Języki i metody programowania I

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 2012-10-04

Informacje o przedmiocie

Zakres przedmiotu 1

Przedmiot	Semestr	Zakres	Forma zaliczenia
Języki i metody programowania I	1	Język C	Ocena końcowa = ocena z laboratorium
Języki i metody programowania II	2	Język C++	Egzamin
Programowanie obiektowe	3	Język Java	Egzamin

Zakres przedmiotu 2

Czego powinniście Państwo nauczyć się?

- **Składnia** języka C
  - deklarowanie zmiennych
  - definiowanie funkcji
  - stosowanie instrukcji sterujących
  - zasady konstrukcji wyrażeń i ich interpretacji deklaracji typów danych
- **Semantyka** - zasady odwzorowania konstrukcji języka C w elementy wykonywalnego programu
  - lokalizacja zmiennych w pamięci
  - wykonanie instrukcji i interpretacja wyrażeń
  - przebieg wywołania funkcji,
  - sposób przekazywania parametrów

Zakres przedmiotu 3

- Podstawowe **funkcje standardowych bibliotek** języka C
- Przebieg **procesu tworzenia programu** (zastosowanie preprocesora, kompilacja i konsolidacja)
- Zasady konstruowania programów **wielomodułowych**
- Przetwarzanie **plików** oraz programy komunikujące się poprzez standardowe wejście i wyjście
- **Uruchamianie programów** (usuwanie błędów kompilacji, konsolidacji i wykonania)
- **Dynamiczna alokacja pamięci** - implementacja struktur danych typu tablice o zmiennym rozmiarze i listy

Czego nie będzie?

- Grafiki
  - nieprzenośna i uzależniona od platformy wykonania
- Graficznego interfejsu użytkownika (GUI)
  - interfejsy okienkowe można wydajnie realizować z użyciem gotowych bibliotek obiektowych dla języka C++, np. Qt, MFC
- Programy będą wykonywane na konsoli i mogą przekłamywać polskie znaki (przynajmniej w systemie Windows).

Literatura

- Brian Kernighan, Denis Ritchie – Język ANSI C, WNT, 2004
- K.N. King. Język C. Nowoczesne programowanie. Wydanie II, Helion 2011
- Składnia: http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

Język C
1. Wprowadzenie

Historia

- Jest produktem ubocznym rozwoju systemu UNIX w Bell Labs (od 1969)
- Jeden z członków zespołu zdecydował się na zastąpienie asemblera językiem wyższego poziomu nazwanym B
- Ritche rozpoczął programowanie systemu w B; język stopniowo ewoluował i zmienił nazwę na C
- W 1973 roku system UNIX został przepisany w języku C

Historia standaryzacji

- 1978 – pojawia się książka Keringhan&Ritchie *The C Programming Language*
- 1989 opublikowany standard ANSI (American National Standards Institute)
- 1990 standard ISO (International Organization for Standardization)
- 1999 – rozszerzenia nowy standard ISO (najważniejsze zmiany dotyczą rozszerzenia zestawów znaków)

Cechy języka

- C jest językiem niskiego poziomu
  - Pozwala na operacje na bajtach, bitach, adresach
  - Wiele konstrukcji jest wprost przeniesionych z języka maszynowego
  - Blisko związany z architekturą sprzętu, np. wielkość typu całkowitoliczbowego int odpowiada długości słowa maszynowego
- C jest niewielkim językiem
  - Większość usług przeniesiona do bibliotek funkcji
- Kompilator C jest permisywny
  - Słaba kontrola typów, mało ograniczeń
  - Zakłada, że programista jest świadomy, tego, co robi

Cechy języka - zalety

- Wydajność – duża szybkość i małe zużycie pamięci
- Przenośność – dzięki standaryzacji, bliskim związkom z systemem UNIX, umieszczeniu nieprzenośnych elementów w bibliotekach
- Ekspresywność – możliwość definiowania dowolnych typów danych i funkcji
- Elastyczność – konstrukcje C mogą być bardzo oszczędne, np. odejmowanie liczb i znaków…
- Standardowa biblioteka – zbiór użytecznych funkcji
- Integracja z systemem UNIX (Linux)

Cechy języka - wady

- C jest językiem podatnym na błędy (adresy, liczby, typy logiczny mogą być mieszane)
- Programy w C mogą być trudno zrozumiałe:

  Czy to naprawdę rozwiązanie problemu 8 hetmanów?

```
v,i,j,k,l,s,a[99];
main()
{
  for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&
(!k&&!!printf(2+"\n\n%c"-(!l<<!j)," #Q"[l^v?(l^j)&1:2])&&
++l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),v||(i==j?a[i+=k]=0:
++a[i])>=s*k&&++a[--i])
    ;
}
```

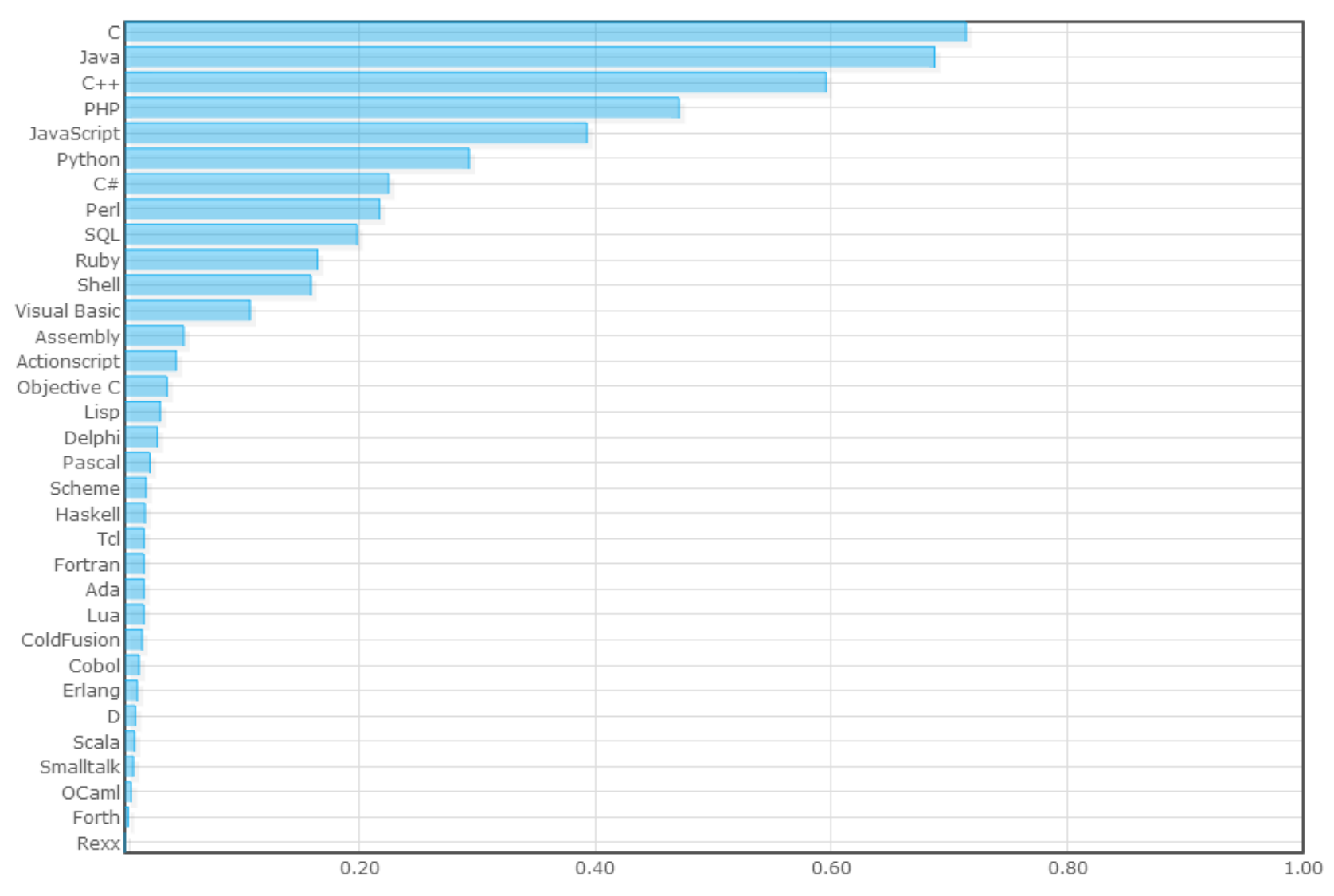
- Programy w C mogą być kłopotliwe w modyfikacji (mało mechanizmów organizujących kod)

Gdzie stosuje się C ?

- Systemy różnej skali – od wbudowanych do dużych systemów
- Oprogramowanie systemów operacyjnych (Linux)
- Przenośne biblioteki (np. algorytmów kompresji, manipulacja formatami obrazów)
- Środowiska wykonawcze (np. maszyna wirtualna Java)

Jak często stosuje się C?

Statystyki z http://langpop.com/ (04-10-2012)



Następcy C

Język stał się źródłem inspiracji dla takich języków jak:

- C++ (nadzbiór C)
- Java
- C# (C-sharp)
- JavaScript
- Perl
- Php
- Python

Podręcznikowy przykład

```
/* hello.c */

#include <stdio.h>

int main   (   )
{
       printf(   ”Hello world\n”   ) ;
       return 0;
}
```

```
> gcc hello.c
> ./a.out
Hello world
>
```

Proces translacji języka interpretacja vs. kompilacja

Interpreter

- Analizuje kolejne instrukcje programu
- Tłumaczy na kod wykonywalny.
- Wykonuje go

- **Wady**
  - Wymagany jest osobny program (interpreter). Interpreter zużywa dostępne zasoby (pamięć).
  - Wykonanie jest znacznie wolniejsze. W przypadku nawrotów ta sama instrukcja jest analizowana i tłumaczona wielokrotnie.
  - Kod programu zazwyczaj musi być w całości załadowany, co ogranicza jego rozmiary.
  - Interpretery dla różnych platform mogą różnić się miedzy sobą, co ogranicza przenośność.
- **Zalety**
  - Łatwa identyfikacja miejsca wystąpienia błędów.
  - Szybkie tworzenie i uruchamianie oprogramowania.

Proces translacji języka interpretacja vs. kompilacja

- **Kompilator**
  - Kod programu jest tłumaczony do postaci kodu maszynowego.
  - Fragmenty programu mogą być umieszczone w odrębnych plikach i kompilowane osobno.
  - Wynikowy kod jest następnie łączony w program wykonywalny przez *konsolidator* (ang.: linker).
- **Zalety**
  - Kod stworzony przez kompilator jest zazwyczaj mniejszy i wymaga mniejszej ilości zasobów (np.: pamięci) w trakcie wykonania.
  - Kod może być wykonywany znacznie szybciej.
  - Możliwe jest tworzenie znacznie większych programów, złożonych z wielu plików źródłowych.
  - Możliwa jest kompilacja skrośna.

Proces translacji języka interpretacja vs. kompilacja

- **Kompilator - wady**
  - Wymagane są odrębne programy (kompilator, linker).
  - Proces kompilacji zajmuje czas i może wymagać środowiska o dużych zasobach (pamięć, prędkość procesora).
  - Proces śledzenia i usuwania błędów jest bardziej skomplikowany.
    Część błędów jest identyfikowana przez kompilator i linker.
    Usuwanie błędów wykonania wymaga użycia odrębnego programu: *debuggera*.
  - Kompilatory zazwyczaj narzucają silne ograniczenia (typizacja zmiennych, konieczność deklaracji zmiennych i funkcji).

Proces translacji języka interpretacja vs. kompilacja

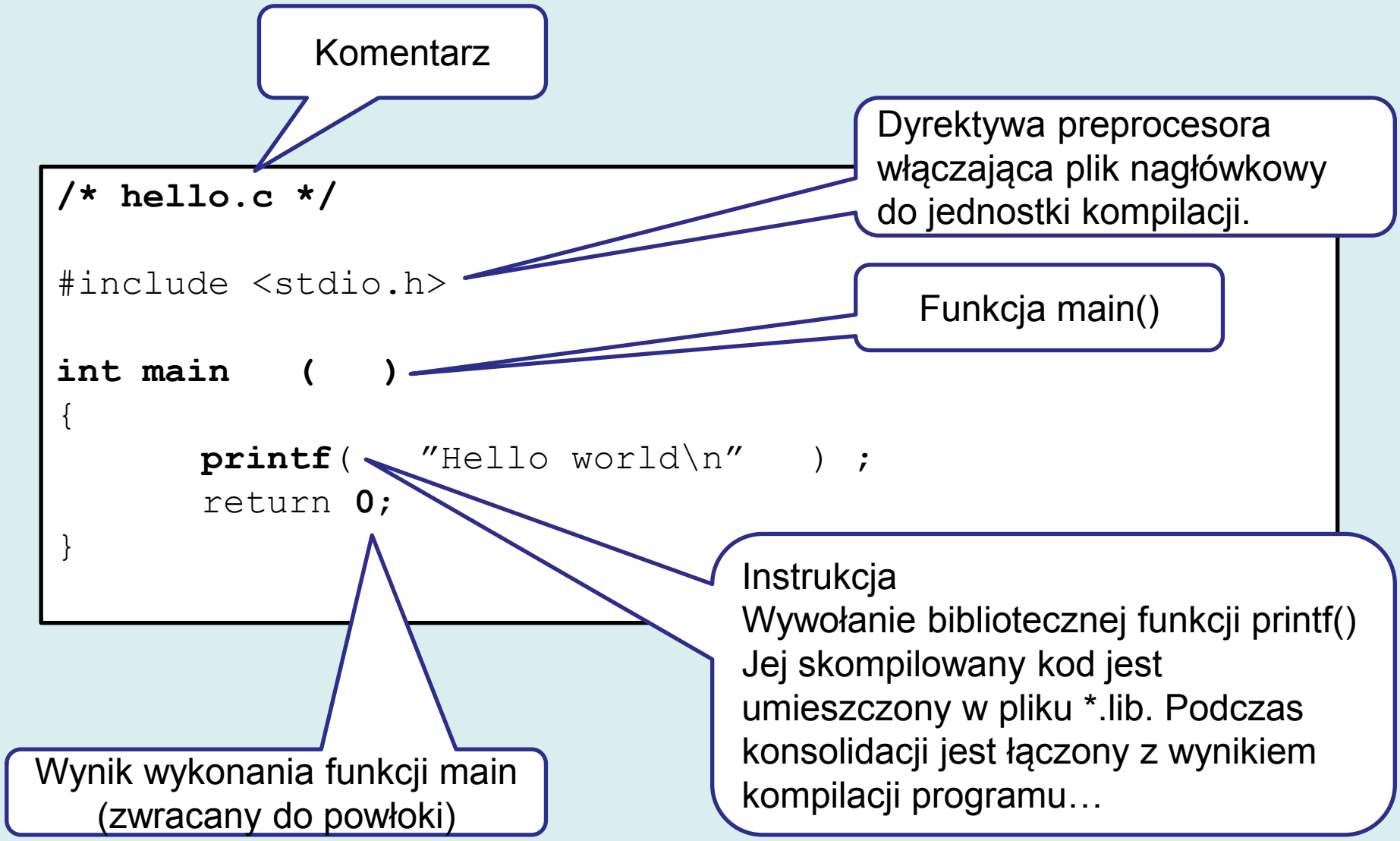
Rozwiązania mieszane

- Współczesne interpretery umożliwiają wstępną kompilację kodu do postaci pośredniej, co znacznie przyspiesza wykonanie.
- Współczesne kompilatory pozwalają na osadzenie pełnej informacji o kodzie źródłowym w programie wykonywalnym i krokowe śledzenie wykonania.
- W przypadku kompilacji skrośnej oferowane są symulatory.

Fazy budowy programu

- Preprocesor – włącza pliki nagłówkowe, zamienia symbole stałych na wartości
- Faza I – analiza kodu, podział na podstawowe symbole (tokeny) i budowa drzewa programu
- Opcjonalnie: globalna optymalizacja drzewa (łączenie i usuwanie podobnych fragmentów)
- Faza II – generacja kodu w postaci plików wynikowych (ang.: *object*) *.obj (*.o)
- Opcjonalnie: optymalizacja – łączenie powtarzających się fragmentów kodu maszynowego.
- Konsolidacja plików wynikowych z bibliotekami *.lib i tworzenie kodu wykonywalnego

Analiza kodu



Analiza kodu 2

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("Tydzien ma %d dni\n",7);
    return 0;
}
```

Funkcja printf() zastępuje ciąg znaków **%d** liczbą całkowitą przekazaną, jako argument

Analiza kodu 3

```
#include <stdio.h>

int main(int argc, char** argv) {
    int days;
    days=7;
    printf("Tydzien ma %d dni\n",days);
    return 0;
}
```

Deklaracja zmiennej

Zamiast stałej można przekazać zmienną. Kompilator automatycznie doda kod, który odczyta jej wartość i przekaże do funkcji.

Analiza kodu 4

Deklaracja stałej preprocesora NUMBER_OF_DAYS

```
#include <stdio.h>
#define NUMBER_OF_DAYS 7

int main(int argc, char** argv) {
    int days;
    days=NUMBER_OF_DAYS;
    printf("Tydzien ma %d dni\n",days);
    return 0;
}
```

W wyniku działania preprocesora każde wystąpienie NUMBER_OF_DAYS zostanie zastąpione wartością stałej (7)

Analiza kodu 5

Włączamy plik nagłówkowy math.h, bo tam są informacje o funkcji sin() i definicja stałej M_PI (czyli π)

```
#include <stdio.h>
#include <math.h>

int main(int argc, char** argv) {
    double y;
    y = sin(M_PI/2);
    printf("Sinus 90 stopni = %f\n",y);
    return 0;
}
```

Wywołanie funkcji sin dla argumentu π/2

Wstawienie %f umożliwia wydruk wartości zmiennoprzecinkowej typu **double**

Analiza kodu 6

```
#include <stdio.h>
#include <math.h>

int main(int argc, char** argv) {
    int x=90;
    double y;
    y = sin(2*M_PI*x/360);
    printf("Sinus %d stopni = %f\n", x ,y);
    return 0;
}
```

Deklaracja wraz z inicjalizacją

Kolejnym znacznikom w tekście muszą odpowiadać odpowiednie typy danych %d – int, %f - double

Analiza kodu 7

Deklaracja wraz z inicjalizacją zmiennej typu napis (string)

```
#include <stdio.h>

int main(int argc, char** argv) {
    int x=20;
    char* osoba="Jan Kowalski";
    printf("%s ma %d lat\n",osoba,x);
    return 0;
}
```

Możemy też wypisać tekst stosując znacznik **%s** (zmienna **osoba**)

Analiza kodu 8

Możemy zadeklarować własną funkcję

```
#include <stdio.h>

double doKwadratu(double x)
{
    return x*x;
}

int main(int argc, char** argv) {
    double x=1.27;
    double y;
    y=doKwadratu(x);
    printf("%f do kwadratu rowna sie %f\n",x, y);
    return 0;
}
```

Oraz ją wywołać…

Analiza kodu 9

```
#include <stdio.h>

double doKwadratu(double x)
{
    return x*x;
}

int main(int argc, char** argv) {
    double x=1.27;
    //double y;
    //y=doKwadratu(x);
    printf("%f do kwadratu rowna sie %f\n",x, doKwadratu(x));
    return 0;
}
```

Nie jest konieczne wywołanie etapami. Jako argument funkcji można przekazać rezultat wywołania innej funkcji…

Analiza kodu 10

Funkcja nie musi zwracać wartości.
Jeśli jej nie zwraca wpisujemy **void**

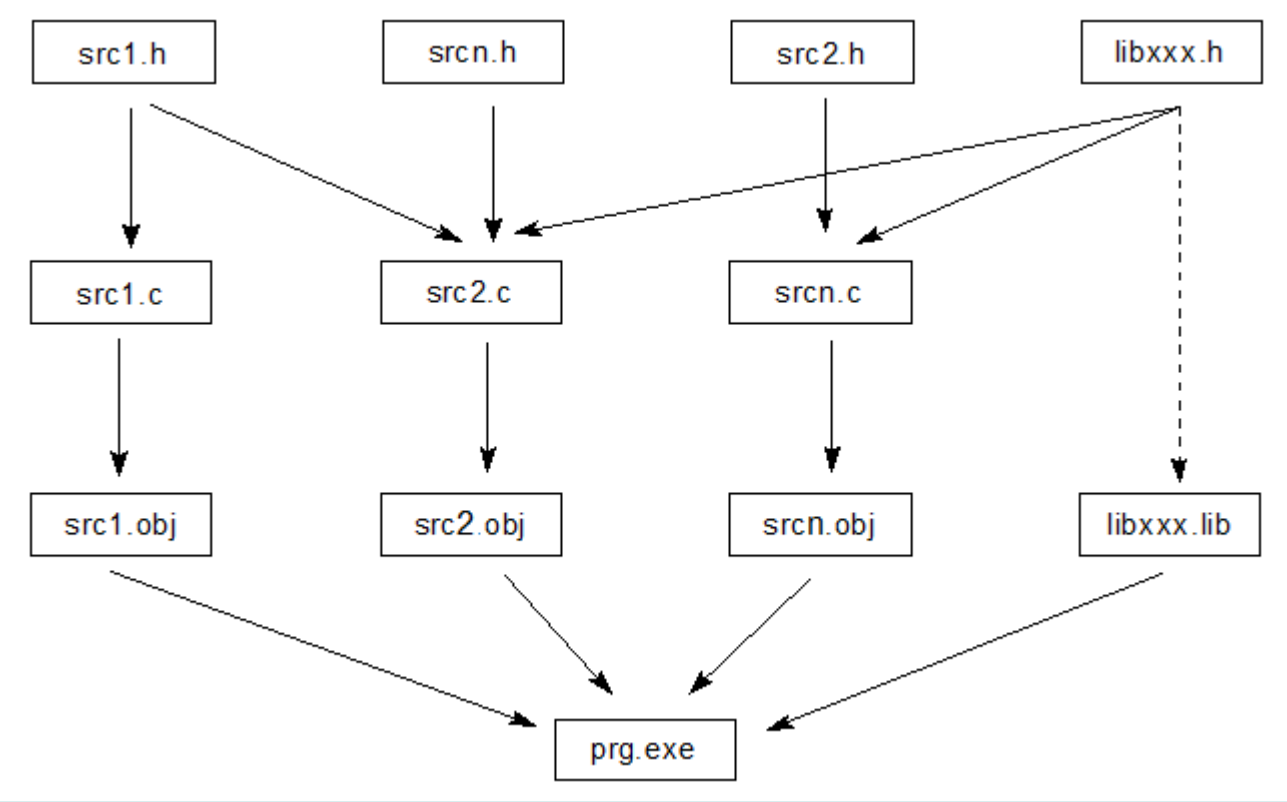
```
#include <stdio.h>

void wypiszKwadrat(double x)
{
    printf("%f do kwadratu rowna sie %f\n",x,x*x);
}

int main(int argc, char** argv) {
    double x=1.27;
    wypiszKwadrat(x);
    return 0;
}
```

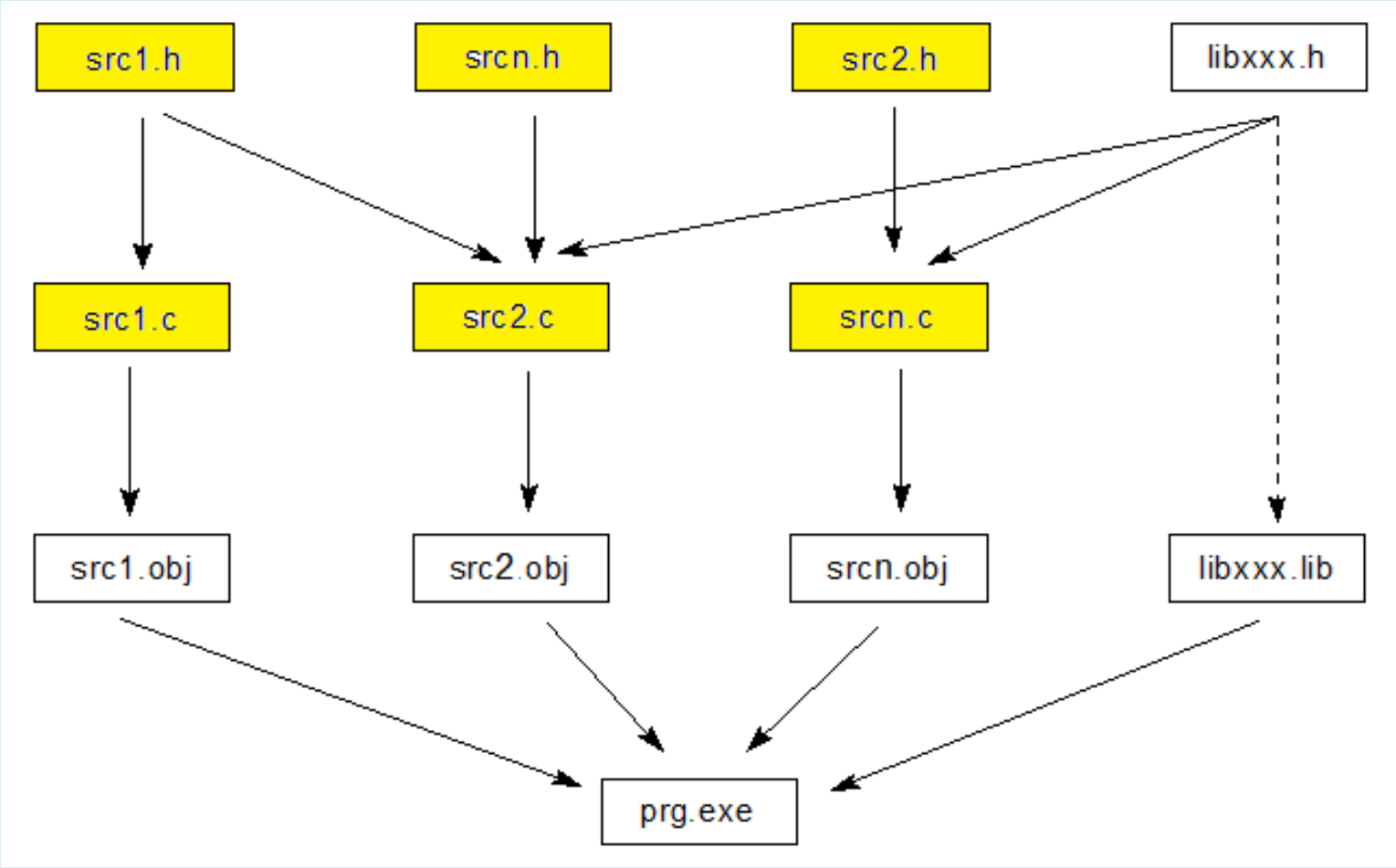
Organizacja kodu 1

- Elementy programu mogą być umieszczone w jednym lub wielu **plikach źródłowych** oraz **bibliotekach**.



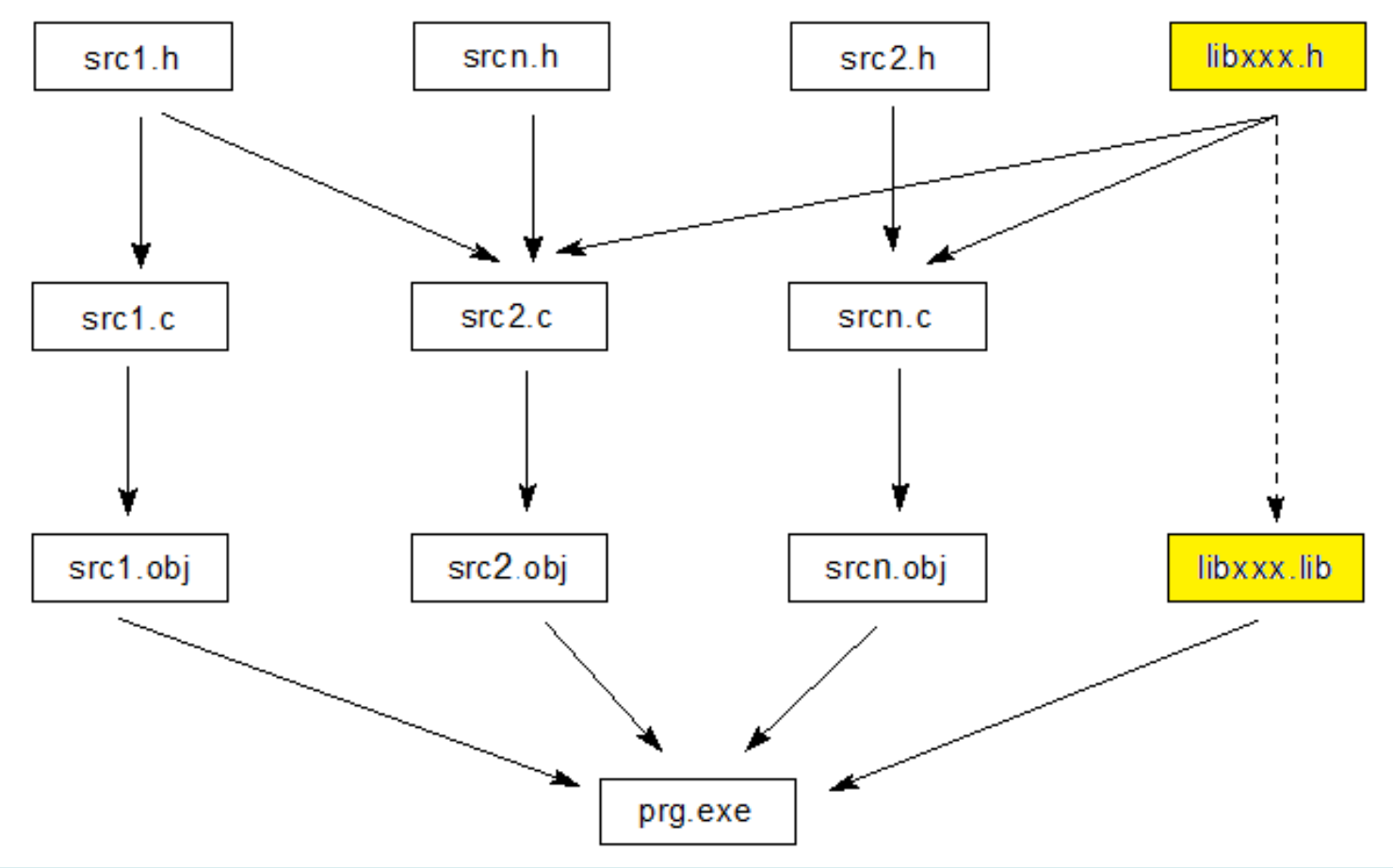
Organizacja kodu 2

- Pliki źródłowe (*.c, *.cpp, *.h) tworzone są przez programistę aplikacji



Organizacja kodu 3

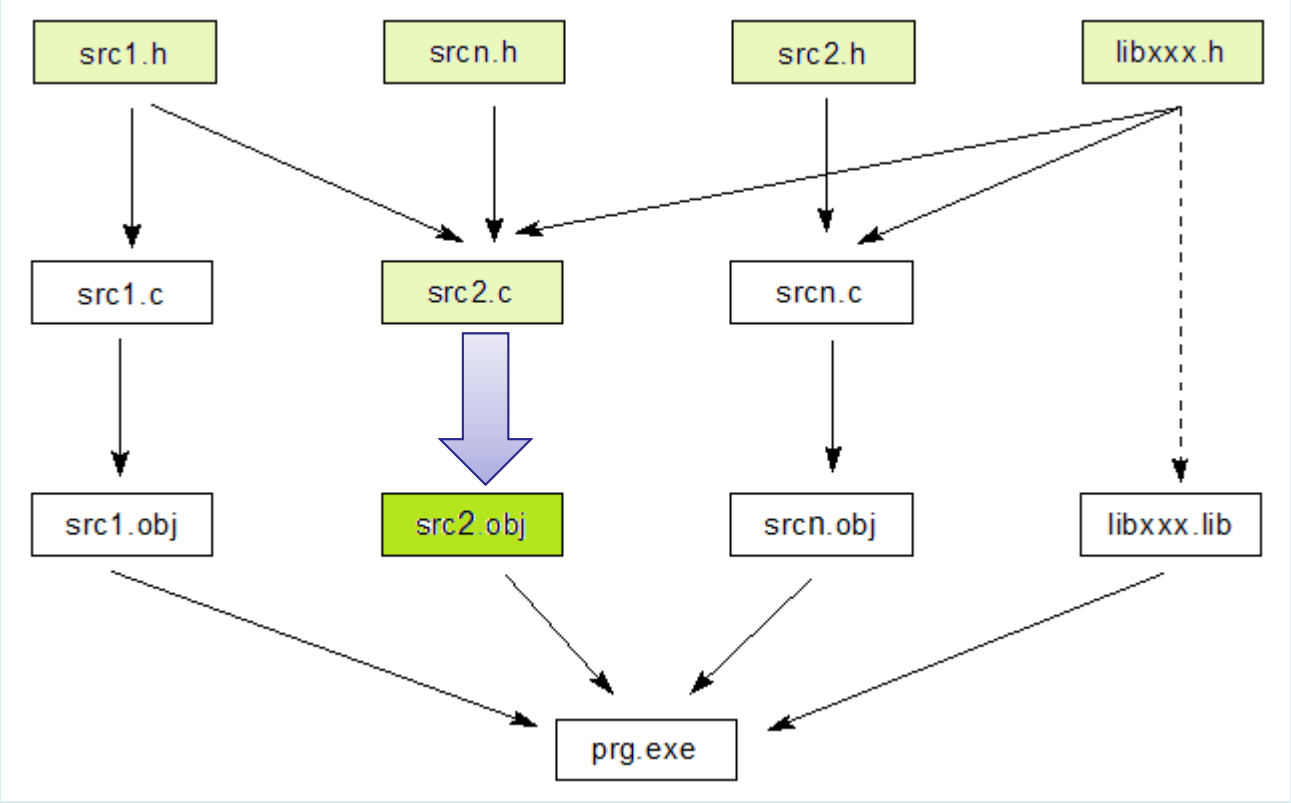
- Pliki biblioteczne wraz z nagłówkami (libxxx.h) najczęściej dostarczane są przez autorów kompilatora.



Organizacja kodu 4

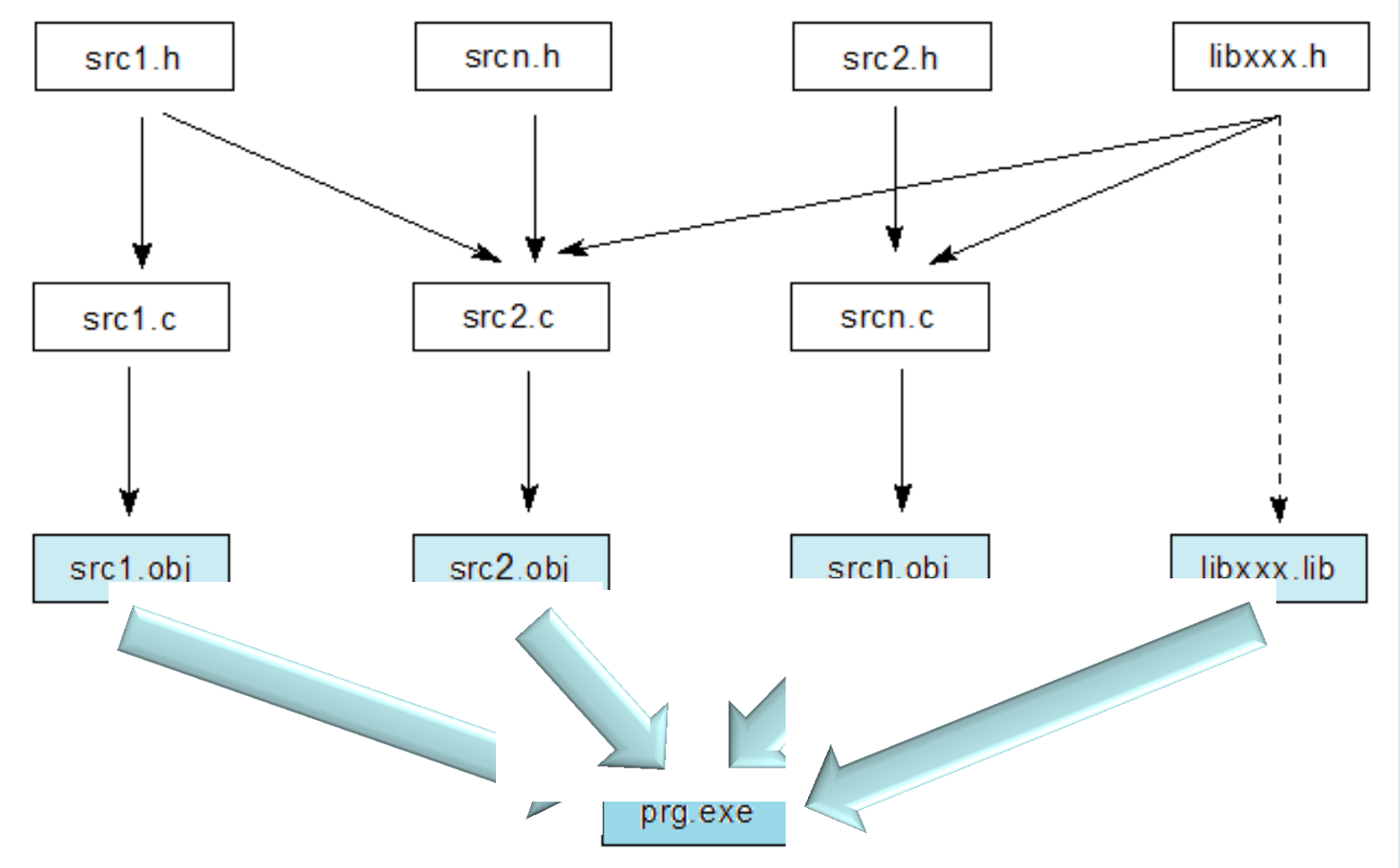
- Podczas kompilacji przetwarzany jest jeden moduł:
  - plik źródłowy *.c
  - wraz z włączonymi plikami nagłówkowymi *.h

Zużycie zasobów jest znacznie mniejsze, niż gdyby poddać kompilacji olbrzymi plik źródłowy złożony ze wszystkich plików składowych.

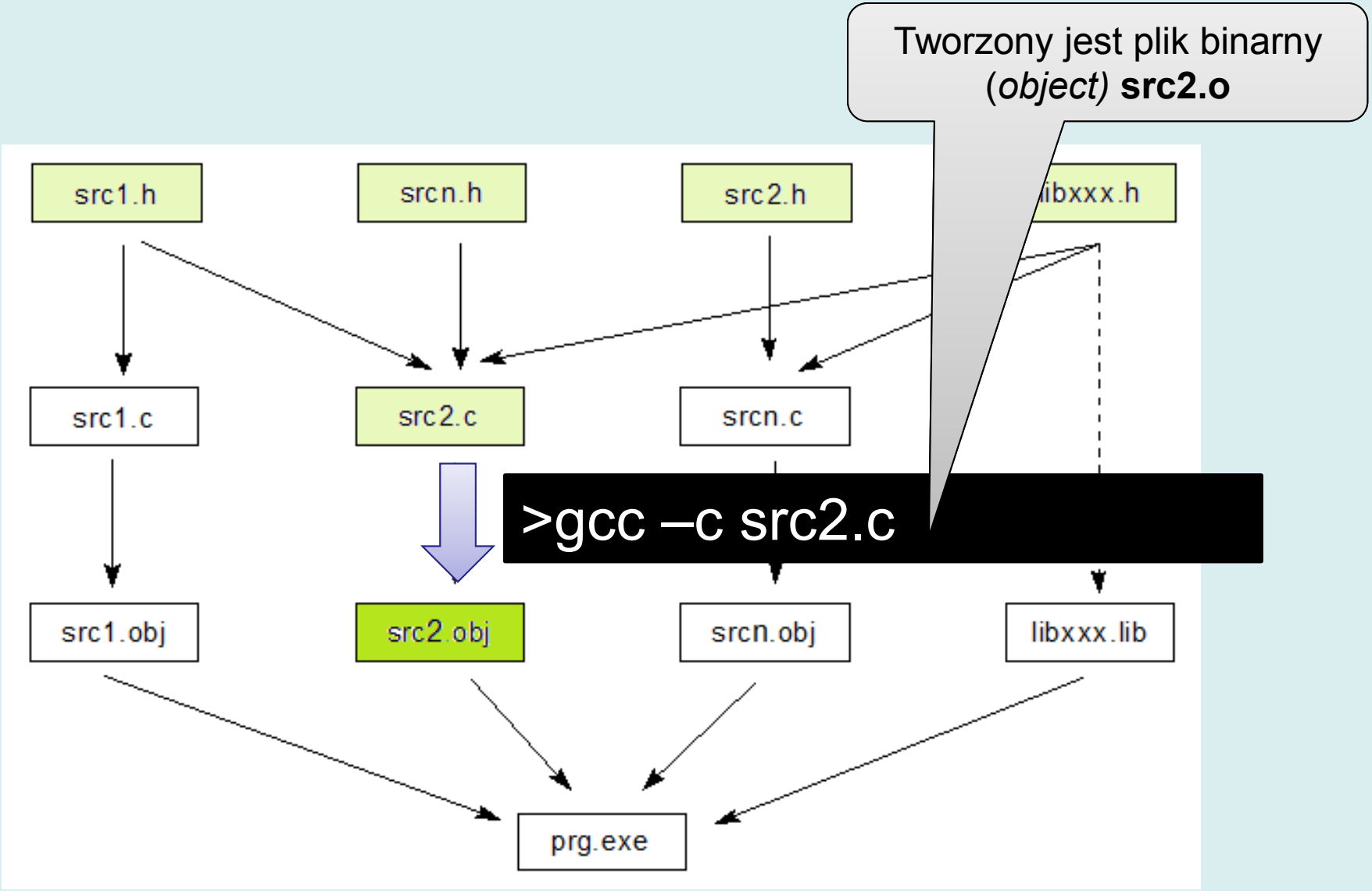


Organizacja kodu 5

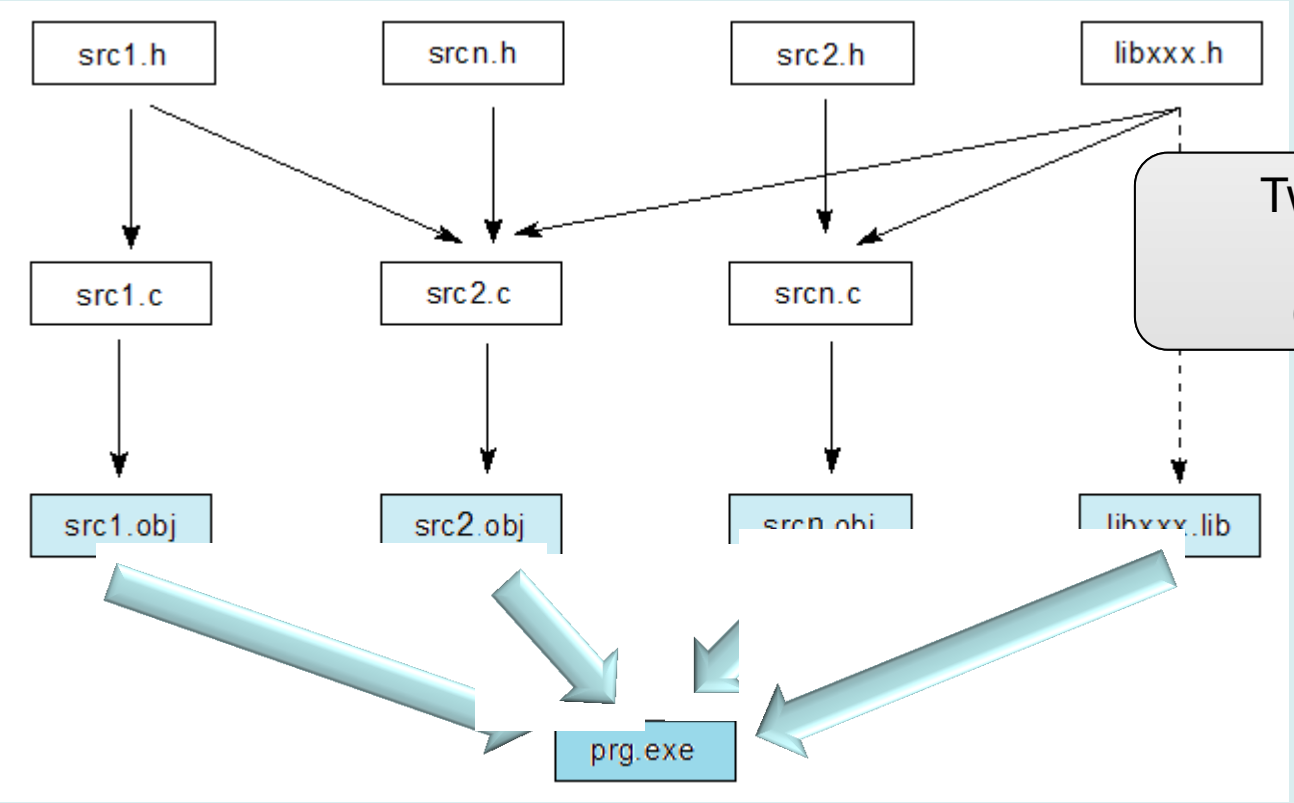
- W wyniku konsolidacji (linkowania) plików *.obj i bibliotek *.lib powstaje kod wykonywalny



Budowa programu wielomodułowego 1



Budowa programu wielomodułowego 2



Tworzony jest plik z programem (wykonywalnym)

```
>gcc src1.o src2.o srcn.o –o –llibxxx prg.exe
```

Budowa programu wielomodułowego 3

- Operacja może być wykonana w jednym wywołaniu:

  gcc scr1.c src2.c srcn.c –llibxx –o prog.exe

- Ale minimalna zmiana w jednym module wymaga rekompilacji wszystkich…
- Zazwyczaj pomocniczą rolę pełni program **make**
  - porównuje czasy plików,
  - jeśli plik wynikowy jest starszy niż źródłowy, buduje wymagany moduł
- Pisanie plików konfiguracyjnych **makefile** dla programu **make** jest dość trudne…

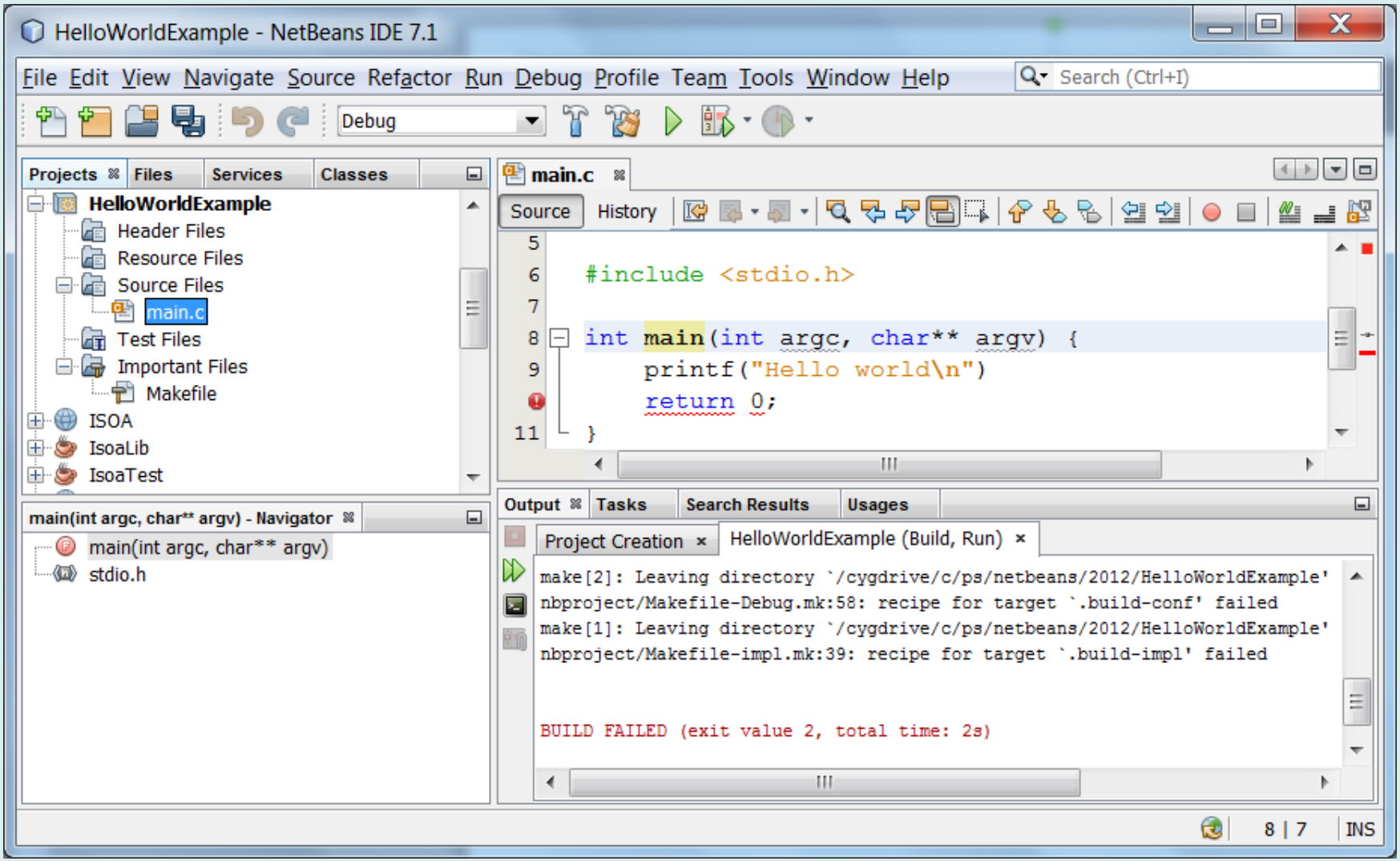
IDE

IDE (*Integrated Development Environment*) to zintegrowane środowisko budowy aplikacji:

- Zarządza zbiorem plików (projektem)
- Pozwala na wybranie bibliotek
- Automatycznie buduje **makefile** (także budując drzewo włączanych plików nagłówkowych: dyrektywy `#include "srcn.h"`)
- Automatycznie wywołuje program **make** → kompilator i konsolidator
- Dostarcza inteligentnego edytora – podświetlanie składni, automatyczne uzupełnianie nazw, refaktoryzacja kodu…
- Integruje się z debuggerem – programem do śledzenia wykonania i usuwania błędów

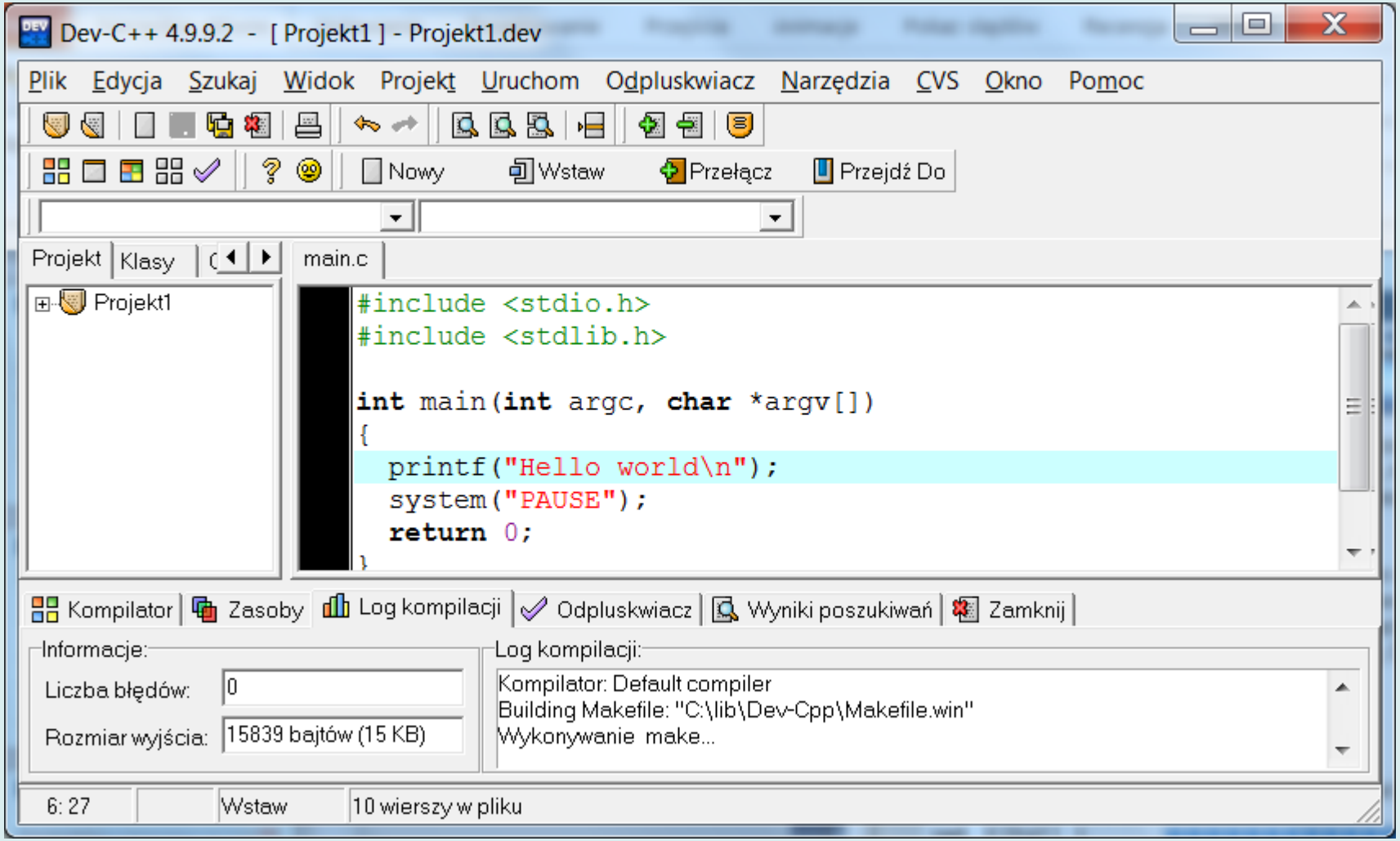
IDE – przykłady 1

- NetBeans http://netbeans.org/



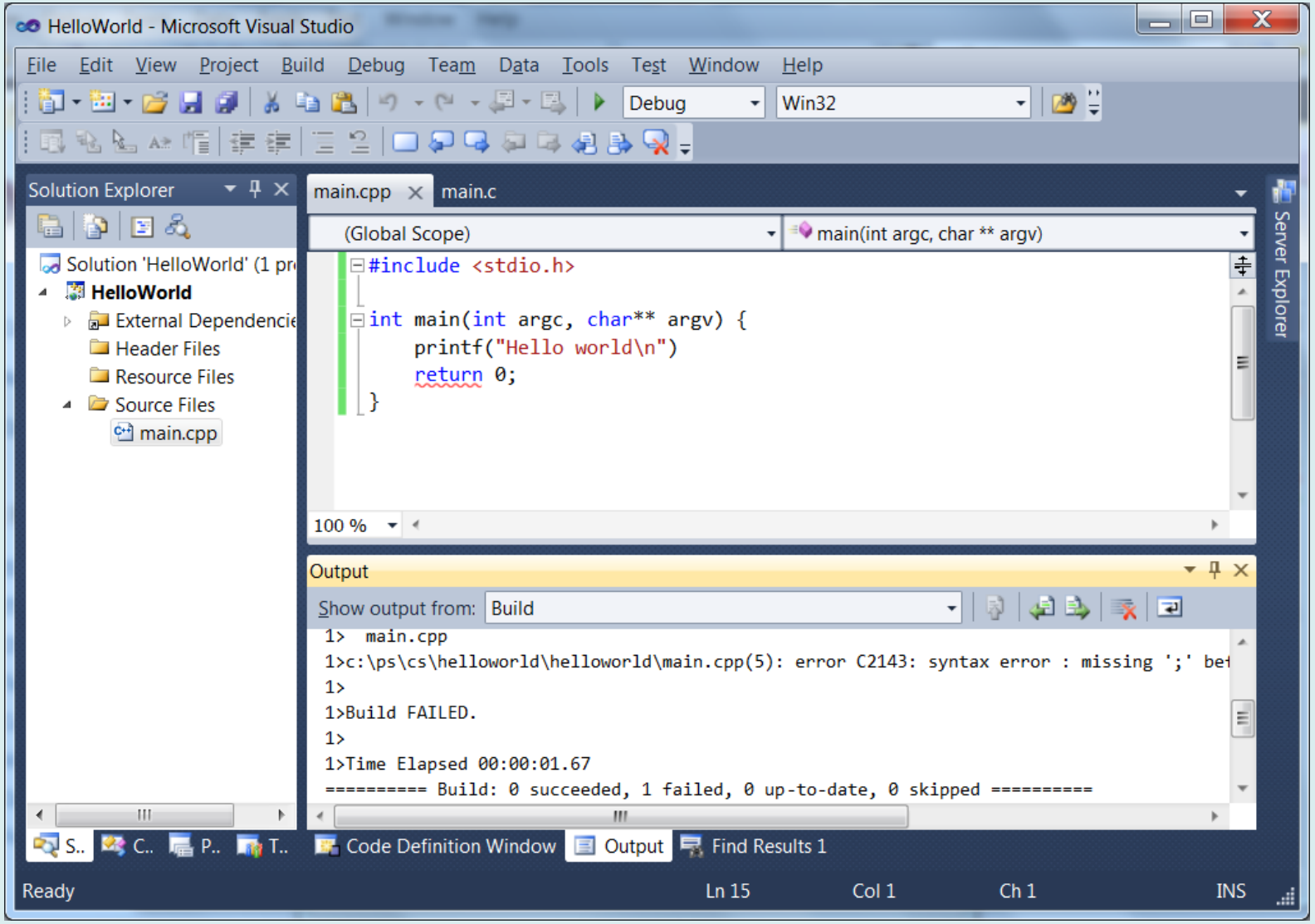
IDE – przykłady 2

- DevCpp http://www.bloodshed.net/dev/devcpp.html



IDE – przykłady 3

- Microsoft Visual Studio (dostępne dla studentów AGH)



IDE – przykłady 4

I wiele innych:

- Eclipse + CDT (C/C++ Development Tooling) http://www.eclipse.org/cdt/
- Code::Blocks http://www.codeblocks.org/
- Google: best IDE C for linux/windows/mac

Do zapamiętania

- **Preprocesor** – włącza pliki nagłówkowe, zamienia symbole na wartości, także zastępuje fragmenty kodu
- **Kompilator** – analizuje składnię, wykrywa błędy, tworzy pliki wynikowe *object*
- **Konsolidator** (linker) łączy pliki wynikowe z bibliotekami i tworzy plik wykonywalny (*.exe)
- Program **make** organizuje proces budowy programu
- **IDE** – pozwala skupić się na programowaniu, ma przyjazny edytor, automatycznie buduje makefile, uruchamia kompilator i konsolidator, wyświetla błędy, integruje się z debuggerem.