Laboratorium 10: „Biblioteki”

dr inż. Arkadiusz Chrobot

4 września 2026

1. Wprowadzenie

W instrukcji zawarto informacje o tworzeniu bibliotek statycznych w języku C oraz o preprocesorze. Rozdział pierwszy poświęcony jest tworzeniu bibliotek, w szczególności rolom jakie odgrywają w bibliotekach słowa kluczowe `extern`, `static` oraz pliki nagłówkowe. Drugi rozdział poświęcony jest konstrukcji i zastosowaniom makr preprocesora oraz kompilacji warunkowej.

2. Biblioteki

Standard języka C przewiduje tworzenie bibliotek, czyli plików z kodem źródłowym (jednostek translacji) lub skompilowanych zawierających elementy programów, które mogą być wielokrotnie wykorzystane, takie jak funkcje, stałe, zmienne i definicje typów. Biblioteki posiadają mechanizmy, które pozwalają określić ich autorom, które z zawartych w nich elementów mają być widoczne na zewnątrz i stanowić *interfejs* biblioteki, a które powinny być ukryte i tworzyć implementację. Interfejs może w kolejnych wersjach biblioteki być rozszerzany, ale nie wolno modyfikować elementów, które już do niego należą. Implementacja może za to ulegać dowolnym zmianom. Istnieją dwa typy bibliotek: statyczne i dynamiczne. W tej instrukcji zostanie opisane tworzenie bibliotek statycznych, ponieważ są one prostsze w użyciu. Biblioteki dynamiczne oferują za to więcej możliwości, ale korzystanie z nich wymaga dodatkowej pracy.

2.1. Słowo kluczowe `extern`

Słowo kluczowe `extern` służy do tworzenia deklaracji, które oznaczają, że funkcja lub zmienna została zdefiniowana w innym pliku źródłowym, innej jednostce translacji. W przypadku deklaracji zmiennej użycie tego słowa jest konieczne, inaczej kompilator potraktuje deklarację zmiennej jako jej jednoczesną definicję[1]. W przypadku definicji funkcji nie jest ono konieczne. Wystarczy sam prototyp (nagłówek) podprogramu zakończony średnikiem. Dodatkowo z listy parametrów można usunąć nazwy pozostawiając tylko i wyłącznie typy.

2.2. Słowo kluczowe `static`

Słowo kluczowe `static` użyte z definicjami zmiennych i funkcji w bibliotece informuje kompilator, że te elementy będą prywatne dla biblioteki. Innymi słowy, `static` pozwala oddzielić interfejs od implementacji. Wszystkie elementy kodu źródłowego biblioteki, których definicje nie zawierają tego słowa kluczowego mogą być udostępnione na zewnątrz.

2.3. Pliki nagłówkowe

Pliki nagłówkowe zawierają definicje typów danych oraz deklaracje funkcji i zmiennych, które zostały zdefiniowane w plikach biblioteki z rozszerzeniem `.c`. Ułatwiają one korzystanie z bibliotek, uwalniając programistów od ręcznego przepisywania wspomnianych deklaracji i definicji. Ich zawartość jest włączana do kodu źródłowego programu przed kompilacją przez preprocesor. Dyrektywą (instrukcją) preprocesora, która nakazuje wykonanie mu takiej czynności jest `#include`. W pliku nagłówkowym deklaracje i definicje umieszczane są zazwyczaj między innymi dyrektywami preprocesora, których zadaniem jest sprawdzenie, czy plik nagłówkowy nie został wcześniej już włączony do programu i jeśli tak faktycznie się stało, to zapobiegają one ponownemu wprowadzeniu jego treści do kodu źródłowego programu. Dzięki temu nie dochodzi do błędów kompilacji. Plik nagłówkowy musi być włączony do pliku, w którym znajduje się kod korzystający z elementów zawartych w bibliotece oraz powinien być włączony do plików zawierających definicje funkcji i zmiennych, których deklaracje są zawarte w pliku nagłówkowym. Jeśli plik nagłówkowy jest umieszczony w katalogu, w którym preprocesor standardowo szuka takich plików, to po dyrektywie `#include` jego nazwa jest umieszczana w nawiasach trójkątnych. W przypadku, kiedy taki plik znajduje się w innym katalogu, to po wspomnianej dyrektywie powinna być umieszczona w cudzysłowie ścieżka do niego. Jeśli plik nagłówkowy znajduje się w bieżącym katalogu, to w cudzysłowie umieszczamy tylko jego nazwę. Nie powinno się włączać do kodu programu plików, których zawartość nigdy nie będzie użyta, bo

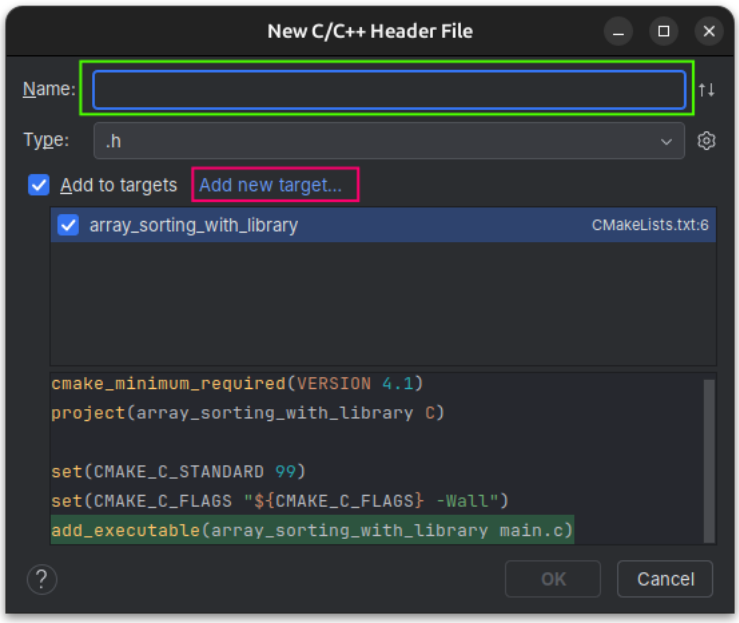
[1] W dotychczas prezentowanych programach tak właśnie było, dlatego definicję i deklarację zmiennej nazywamy w skrócie deklaracją.

preprocesor mimo to i tak przepisze ich treść, zwiększając objętość kodu źródłowego programu. W pliku nagłówkowym powinny znajdować się wyłącznie definicje typów danych i stałych oraz deklaracje funkcji i zmiennych, a więc wszystko, co nie wymaga od kompilatora decyzji o przydzieleniu pamięci.

2.4. Tworzenie bibliotek w środowisku CLion

Środowisko CLion oferuje możliwość utworzenia osobnego projektu dla biblioteki statycznej. W tej instrukcji zostanie jednak pokazane prostsze rozwiązanie, tzn. pliki biblioteki zostaną dodane do katalogu, gdzie znajduje się plik główny programu.

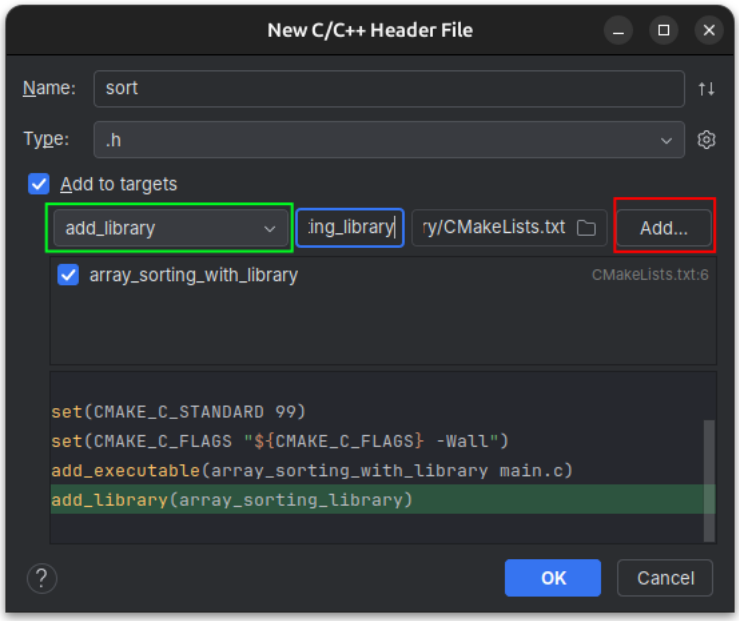
Projekt dla programu będzie nazywał się `array_sorting_with_library`. Aby dodać do niego nowy plik z rozszerzeniem `.h` w środowisku CLion należy wybrać z menu głównego następującą opcję File→New→C/C++ Header File. W oknie, które się pojawi (Rysunek 1) należy w polu „Name” (zaznaczone na rysunku zielonym prostokątem) wprowadzić nazwę nowego pliku nagłówkowego. W przy-



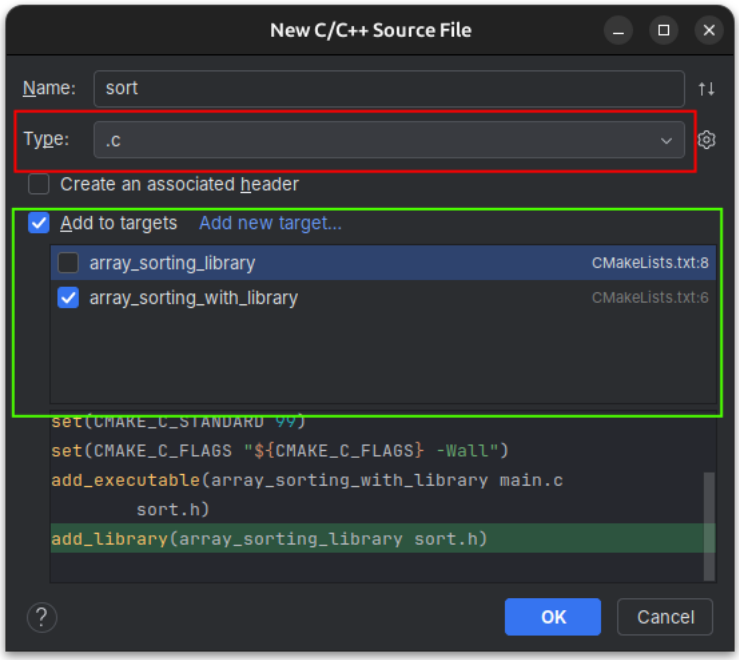
Rysunek 1: Okno dodawania nowego pliku nagłówkowego

padku opisywanego projektu będzie to `sort`. Następnie należy kliknąć link „Add new target…”, zaznaczony na rysunku czerwonym prostokątem. Widok okna zmieni się na podobny, jak ten na rysunku 2. Z rozwijanej listy, oznaczonej zielonym prostokątem na rysunku, należy wybrać opcję „add_library”, a w polu zaznaczonym na niebiesko trzeba wpisać nazwę tej biblioteki. W pisywanym projekcie to będzie `array_sorting_library`. Następnie należy kliknąć przycisk „Add…”, który na rysunku jest zaznaczony czerwonym prostokątem. Po wykonaniu tych czynności należy kliknąć przycisk „OK”.

Dodanie nowych plików z rozszerzeniem „.c” przebiega podobnie. Najpierw należy wybrać z menu głównego opcję File→New→C/C++ Source File, a następnie w oknie, które się pojawi (Rysunek 3) uzupełnić nazwę pliku w polu „Name” (tak jak na rysunku będzie to „sort”), wybrać z listy „Type” (oznaczona czerwonym prostokątem na rysunku) opcję „.c” i upewnić się, że na liście celów (oznaczona zielonym prostokątem) jest wybrany „array_sorting_library”. Potem można kliknąć przycisk „OK”. Procedurę tę należy powtórzyć dla drugiego pliku źródłowego biblioteki, który będzie się nazywał `handle_array.c`. Po wykonaniu tych czynności w pliku CMakeList.txt, należy na końcu dodać wiersz `target_link_libraries(array_sorting_with_library array_sorting_library)` Co poinformuje środowisko, że kompilując program powinno do utworzyć bibliotekę i dołączyć ją do pliku wynikowego. Aby kompilacja opisanego programu się powiodła, należy pliki źródłowe i plik nagłówkowy wypełnić kodem z listingów 4, 2, 3 i 1, zgodnie z ich nazwami.



Rysunek 2: Okno dodawania nowego pliku nagłówkowego z określeniem celu



Rysunek 3: Okno dodawania nowego pliku źródłowego

2.5. Przykład

W tym podrozdziale zaprezentowany jest przykładowy program podzielony na kilka jednostek kompilacji. Pliki `sort.h`, `sort.c` oraz `handle_array.c` tworzą bibliotekę z której korzysta plik `main.c`. Program sortuje jednowymiarową tablicę o 10 elementach typu `int`. Zarówno typ tablicy, jak i obsługujące ją funkcje są zgromadzone w plikach biblioteki.

```c
#ifndef SORT_H
#define SORT_H

typedef int array_type[10];
extern void sort(array_type);
extern void print_array(array_type);
extern void fill_array(array_type);

#endif
```

Listing 1: Plik nagłówkowy `sort.h`

W pliku `sort.h`, którego treść przedstawia listing 1 umieszczona jest definicja typu tablicy stworzona za pomocą słowa kluczowego `typedef` oraz zamieszczone są deklaracje funkcji obsługujących tę tablicę. Dyrektywa preprocesora z wiersza[2] 1 nakazuje mu sprawdzenie, czy został już w programie zdefiniowany wartownik pliku nagłówkowego, czyli w tym przypadku identyfikator `SORT_H`. Jeśli nie, to pozostała część pliku nagłówkowego, aż do wiersza nr 8 zostanie włączona do programu. Instrukcja z wiersza nr 2 definiuje wspomnianą nazwę, więc ponowne włączenie do programu tego pliku nagłówkowego nie powiedzie się. W deklaracjach funkcji na listach parametrów podano jedynie typy parametrów.

```c
#include"sort.h"

static void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void sort(array_type array)
{
    int i;
    for(i=0;i<sizeof(array_type)/sizeof(int)-1;i++) {
        int minimum,j;
        minimum=i;
        for(j=i;j<sizeof(array_type)/sizeof(int);j++)
            if(array[j]<array[minimum])
                minimum=j;
        if(minimum!=i)
            swap(&array[i],&array[minimum]);
    }
}
```

Listing 2: Plik `sort.c`

Plik `sort.c`, którego zawartość przedstawiona jest w listingu nr 2, zawiera definicje funkcji `swap()` oraz `sort()`. Pierwsza z nich jest używana tylko i wyłącznie wewnątrz pliku, dlatego jest zdefiniowana jako niewidoczna na zewnątrz, dzięki użyciu słowa kluczowego `static`. Druga jest widoczna na zewnątrz, gdyż wykonuje czynność bezpośrednio związaną z obsługą tablicy - sortuje ją. W tym pliku został włączony plik nagłówkowy `sort.h`, aby kompilator sprawdził, czy deklaracja funkcji `sort()` pokrywa się z jej definicją.

[2] **Uwaga!** Numery wierszy nie stanowią części kodu źródłowego pliku. Zostały wprowadzone w celu ułatwienia opisu programu.

```c
#include"sort.h"
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

void print_array(array_type array)
{
    int i;
    for(i=0;i<sizeof(array_type)/sizeof(int);i++)
        printf("%d ",array[i]);
    puts("");
}

void fill_array(array_type array)
{
    srandom(time(0));
    int i;
    for(i=0;i<sizeof(array_type)/sizeof(int); i++)
        array[i]=-10+random()%21;
}
```

Listing 3: Plik `handle_array.c`

Treść pliku `handle_sort.c` przedstawia listing 3. Zdefiniowano w nim dwie funkcje obsługujące tablicę. Obie są udostępnione na zewnątrz pliku. Funkcja `fill_array()` wypełnia tablice liczbami całkowitymi z zakresu od `-10` do `10`. Funkcja `print_array()` wypisuje zawartość tablicy na ekran. Oprócz plików nagłówkowych zawierających deklaracje funkcji używanych przez funkcje zdefiniowane w tym pliku włączono także plik `sort.h`, ponieważ zawiera on definicję typu tablicy obsługiwanej przez te funkcje. Dzięki temu typowi możliwe jest również wyznaczenie liczby elementów jaką posiada tablica. Wystarczy podzielić rozmiar typu tablicy, przez rozmiar typu pojedynczego elementu tablicy. Drugim powodem włączenia pliku `sort.h` do pliku `handle_array.c` jest umożliwienie kompilatorowi sprawdzenia, czy deklaracje w tym pliku pokrywają się z definicjami funkcji.

```c
#include"sort.h"

int main(void)
{
    array_type small_array;
    fill_array(small_array);
    print_array(small_array);
    sort(small_array);
    print_array(small_array);
    return 0;
}
```

Listing 4: Plik `main.c`

Plik `main.c` z listingu 4 zawiera główną funkcję programu. Do tego pliku włączono jedynie plik nagłówkowy `sort.h`. Inne nie są potrzebne. Funkcja `main()` deklaruje i definiuje tablicę, która jest zmienną lokalną i wywołuje funkcje ją obsługujące. Typ tej tablicy jest zdefiniowany we włączonym pliku nagłówkowym, gdzie również umieszczone są deklaracje użytych funkcji.

3. Preprocesor

Zadaniem preprocesora jest przygotowanie kodu źródłowego do kompilacji. Posiada on własny język programowania, który składa się z instrukcji nazywanych *dyrektywami*. Do tej pory wykorzystywaliśmy jedną z nich - `#define` - do definiowania stałych. Może ona być użyta także do innych celów. Za jej pomocą można definiować podprogramy, które nazywane są makrami lub makrodefinicjami. Dyrektywy warunkowe preprocesora są także używane do sterowania procesem kompilacji kodu źródłowego.

3.1. Makra

Makro lub makrodefinicja jest podprogramem, który nie jest kompilowany przez kompilator, a obsługiwany przez preprocesor. W przeciwieństwie do funkcji nie jest on wywoływany, a rozwijany w miejscu użycia. Oznacza to, że preprocesor umieszcza wszystkie instrukcje zawarte w makrze, a nie rozkaz wywołania makra, w miejscu, gdzie ono zostało użyte. Podobnie jak funkcja makro może dysponować listą parametrów. W czasie rozwijania te parametry zastępowane są argumentami, a nie wartościami tych argumentów. Oznacza to, że jeśli za parametr np. `x` makra zostanie podstawiony argument w postaci wyrażenia `a+b*c`, to podczas rozwijania makrodefinicji każde wystąpienie `x` zostanie zastąpione wyrażeniem `a+b*c`. Dodatkowo parametry makra nie mają typów, bo preprocesor nie wykonuje ich kontroli w trakcie rozwijania makrodefinicji. Wszystkie błędy, które mogą być spowodowane brakiem kontroli typów przez preprocesor są wykrywane dopiero na etapie kompilacji. Tworząc makra należy pamiętać o następujących regułach:

1. makro preprocesora definiuje się za pomocą dyrektywy `#define` preprocesora, ale istnieje możliwość „wyłączenia" tej definicji w określonych częściach kodu za pomocą dyrektywy `#undef`,
2. nazwy parametrów w miejscu ich użycia wewnątrz makra powinny być umieszczone w nawiasach okrągłych,
3. jeśli definicja makra obejmuje więcej niż jeden wiersz, to muszą się one kończyć kończyć się znakiem `\` (lewy ukośnik), który informuje preprocesor, że kolejny wiersz też należy do definicji makra; wyjątkiem od tej reguły jest ostatni wiersz makrodefinicji,
4. na końcu definicji makra nie należy stawiać średnika,
5. makra obejmujące wiele wierszy kodu powinny być umieszczane nie w zwykłych nawiasach klamrowych, ale w bloku instrukcji pętli `do…while(0)`, aby można było postawić średnik za miejscem użycia makra w instrukcji warunkowej.

Panująca wśród programistów języka C konwencja nakazuje pisać nazwy makrodefinicji i ich parametrów wielkimi literami. Definicje makr należy umieszczać w plikach nagłówkowych, jednakże mogą one też być zawarte w plikach z rozszerzeniem `.c`. W definicji makra można użyć także specjalnych operatorów. Operator `#` mieszczony przed nazwą parametru zamienia argument podstawiony za ten parametr w ciąg znaków. Z kolei operator `##` służy do konkatenacji (sklejania) łańcuchów, które są ustawione po obu jego stronach. Te dwa operatory są także wyjątkami od reguły nr 2. Jeśli ich argumentami są parametry makrodefinicji, to ich nazwy nie mogą być umieszczone w nawiasach okrągłych. Listing 5 zawiera przykład definicji i użycia makra.

```c
#include<stdio.h>

#define PRINT_SUM(START,END) do { \
    unsigned int i, sum=0; \
    for(i=(START); i<=(END); i++) \
        sum+=i; \
    printf("Suma podanych liczb to %u.\n", sum); \
} while(0)

int main(void)
{
    unsigned int start, end;
    puts("Podaj dwie liczby naturalne.");
    scanf("%u",&start);
    scanf("%u",&end);
    if(start<=end)
        PRINT_SUM(start,end);
    else
        puts("Pierwsza liczba powinna być mniejsza lub równa drugiej.");
    return 0;
}
```

Listing 5: Przykład zdefiniowania i użycia makra

Makro `PRINT_SUM` liczy i wypisuje na ekran sumę ciągu arytmetycznego, którego pierwszy i ostatni wyraz podaje użytkownik, a różnica między wyrazami wynosi jeden. Przykłady innych makr znajdują się w materiałach do wykładu. Pewną alternatywą dla makr są funkcje `inline`, które również zostały opisane we wspominanych materiałach.

3.2. Asercje

Asercje nazywane też niezmiennikami to wyrażenia, które służą do kontroli poprawności działania programu. Ich wartość musi być prawdziwa, jeśli program wykonuje się prawidłowo. Niezmienniki umieszcza się w miejscach programu, które wykonują operacje powodujące zmianę stanu zmiennych i bada się w ten sposób, czy ta zmiana była prawidłowa. Asercje mogą być umieszczone przed rozpoczęciem pętli, po jej zakończeniu oraz w jej wnętrzu. Mogą być także umieszczone przed jak i po wywołaniu funkcji, wewnątrz definicji tej funkcji, jak i w wielu innych miejscach, jeśli badanie poprawności tego wymaga. Twórcy standardu języka C zaproponowali specjalne makro o nazwie `assert`[3], które służy do umieszczania w kodzie źródłowym programu niezmienników. Jego argumentem jest wyrażenie, które powinno być prawdziwe po wykonaniu danej części kodu. Jeśli tak nie jest, to makro przerywa wykonanie programu i wypisuje na ekran komunikat o miejscu wystąpienia błędu. Listing 6 zawiera przykład użycia takiego makra.

[3]Nazwa tego makra nie jest zgodna z konwencją, bo jest pisana małymi literami.

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

double get_circle_circumference(double radius)
{
    assert(radius>=0);
    return 2*M_PI*radius;
}

int main()
{
    puts("Podaj promień okręgu.");
    double circle_radius;
    scanf("%lf",&circle_radius);
    double circumference = get_circle_circumference(circle_radius);
    assert(circumference>=0);
    printf("Obwód okręgu to: %lf.\n",circumference);
    return 0;
}
```

Listing 6: Przykład użycia makra assert

W programie użyto makra `assert` dwukrotnie. Pierwszy raz w wierszu nr 7, aby zbadać, czy podana przez użytkownika długość promienia jest prawidłowa, a drugi raz w wierszu nr 18, aby określić czy wyliczona przez funkcję długość obwodu okręgu jest poprawna. Jeśli wykonanie programu zostanie przerwane w siódmym wierszu, to oznacza to, że użytkownik podał złą wartość, a jeśli w 18, to znaczy to, że funkcja ma defekt. Aby wyłączyć działanie makra należy zdefiniować znacznik (nazwę) `NDEBUG`. Można to zrobić na dwa sposoby — albo w kodzie źródłowym programu za pomocą dyrektywy `#define`, tuż przed włączeniem pliku nagłówkowego `assert.h`, albo w opcjach wywołania kompilatora. Ten drugi sposób w przypadku środowiska CLion polega na dodaniu do pliku „CMakeList.txt" instrukcji `add_compile_definitions(NDEBUG)`, np. na jego końcu. Jeśli chcemy przywrócić działanie makra `assert`, to należy usunąć opisaną instrukcję z pliku. Język C ma również inne predefiniowane makra, które mogą być użyteczne:

`__DATE__` zamieniane jest na datę w momencie kompilacji,

`__TIME__` zamieniane jest na czas w momencie kompilacji,

`__FILE__` zamieniane jest na nazwę kompilowanego pliku,

`__LINE__` zamieniane jest na numer wiersza w kompilowanym pliku.

3.3. Kompilacja warunkowa

Preprocesor dysponuje instrukcjami warunkowymi, które pozwalają na sterowanie procesem kompilacji programu. Wybrane spośród tych dyrektyw są zamieszczone w tabeli 1 wraz z opisem do czego służą.

`#ifdef`	Jeśli znacznik występujący za nią został wcześniej zdefiniowany, to preprocesor uwzględnia występujące za nią instrukcje, a jeśli nie, to je pomija.
`#else`	Działa podobnie do `else` w „zwykłej” instrukcji warunkowej.
`#ifndef`	Jeśli występujący za nią znacznik nie został wcześniej zdefiniowany, to preprocesor uwzględnia występujące za nią instrukcje, w przeciwnym razie pomija je.
`#elif`	Stanowi połączenie dyrektyw `#else` i `#if`.
`#endif`	Kończy blok dyrektywy `#ifdef` lub `#ifndef`.

Tabela 1: Dyrektywy preprocesora sterujące kompilacją

Listing 7 zawiera przykład użycia takich instrukcji.

```c
#include<stdio.h>

#ifdef VERBOSE
#define PRINT_VERBOSE(VARIABLE)\
    printf("Zmienna %s ma następującą wartość: %lu.\n",#VARIABLE,(VARIABLE))
#else
#define PRINT_VERBOSE(VARIABLE)
#endif

int main(void)
{
    unsigned long int sum=0, i;
    for(i=0;i<100;i++) {
        sum+=i;
        PRINT_VERBOSE(sum);
    }
    printf("Wynik: %lu\n",sum);
    return 0;
}
```

Listing 7: Przykład użycia kompilacji warunkowej

W zamieszczonym programie użyto kompilacji warunkowej do dodania kodu, który wpisuje wartość zmiennej `sum` w każdej iteracji pętli `for`. Kod ten jest umieszczony w makrze `PRINT_VERBOSE` i jest włączany do programu wtedy i tylko wtedy gdy zdefiniowany jest znacznik `VERBOSE`. Można go zdefiniować tymi samymi sposobami, co znacznik `NDEBUG`. Jeśli to makro byłoby wielokrotnie użyte w programie, to można je wyłączać i ponownie włączać w określonych miejscach kodu źródłowego za pomocą dyrektyw `#define` i `#undef` użytych razem ze znacznikiem `VERBOSE`.

4. Zadania

Uwaga! Wszystkie programu muszą być napisane z podziałem na funkcje z parametrami.

1. Wykonaj krokowo program opisany w podrozdziale 2.5. Zaobserwuj co się dzieje, jeśli użyjesz klawisza `F7`, do wykonania instrukcji wywołujących funkcje z biblioteki.
2. Zmień przykładowy program z rozdziału o bibliotekach tak, aby funkcje sortujące i losujące miały dodatkowe parametry, które będą określały kierunek sortowania tablicy i zakres z jakiego będą losowane liczby dla tablicy.
3. Zmień przykładowy program z rozdziału o bibliotekach zastępując funkcję sortującą z użyciem algorytmu sortowania przez wybór na funkcję sortującą z użyciem sortowania przez wstawianie. Wolno Ci zmienić jedynie zawartość pliku `sort.c`.

4. Napisz program, który posortuje tablicę o 10 elementach typu `int` wylosowanych z zakresu od `-10` do `10`, a następnie za pomocą makra `assert` sprawdzi, czy ta tablica jest posortowana. Zastanów się jak użyć tego makra, aby po zdefiniowaniu znacznika `NDEBUG` zostało w programie jak najmniej zbędnego kodu.

5. Napisz makro, które będzie przyjmowało dwa argumenty - wyrażenie dowolnego typu i ciąg formatujący dla wartości tego wyrażenia i będzie wypisywało komunikat według następującego wzorca: „Wyrażenie (wyrażenie) ma wartość (wartość) w linii nr (numer linii) pliku (ścieżka/nazwa pliku).". Frazy w nawiasach powinny być zastąpione przez makro odpowiednimi wartościami. Użyj tego makra w programie. **Wskazówka:** Jeśli w makrodefinicji umieścimy obok siebie kilka łańcuchów znaków, to zostaną one połączone.

6. Napisz program, który będzie włączał lub wyłączał zdefiniowane w poprzednim zadaniu makro w zależności od tego, czy został zdefiniowany znacznik `CHECK`.

7. Napisz program, w którym kilkukrotnie użyjesz makra zdefiniowanego w czwartym zadaniu. Pokaż jak za pomocą dyrektyw `#define` i `#undef` można włączać i wyłączać to makro w określonych miejscach w kodzie.