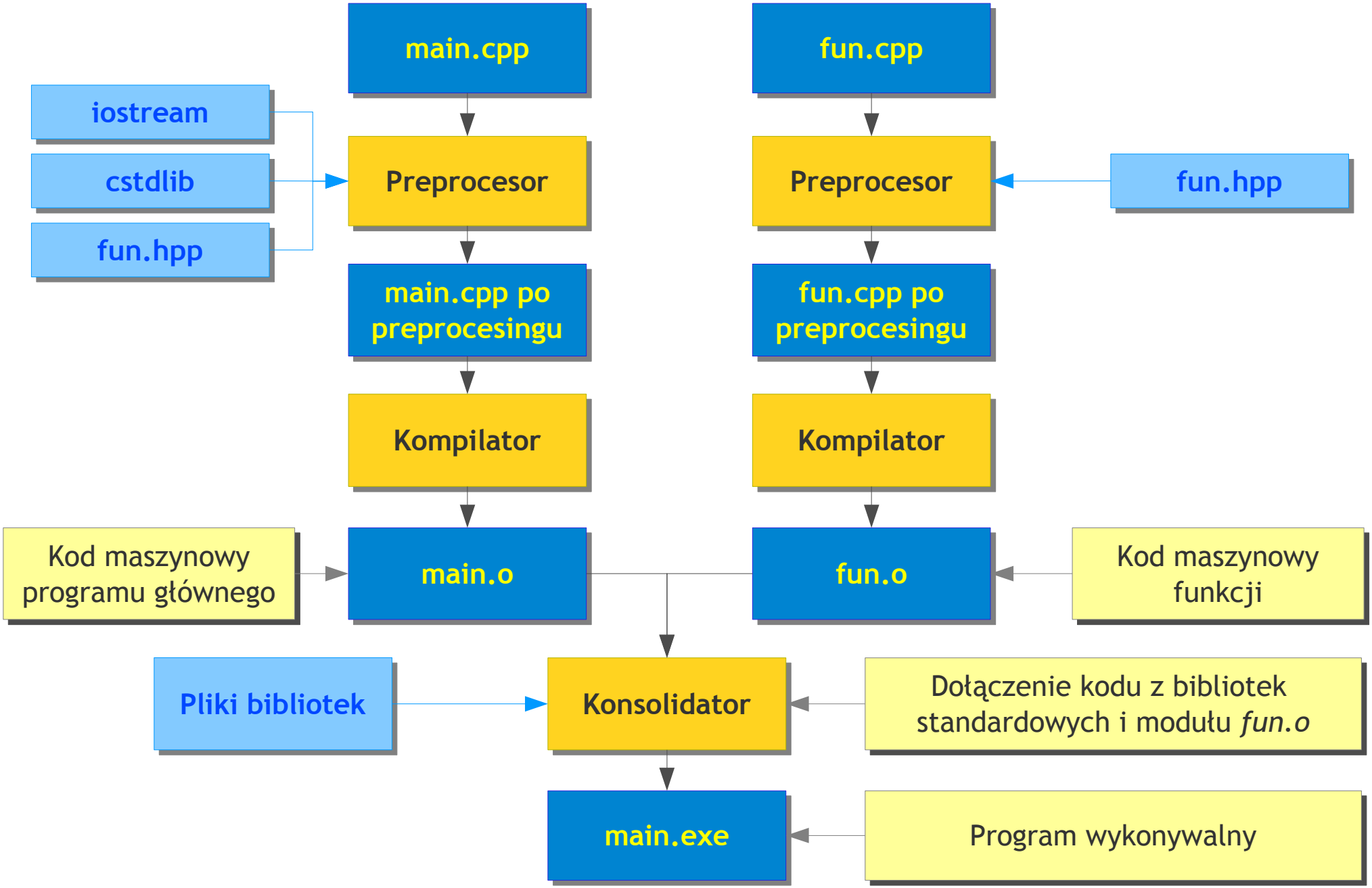
Kompilacja rozłączna — koncepcja



Kompilacja rozłączna — koncepcja



Kompilacja rozłączna — koncepcja



Kompilacja rozłączna — manualne zarządzanie kompilacją

Załóżmy, że kompilator dostępny jest poleceniem *cc* (jak w systemach Unixowych) i zmienne środowiskowe są ustalone.

```
cc test.cpp
```

- Wywołanie to spowoduje:
  - kompilacje programu *test.cpp*,
  - wygenerowanie pliku pośredniego *test.o* (w przypadku braku błędów),
  - połączenie *test.o* z plikami bibliotek i generacja i wynikowego pliku *a.out*.

- Użycie flagi *-o* umożliwia określenie nazwy pliku wynikowego (np. *uruchom*).

```
cc test.cpp -o uruchom
```

- Użycie flagi *-c* pozwala na samą kompilację (bez konsolidacji programu).

```
cc -c test.cpp
```

Powstaje tylko plik pośredni *test.o*.

Kompilacja rozłączna — manualne zarządzanie kompilacją

- Załóżmy, że nasz program składa się z modułów:
  - *main.cpp* — program główny z funkcją *main*,
  - *fun.cpp* — moduł z definicjami funkcji.

- Kompilujemy oddzielnie poszczególne moduły:

```
cc -c main.cpp
```

```
cc -c fun.cpp
```

Powstają pliki pośrednie *main.o* oraz *fun.o*. Łączymy je w program wykonywalny o nazwie uruchom:

```
cc fun.o main.o -o uruchom
```

- W przypadku modyfikacji pliku *fun.cpp* wystarczy:

```
cc -c fun.cpp
```

```
cc fun.o main.o -o uruchom
```

Kompilacja rozłączna — automatyzacja

- *make* — program realizujący niezbędne kompilacje w oparciu o reguły zdefiniowane w pliku sterującym (domyślna nazwa — *Makefile*):
  - plik ten opisuje zależności pomiędzy modułami programu oraz określa jaki wywołania powinny zostać wykonane dla każdego z modułów,
  - program *make* sprawdza czasy modyfikacji plików źródłowych i docelowych i przeprowadza jedynie niezbędne kompilacje.

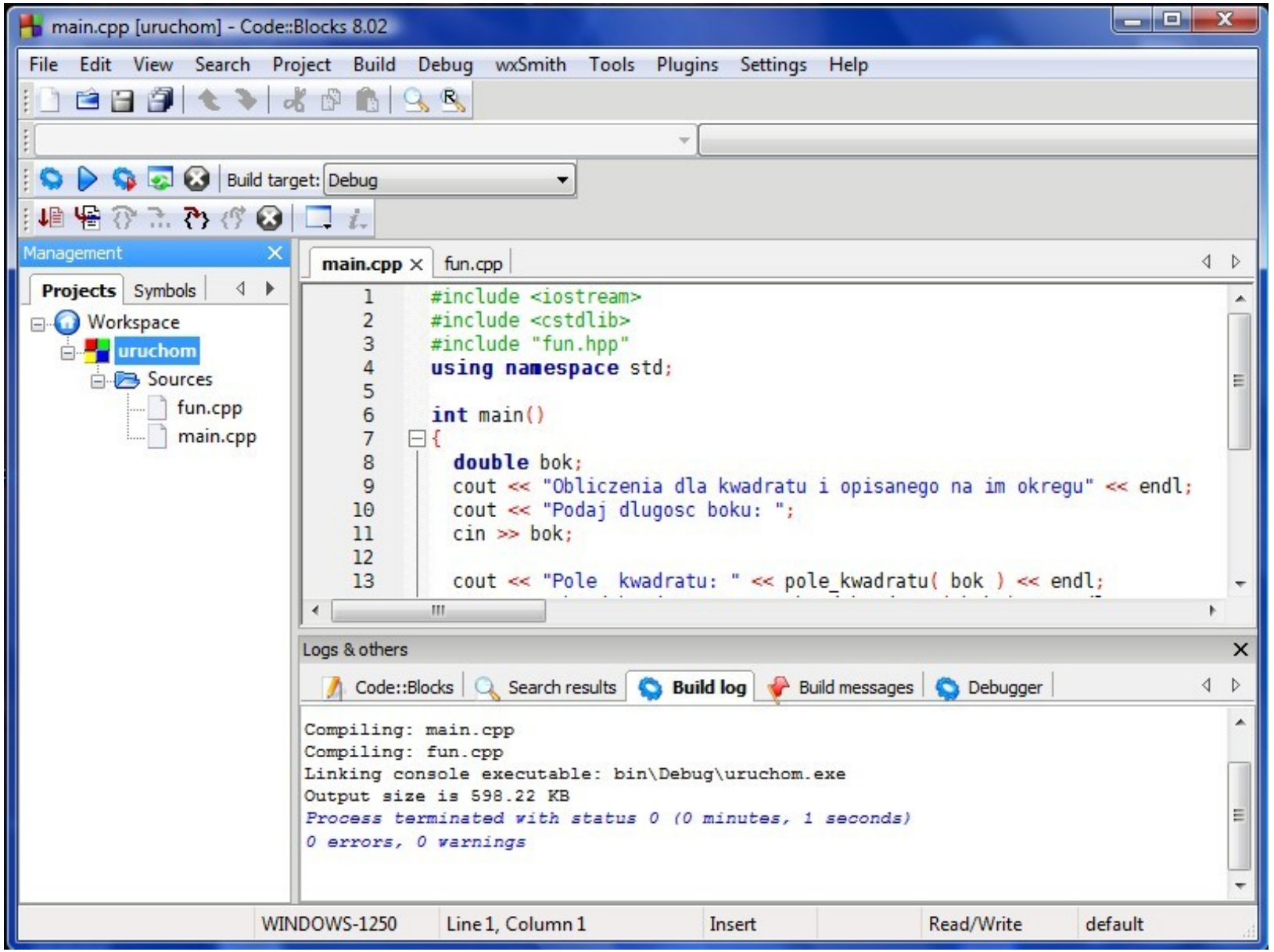
Prosty *Makefile* dla omawianego przykładu:

```
uruchom : fun.o main.o
    cc fun.o main.o –o uruchom
fun.o : fun.cpp fun.hpp
    cc -c fun.cpp
main.o : main.cpp fun.hpp
    cc -c main.cpp
```

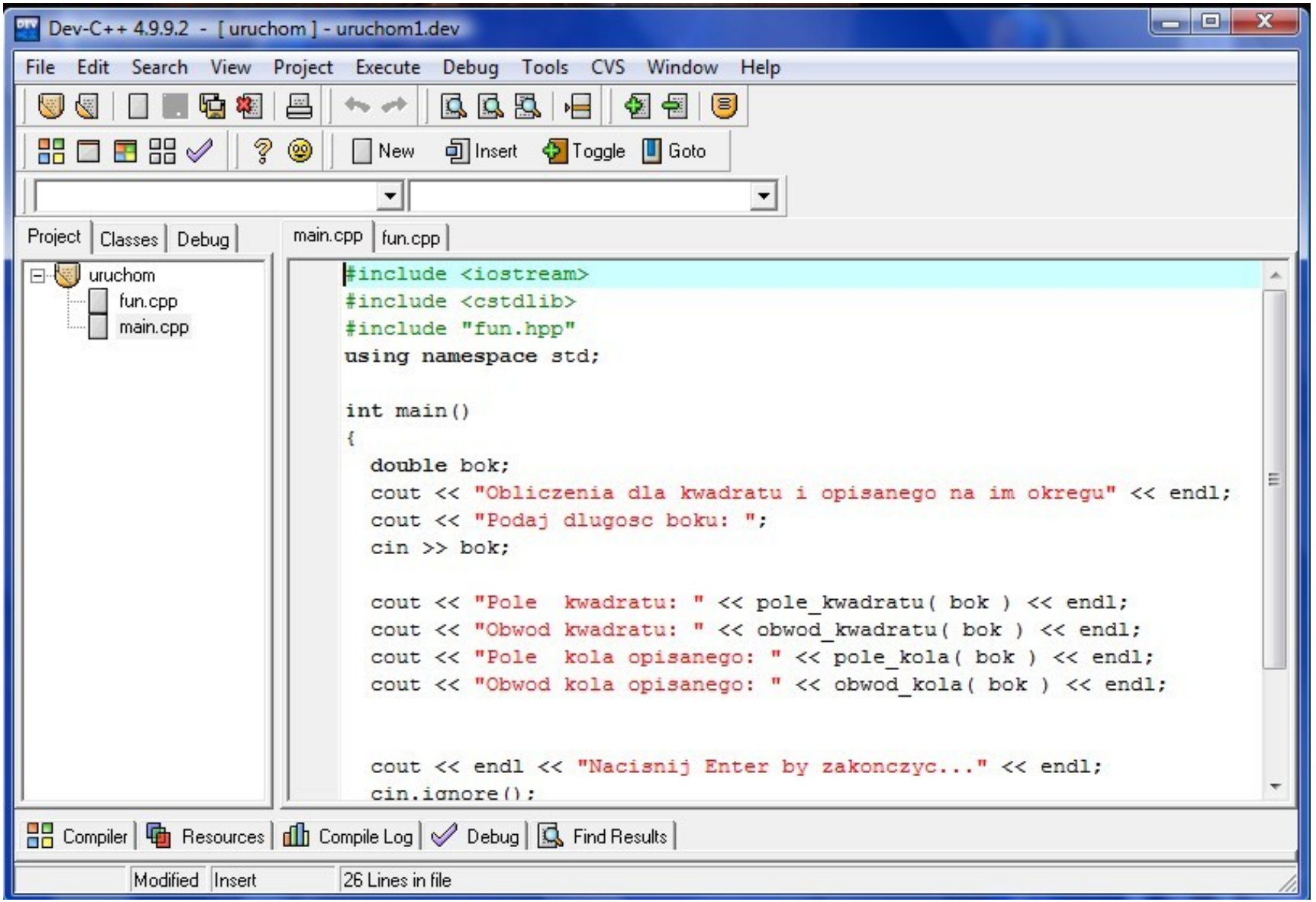
Kompilacja rozłączna — tworzenie projektów w środowiskach IDE

- Większość środowisk programistycznych oferuje możliwość budowania projektów. Projekt określa przynajmniej:
  - pliki źródłowe i pomocnicze wchodzące w skład programu,
  - lokalizację bibliotek, plików nagłówkowych,
  - opcje kompilacji, typ kodu wynikowego,
  - oraz inne, zależne od implementacji, cechy i parametry programu.

Kompilacja rozłączna — tworzenie projektów w Code::Blocks



Kompilacja rozłączna — tworzenie projektów w Dev-C++



Funkcje w osobnym module — włączanie własnego pliku nagłówkowego

Część implementacyjna : *fun.cpp*

```
#include "fun.hpp"



double pole_kwadratu( double bok )
{
   return bok * bok;
}

double obwod_kwadratu( double bok )
{
  return 4 * bok;
}

double pole_kola( double promien )
{
  return 3.14 * promien * promien;
}

double obwod_kola( double promien )
{
  return 2 * 3.14 * promien;
}
```

Moduły zwykle włączają własny plik nagłówkowy.
W tym akurat przypadku nie jest to konieczne.

Część publiczna modułu: *fun.hpp*

```
double pole_kwadratu( double bok );
double obwod_kwadratu( double bok );
double pole_kola( double promien );
double obwod_kola( double promien );
```

Funkcje w osobnym module — włączanie innych plików nagłówkowych

Część implementacyjna : *fun.cpp*

```
#include "fun.hpp"

#include <cmath>

double pole_kwadratu( double bok )
{
   return bok * bok;
}

double obwod_kwadratu( double bok )
{
  return 4 * bok;
}

double pole_kola( double promien )
{
  return M_PI * promien * promien;
}

double obwod_kola( double promien )
{
  return 2 * M_PI * promien;
}
```

Włączenie pliku nagłówkowego zawierającego definicję stałej π : *M_PI*

Część publiczna modułu: *fun.hpp*

```
double pole_kwadratu( double bok );
double obwod_kwadratu( double bok );
double pole_kola( double promien );
double obwod_kola( double promien );
```

Funkcje w osobnym module — włączanie innych plików nagłówkowych

Część publiczna modułu: *fun.hpp*

```
#ifndef _fun_hpp_
#define _fun_hpp_

double pole_kwadratu( double bok );
double obwod_kwadratu( double bok );
double pole_kola( double promien );
double obwod_kola( double promien );

#endif
```

Dyrektywa umożliwiająca kompilację warunkową

Próba włączenia pliku wiele razy:

```
#include <iostream>
#include <cstdlib>
#include "fun.hpp"
.
.
.
#include "fun.hpp"
.
.
.
#include "fun.hpp"
```

Prototypy funkcji zostaną włączone jeden raz

Co może eksportować moduł?

- Stałe, jako symbole preprocesora:

```
#define KEY_UP    0x48
#define KEY_DOWN  0x50
#define KEY_LEFT  0x4b
#define KEY_RIGHT 0x4d
```

- Typy wyliczeniowe:

```
enum ctrl_key_codes
{
  KEY_UP    =  0x48,
  KEY_DOWN  =  0x50,
  KEY_LEFT  =  0x4b,
  KEY_RIGHT =  0x4d
};
```

- Nazwy typów zdefiniowanych przez programistę:

```
typedef unsigned char         byte;
typedef unsigned short int    word;
typedef unsigned long int     counter_t;
```

Co może eksportować moduł? cd... .

- Definicje zmiennych *const*:

```
const int MAKS_PREDK_ZABUD = 50;
const double MOJE_PI = 3.14;
```

- Prototypy funkcji:

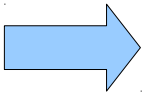
```
double pole_kwadratu( double bok );
double obwod_kwadratu( double bok );
double pole_kola( double promien );
double obwod_kola( double promien );
```

Uwaga, jeżeli moduł eksportuje definicje typów czy stałych, zwykle trzeba zabezpieczać plik nagłówkowy przed wielokrotnym włączaniem w tym samym zakresie.

Zabezpieczenie przed wielokrotnym włączaniem pliku nagłówkowego

- Dyrektywa kompilacji warunkowej:

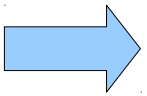
```
#ifndef _fun_hpp_
#define _fun_hpp_

const double MOJE_PI = 3.14;

double pole_kwadratu( double bok );
. . .
#endif
```



```
#include "fun.hpp"
.
.
#include "fun.hpp"
.
.
#include "fun.hpp"
```

- Gdyby nie było tej dyrektywy:

```
const double MOJE_PI = 3.14;

double pole_kwadratu( double bok );
. . .
```



```
#include "fun.hpp"
.
.
#include "fun.hpp"
.
.
#include "fun.hpp"
```

```
Message
error: redefinition of `const double MOJE_PI'
error: `const double MOJE_PI' previously defined here
error: redefinition of `const double MOJE_PI'
error: `const double MOJE_PI' previously defined here
```

Eksport zmiennej zewnętrznej

Uwaga, jeżeli moduł ma eksportować zmienną, należy w odpowiedni sposób to opisać.

- *Deklaracja zmiennej* — informacja o *typie* i *nazwie zmiennej*, nie musi zawierać informacji o klasie pamięci i wartości incjalizującej.
- *Definicja zmiennej* — to deklaracja zawierająca informacje o *klasie pamięci* i wartości inicjalizującej.

Definicja zmiennej występuje *raz i rezerwuje pamięć dla zmiennej*, zgodna z definicją deklaracja może występować w programie *wiele razy*, ma charakter informacyjny.

Eksport zmiennej zewnętrznej

- Plik nagłówkowy zawiera *deklarację* zmiennej:

```
extern int input_data_error;
```

- Plik implementacyjny zawiera *definicję* zmiennej:

```
int input_data_error;
```

```
#ifndef _fun_hpp_
#define _fun_hpp_

extern int input_data_error;

double pole_kwadratu( double bok );
double obwod_kwadratu( double bok );
double pole_kola( double promien );
double obwod_kola( double promien );

#endif
```

```
#include "fun.hpp"
#include <cmath>

int input_data_error;

double pole_kwadratu( double bok )
{
   return bok * bok;
}

. . .
```

Każda funkcja i zmienna zewnętrzna jest domyślnie eksportowana

Każda funkcja i zmienna zewnętrzna występująca w module może być dostępna dla innych modułów programu. Czasem chcemy jednak ograniczyć dostęp do takich zmiennych i funkcji — uprywatnić je w obrębie modułu.

- Jeżeli definicja zmiennej zewnętrznej lub funkcji zawiera słowo *static*, nie są one dostępne dla innych modułów programu:

```
static int private_error_flag;
. . .
static void private_error_handler( void )
{
 . . .
}
```

- Nazwy takich zmiennych i funkcji mogą się powtarzać w innych modułach programu.