Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 2. Struktura aplikacji w paradygmacie programowania proceduralnego

Zagadnienia do opracowania:

- języki niskiego i wysokiego poziomu
- funkcja *main()*
- paradygmat programowania proceduralnego
- struktura aplikacji w językach C/C++
- zadania preprocesora, kompilatora i konsolidatora w procesie budowania aplikacji
- kompilacja i asemblacja kodu źródłowego
- rodzaje zmiennych i ich rozmiar w pamięci komputera
- operator *sizeof*

Spis treści

1. Cel ćwiczenia

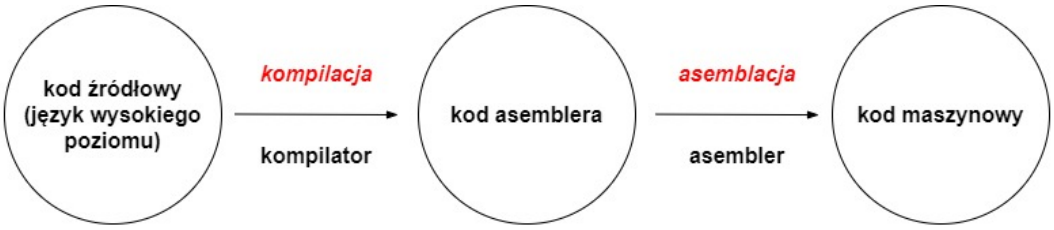
Celem ćwiczenia jest zapoznanie z procedurą tworzenia prostych aplikacji w językach C/C++. Laboratorium obejmuje zagadnienia struktury aplikacji w paradygmacie programowania proceduralnego oraz procesu budowania pliku wykonywalnego z kodów źródłowych.

2. Wprowadzenie

2.1. Języki niskiego i wysokiego poziomu

Zarówno język *C*, jak i język *C++*, to ***języki wysokiego poziomu***. Oznacza to, że ich **składania** oraz **słowa kluczowe** nie są w swojej postaci zrozumiałe dla procesora komputera. Idea ***języków wysokiego poziomu*** stoi w opozycji do ***języków niskiego poziomu*** takich, jak ***języki asemblera***. ***Języki niskiego poziomu*** wykorzystują zbiór instrukcji procesora, odpowiadający określonym poleceniom w ***kodzie maszynowym***, np. polecenie skoku bezwarunkowego do adresu 0xA4C31F08 w kodzie asemblera może wyglądać następująco: ***JMP A4C31F08***, a odpowiadający mu zapis w szesnastkowym kodzie maszynowym tak: ***EB 08 1F C3 A4***. Z tego względu programy napisane w ***językach niskiego poziomu*** nie są z reguły przenośne między różnymi architekturami procesorów (*ARM, RISC-V, CISC, ...*). ***Języki wysokiego poziomu*** stawiają zarówno na przenośność kodu, jak i wygodę programisty w tworzeniu aplikacji. Kod ma być jak najbardziej zrozumiały i wygodny w użyciu dla użytkownika (programisty), kosztem dodatkowych procesów tłumaczenia go (***kompilacji***) na instrukcje, które jest w stanie wykonać procesor. ***Kompilator*** to program komputerowy służący do tłumaczenia kodu napisanego w jednym języku (*źródłowym*) na równoważny kod w innym języku (*wynikowym*). Proces tłumaczenia nazywa się ***kompilacją*** i w naszym przypadku będzie służył do tłumaczenia **kodu źródłowego** na **kod asemblera** (rys. 2.1). **Kod asemblera** należy

następnie zmienić na **kod maszynowy**. Taka zamiana nazywana jest ***asemblacją***, a program który ją wykonuje – ***asemblerem***.



Rys. 2.1. Proces kompilacji i asemblacji kodu źródłowego

2.2. Funkcja *main()*

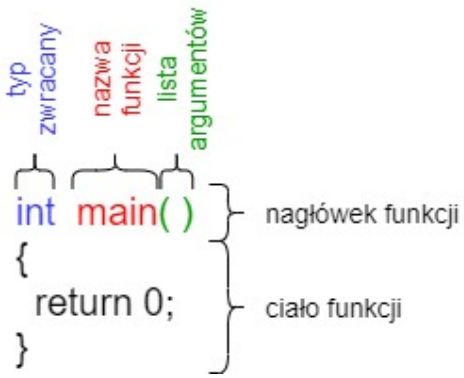
Każdy program napisany zarówno w języku *C*, jak i *C++*, zawiera funkcję ***main()***, w której wywołane są (bezpośrednio lub pośrednio) wszystkie instrukcje przeznaczone do wykonania w ramach aplikacji, w określonej kolejności. Przyjęło się, żeby umieszczać funkcję ***main()*** w pliku **main.c** (lub **main.cpp** w przypadku języka *C++*). Najprostszy program napisany w językach *C/C++* przedstawiono na listingu 1. Jedyną instrukcją jaką wykonuje w tym przypadku funkcja ***main()*** jest zwrócenie liczby **0**.

```
1 int main() {
2     return 0;
3 }
```

Listing 1. Funkcja ***main()***

Funkcja (inaczej *podprogram*) to wydzielony kod programu komputerowego, realizujący konkretne zadanie, na podstawie przekazanych argumentów. Jeżeli zachodzi taka potrzeba, to funkcja w rezultacie swojego działania może zwrócić określoną wartość (funkcje niezwracające żadnej wartości nazywa się również *proceduramі*). Kod znajdujący się przed nawiasami klamrowymi,

tj. ***int main()*** nazywa się ***nagłówkiem funkcji***. Zawiera on informację o liście argumentów przyjmowanych przez funkcję (tu: pusta lista argumentów) oraz o typie zwracanym (tu: *int* – liczba całkowita). Wewnątrz nawiasów klamrowych znajduje się ***ciało funkcji***, w którym umieszczone są wszystkie instrukcje przeznaczone do wykonania w czasie działania programu (tu: jedna instrukcja – zwróć 0) (rys. 2.2). Nazwa funkcji wraz z jej listą argumentów nazywana jest również ***sygnaturą funkcji***. Warto podkreślić, że typ zwracany **nie należy do sygnatury funkcji**. Jest to istotne przy zagadnieniu ***przeciążania funkcji***.



Rys. 2.2. Budowa funkcji ***main()***

Warto zastanowić się nad następującym problemem: jeśli wszystkie instrukcje aplikacji wykonują się wewnątrz ciała funkcji ***main()***, to co dzieje się z wartością przez nią zwracaną? Wartość ta to nic innego, jak kod błędu wykonania aplikacji. Przyjęło się, że kod 0 oznacza brak błędu, natomiast każda inna niezerowa wartość koduje określony błąd. Na przykład wartość 14 zwrócona z funkcji ***main()*** mogłaby sygnalizować błąd odczytu danych z pliku. Zwrócony kod błędu przechwytywany jest i obsługiwany przez proces, który uruchomił aplikację; dla uproszczenia można przyjąć, że jest to system operacyjny.

2.3. Proces budowania aplikacji w językach C/C++

Zarówno język *C*, jak i *C++*, wspierają ***paradygmat programowania proceduralnego***. Zakłada on dzielenie kodu na **fragmenty realizujące pojedynczą odpowiedzialność**. Realizowane jest to przez wydzielanie odpowiednich funkcji przetwarzających przekazane argumenty i zwracających określone wartości. Funkcje wykonywane są w odpowiedniej kolejności, określonej wewnątrz ciała funkcji ***main()***. Wynik zwrócony z danego podprogramu może być wykorzystany przez kolejne wykonywane funkcje.

Aplikacja tworzona w językach *C/C++* składa się z plików nagłówkowych (rozszerzenie **.h/.hpp**) oraz plików źródłowych (rozszerzenie **.c/.cpp**, w tym plik ***main.c/main.cpp***). Pliki nagłówkowe zawierają (głównie) ***deklaracje*** (zapowiedzi) zmiennych i funkcji wykorzystywanych w aplikacji, natomiast pliki źródłowe ich ***definicje***. Każdy plik źródłowy stanowi odrębną ***jednostkę translacji (kompilacji)*** i jest poddawany procesowi kompilacji oddzielnie. ***Kompilator*** przeprowadza statyczną analizę kodu, optymalizuje go, a następnie tłumaczy kod źródłowy do postaci **kodu asemblera** (rozszerzenie **.s**). Kod asemblera przetwarzany jest do postaci kodu maszynowego, zawartego w tzw. ***plikach obiektowych*** (rozszerzenie **.o**). Pliki obiektowe, stanowiące wynik procesu kompilacji i asemblacji poszczególnych jednostek translacji, są pozbawione referencji do wywoływanych symboli. Oznacza to, że są od siebie odseparowane i nie zawierają informacji o innych plikach obiektowych. W przypadku programowania mikrokontrolerów często wykorzystuje się pliki ***Intel HEX*** (rozszerzenie **.hex**). Są to pliki zawierające skompilowany kod maszynowy w postaci heksadecymalnej. Plik Intel HEX można utworzyć z pliku obiektowego, np. wykorzystując program *GNU objcopy*.

Wynik procesów kompilacji i asemblacji przetwarzany jest przez ***konsolidator*** (*ang. linker*). Jego zadaniem jest połączenie wszystkich plików obiektowych oraz ***bibliotek statycznych*** w jeden plik wykonywalny (możliwy do uruchomienia z poziomu systemu operacyjnego). Konsolidator zapewnia referencje między symbolami znajdującymi się w różnych jednostkach translacji.

Rozdzielenie kompilatora i konsolidatora nazywa się kompilacją rozłączną. Jednym z powodów takiego zabiegu jest możliwość ponownej kompilacji jedynie zmodyfikowanego pliku, a nie wszystkich plików projektu.

Na listingu 1. przedstawiona jest zawartość pliku ***main.c***, który posłuży do zaprezentowania, krok po kroku, procesu kompilacji prostej aplikacji w języku *C* z poziomu konsoli systemowej. Wykorzystane zostaną do tego darmowe kompilatory **gcc** (język *C*) oraz **g++** (język *C++*), rozwijane w ramach projektu *GNU*. Wywołując w konsoli systemowej polecenie ***gcc main.c***, z poziomu katalogu zawierającego plik ***main.c***, przeprowadzony zostanie proces kompilacji i konsolidacji kodu źródłowego. Wynikiem tej operacji w systemie Windows jest plik wykonywalny **a.exe** (w systemie Linux **a.out**). Aby zmienić nazwę pliku wynikowego można posłużyć się flagą **-o**, np. **gcc main.c -o test** poskutkuje utworzeniem pliku **test.exe**. Aby zatrzymać proces kompilacji na etapie translacji kodu do języka asemblera, można posłużyć się flagą **-S**. Wówczas wynikiem wykonania polecenia **gcc main.c -S** jest plik ***main.s***. Możliwa jest również zmiana stopnia optymalizacji kodu przez kompilator, wykorzystując flagę **-O** od **-O0** (domyślna, najniższa wartość) do **-O3** (najwyższa wartość – agresywna optymalizacja). W celu wygenerowania plików obiektowych można posłużyć się poleceniem **gcc -c main.c**. Listę wszystkich dostępnych flag kompilatora można wyświetlić korzystając z polecenia **gcc --help**.

Uruchomienie skompilowanego pliku wykonywalnego w konsoli nie wywoła żadnych zmian. Nic dziwnego, program z listingu 1. nie robi nic poza zwróceniem do systemu operacyjnego kodu błędu. Można natomiast sprawdzić jaki kod błędu odebrał system operacyjny. W systemie Windows kod błędu ostatnio wykonanej instrukcji przechowywany jest w zmiennej **ERRORLEVEL**. Aby wypisać jej wartość z poziomu konsoli systemowej można posłużyć się poleceniem **echo %ERRORLEVEL%**. W systemie Linux analogiczne zachowanie otrzymamy wykonując polecenie **echo $?**

Na listingu 2. przedstawiono aplikację (plik ***main.c***), od implementacji której zaczyna każdy początkujący programista – aplikacji wyświetlającej na

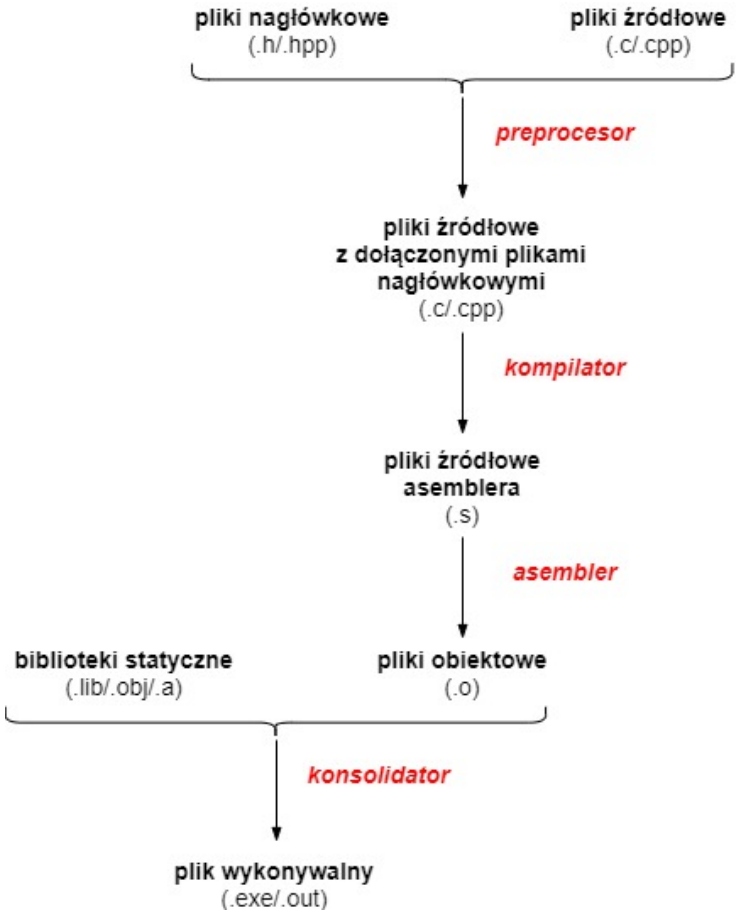
ekranie napis "Hello, world!". W języku *C* zadanie przesyłania łańcucha znaków do ***standardowego wyjścia*** realizuje funkcja ***printf()***, znajdująca się w pliku nagłówkowym ***stdio.h*** *[więcej informacji na temat standardowego wejścia i wyjścia w Ćw. 4].*

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello, world!");
5     return 0;
6 }
```

Listing 2. Program *Hello, world!*

Na listingu 2. widać, że w języku *C* (analogicznie jest w języku *C++*), każde polecenie musi kończyć się średnikiem (patrz: ***printf()*** i ***return***). A co w przypadku instrukcji ***#include***, która pojawiła się **nad definicją funkcji *main()***? Jest to tzw. ***dyrektywa preprocesora***. ***Preprocesor*** to program komputerowy, którego zadaniem jest odpowiednie przetworzenie instrukcji zawartych w kodzie, zwanych właśnie ***dyrektywami preprocesora***. ***Preprocesor*** przetwarza kod źródłowy, zanim ***kompilator*** rozpocznie jego analizę. Można powiedzieć, że ***preprocesor*** przygotowuje kod źródłowy dla ***kompilatora***. Każda dyrektywa rozpoczyna się symbolem #. Preprocesor przeszukuje kod źródłowy w poszukiwaniu dyrektyw i zastępuje je odpowiednim kodem, zrozumiałym dla kompilatora. Dyrektywa **#include** załącza (wkleja) zawartość podanego pliku (tu: ***stdio.h***) w miejscu wywołania. Wyróżnia się dwa sposoby zadawania plików do załączenia przez preprocesor. **#include ⟨plik⟩** określa, że preprocesor powinien poszukiwać zadanego pliku pośród plików ***biblioteki standardowej***. **#include "plik"** określa, że preprocesor powinien w pierwszej kolejności poszukiwać zadanego pliku wśród plików projektu (w tym samym katalogu, jeśli nie określono inaczej), a następnie pośród plików ***biblioteki standardowej***. Ten drugi zapis sto-

sowany jest do załączania plików utworzonych przez programistę. **Istotną regułą jest, aby za pomocą dyrektywy #include załączać tylko pliki nagłówkowe (rozszerzenie .h). Pliki źródłowe (rozszerzenie .c) są kompilowane rozdzielnie i wiązane przez konsolidator**. Podsumowując, proces budowania aplikacji w językach *C/C++* może być schematycznie przedstawiony tak, jak na rys. 2.3.



Rys. 2.3. Proces budowy aplikacji w językach *C/C++*

2.4. Podstawowe typy zmiennych

Kolejnym krokiem w rozszerzeniu aplikacji o dodatkowe funkcje jest utworzenie nowego pliku nagłówkowego ***defs.h*** w tym samym katalogu, w którym znajduje się plik ***main.c***. Na potrzeby tego ćwiczenia znajdą się w nim deklaracje zmiennych i funkcji, do których będzie odwoływać się funkcja ***main()***. ***Deklaracją*** w językach *C/C++* nazywamy pewną *zapowiedź* zmiennej bądź funkcji. Jest to informacja dla kompilatora, że jeżeli napotka w kodzie daną nazwę, to jest to zmienna zdeklarowanego typu lub funkcja o określonym nagłówku. Zatem ***deklaracja*** określa nazwę i typ zmiennej lub funkcji. Z kolei ***definicja*** zapewnia wszystkie informacje potrzebne do utworzenia konkretnego egzemplarza zmiennej lub funkcji (określa wartość zmiennej lub ciało funkcji). Warto zwrócić uwagę, że możliwe jest definiowanie zmiennych/funkcji bez ich wcześniejszego deklarowania (jak w przypadku funkcji ***main()***). Zawartość pliku ***defs.h*** przedstawiono na listingu 3. Zawarto w nim dwie deklaracje zmiennych (*integerValue* i *floatingPointValue*) oraz jedną deklarację funkcji (*multiply()*). Wykorzystano tutaj różne typy zmiennych (***int***, ***float***). Są to przykłady podstawowych typów danych dostępnych w językach *C/C++*. Więcej typów, wraz z ich rozmiarem w pamięci komputera (w bajtach) oraz zakresem przyjmowanych wartości zestawiono w tabeli 1. Słowo kluczowe ***void*** nie określa stricte typu, ponieważ nie można utworzyć zmiennej typu ***void***. Służy ono do deklarowania funkcji, które nic nie zwracają (*procedur*). Na podstawie tabeli 1. widać, że zmienna *integerValue* to zmienna typu całkowitego ze znakiem o rozmiarze min. 2 B, natomiast zmienna *floatingPointValue* to zmienna typu zmiennoprzecinkowego pojedynczej precyzji (o rozmiarze typowo 4 B). Funkcja *multiply()* przyjmuje dwa argumenty (typu ***int*** oraz ***float***) oraz zwraca wynik typu ***float***. Warto zwrócić uwagę, że przy typach przyjmowanych przez funkcję zmiennych pojawiły się ich nazwy. Są to nazwy pod jakimi można odwołać się do danej zmiennej w obrębie ciała funkcji i tylko tam (podczas definiowania funkcji *multiply()* nagłówek funkcji musi być zgodny co do deklarowanych typów zmiennych, ale nazwy argumentów nie muszą się pokrywać). Dyrektywa **#pragma once** zapew-

nia, że dany plik nagłówkowy zostanie załączony w obrębie zadanej jednostki translacji tylko raz, nawet, jeżeli w innym załączanym pliku nagłówkowym zastosujemy dyrektywę **#include "defs.h"**. Zabezpiecza to program przed wystąpieniem konfliktu nazw (wielokrotna deklaracja), skutkującego błędem kompilacji. Linie rozpoczynające się podwójnym ukośnikiem (//) to komentarze (nie podlegają procesowi kompilacji).

```
1 // Zalacz plik naglowkowy defs.h maksymalnie raz w
     obrebie jednostki translacji
2 #pragma once
3
4 // Deklaracje zmiennych
5 int integerValue;
6 float floatingPointValue;
7
8 // Deklaracja funkcji
9 float multiply(int multiplier, float multiplicand);
```

Listing 3. Zawartość pliku ***defs.h***

Kolejnym krokiem jest dostarczenie definicji zmiennych i funkcji zdeklarowanych w pliku ***defs.h***. Na potrzeby tego ćwiczenia zostaną one umieszczone w pliku ***defs.c*** (listing 4). Na wstępie załączany jest plik nagłówkowy, zawierający deklaracje zmiennych i funkcji. Następnie zostają przypisane wartości do zmiennych oraz zdefiniowane operacje wykonywane wewnątrz funkcji *multiply()* (mnożenie dwóch zmiennych).

Tabela 1. Podstawowe typy zmiennych w językach *C/C++*

Typ	**Rozmiar [B]**	**Zakres wartości**
char	1	[−128; +127]
unsigned char	1	[0; 255]
short	min. 2	min. [−32768; +32767]
unsigned short	min. 2	min. [0; 65535]
int	min. 2	min. [−32768; +32,767]
unsigned int	min. 2	min. [0; 65535]
long	min. 4	min. [−2147483648 +2147483647]
unsigned long	min. 4	min. [0; 4294967295]
long long	min. 8	min. [−9223372036854775808, +9223372036854775807]
unsigned long long	min. 8	min. [0; 18446744073709551615]
float	typ. 4	typ. $3.4E \pm 38$
double	typ. 8	typ. $1.7E \pm 308$
long double	typ. 12	typ. $1.1E \pm 4932$
bool	1[1]	true/false
void	0	brak wartości

[1]Może się różnić w zależności od implementacji kompilatora

```c
// Zalacz (wklej) plik naglowkowy defs.h
#include "defs.h"

// Definicje zmiennych
int integerValue = 2;
float floatingPointValue = 5.5;

// Definicja funkcji
float multiply(int multiplier, float multiplicand) {
    return multiplier * multiplicand;
}
```

Listing 4. Zawartość pliku ***defs.c***

Zdefiniowane zmienne i funkcja *multiply()* mogą zostać wykorzystane wewnątrz funkcji ***main()***, czego przykład przedstawiono na listingu 5. Warto zwrócić uwagę na różny sposób dołączania plików nagłówkowych ***stdio.h*** oraz ***defs.h***. Wewnątrz funkcji ***main()*** wywoływana jest funkcja *multiply()* z argumentami *integerValue* oraz *floatingPointValue*. Wynik mnożenia przypisywany jest do zmiennej *result* i wyświetlany w konsoli (funkcja *printf()*).

```c
#include <stdio.h>
#include "defs.h"

int main() {
    float result = multiply(integerValue,
   floatingPointValue);
    printf("Multiplication result: %f", result);
    return 0;
}
```

Listing 5. Wywołanie funkcji *multiply()*

W celu zbudowania aplikacji należy wykonać polecenie **gcc main.c defs.c -o multApp**. Po uruchomieniu pliku wykonywalnego **multApp.exe** (lub **multApp.out**) na konsoli wyświetlony zostaje wynik mnożenia 2 przez 5,5. Możliwe jest zatrzymanie proces kompilacji na etapie proprocesingu, aby zobaczyć wynik pracy ***preprocesora***. W tym celu należy skorzystać z flagi **-E** (**gcc -E main.c defs.c**).

2.5. Typy zmiennych o stałym rozmiarze

Rozmiar przeważającej większości podstawowych typów zmiennych w językach *C/C++* jest zależny od implementacji kompilatora i architektury systemu operacyjnego (patrz: tabela 1). Jednakże, w przypadku typów całkowitych, zdefiniowane są również typy zmiennych o stałym rozmiarze (począwszy od standardów *C99* i *C++11*). Takie rozwiązanie zapewnia przenośność między różnymi platformami i niezależność od zastosowanego kompilatora. Typy zmiennych o stałym rozmiarze zebrano w tabeli 2. Aby skorzystać z całkowitoliczbowych typów zmiennych o stałym rozmiarze należy załączyć plik nagłówkowy **stdint.h** w przypadku języka *C* lub **cstdint** w przypadku języka *C++*.

Tabela 2. Typy zmiennych o stałym rozmiarze w językach *C/C++*

Typ	**Rozmiar [B]**	**Zakres wartości**
int8_t	1	[−128; +127]
int16_t	2	[−32768; +32767]
int32_t	4	[−2147483648 +2147483647]
int64_t	8	[−9223372036854775808, +9223372036854775807]
uint8_t	1	[0; +255]
uint16_t	2	[0; 65535]
uint32_t	4	[0; 4294967295]
uint64_t	8	[0; 18446744073709551615]

2.6. Więcej o funkcjach

Najprostsza funkcja w językach *C/C++* nie pobiera ani nie zwraca żadnych wartości; wówczas typ ***void*** ma zastosowanie zarówno w odniesieniu do typu zwracanego, jak i listy argumentów w nagłówku funkcji. W przypadku listy argumentów dozwolone jest pominięcie słowa kluczowego ***void***, pozostawiając puste nawiasy okrągłe:

```
1 // Rownowaznie z void sayHello()
2 void sayHello(void) {
3     printf("Hello, world!");
4 }
```

Ponieważ funkcja **sayHello()** nie zwraca żadnej wartości, to nie został zastosowany operator ***return*** (por. z funkcją ***main()*** z listingu 2.). Wywołanie funkcji **sayHello()** z ciała funkcji ***main()*** wyglądałoby następująco:

```
1 int main() {
2     sayHello();
3     return 0;
4 }
```

Jeżeli dana funkcja wywołuje w swoim ciele inną funkcję, to należy zachować odpowiednią kolejność deklaracji funkcji. Oznacza to, że funkcja wywoływana (tu: **sayHello()**) musi być znana dla kompilatora na etapie jej wywołania. W przeciwnym razie próba kompilacji kodu źródłowego może zakończyć się błędem. Przykład przedstawiono na listingu 6. Rozwiązaniem tego problemu byłoby przeniesienie definicji funkcji **sayHello()** przed definicję funkcji ***main()*** lub zastosowanie tzw. ***deklaracji zapowiadającej***. ***Deklaracja zapowiadająca*** polega na rozdzieleniu deklaracji i definicji funkcji i umieszczeniu deklaracji przed definicją funkcji wywołującej (tu: ***main()***), co przedstawiono na listingu 7.

```c
#include <stdio.h>

int main() {
    // Pierwsze wystapienie symbolu sayHello w kodzie
  sayHello();
  return 0;
}

// Definicja (i deklaracja) funkcji sayHello()
void sayHello() {
  printf("Hello, world!");
}
```

Listing 6. Błędna kolejność deklaracji funkcji

```c
#include <stdio.h>
// Deklaracja zapowiadajaca funkcji sayHello()
void sayHello();

int main() {
    // Symbol sayHello zostal zdeklarowany wczesniej
  sayHello();
  return 0;
}

// Definicja funkcji sayHello()
void sayHello() {
  printf("Hello, world!");
}
```

Listing 7. Zastosowanie deklaracji zapowiadającej

Funkcje zwracające wartość w wyniku swojego działania (o typie zwracanym różnym od ***void***) są podatne na jeszcze jeden, dość trudny do wykrycia błąd, a mianowicie na pominięcie instrukcji ***return*** w ciele funkcji. Przykład takiej sytuacji zobrazowano na listingu 8. Funkcja **multiply()** przeprowadza mnożenie dwóch liczb (całkowitej **multiplier** i zmiennoprzecinkowej **multiplicand**), a wynik mnożenia zapisywany jest pod zmienną **result**. W ciele funkcji pominięto użycie operatora ***return*** (**return result;**). Funkcja ***main()*** wywołuje funkcję **multiply()** z parametrami: 3 (**multiplier**) oraz 1,23 (**multiplicand**), a wartość zwróconą z funkcji zapisuje pod zmienną **value**. Warto w tym miejscu zwrócić uwagę, że separatorem dziesiętnym w językach *C/C++* jest kropka, a nie przecinek. Zapisana wartość jest następnie wyświetlana na konsoli. Zgodnie z deklaracją funkcji **multiply()**, z funkcji zostanie zwrócona liczba zmiennoprzecinkowa (***float***). Ponieważ definicja funkcji nie zawiera instrukcji ***return***, to wynikiem jej działania jest ***niezdefiniowane zachowanie***. Wartość zapisywana pod zmienną **value** jest nie do przewidzenia. Takie działanie programu może być bardzo niekorzystne, a wykrycie błędu w przypadku rozbudowanych projektów programistycznych niezwykle uciążliwe. Co gorsza, program przedstawiony na listingu 8. może ze znacznym prawdopodobieństwem skompilować się bez żadnych ostrzeżeń o brakującej instrukcji ***return*** (zachowanie zależne od kompilatora). W sprzyjających okolicznościach kompilator zgłosi ostrzeżenie **warning: control reaches end of non-void function [-Wreturn-type]**.

```c
#include <stdio.h>

float multiply(int multiplier, float multiplicand) {
  // Obliczenie wyniku mnozenia
  float result = multiplier * multiplicand;
  // Brak instrukcji return - wynik (result) nie
   jest zwracany z funkcji
}

int main() {
  // Wywolanie funkcji multiply() z parametrami 3 i
   1.23
  // Przypisanie wartosci zwroconej z funkcji
   multiply() do zmiennej value
  float value = multiply(3, 1.23);
  // Niezdefiniowane zachowanie
  printf("Value=%f", value);
  return 0;
}
```

Listing 8. Pominięcie instrukcji return w ciele funkcji

Na szczęście przed przedstawioną sytuacją można się za wczasu zabezpieczyć stosując odpowiednią ***flagę kompilatora***. ***Flagi kompilatora*** to dodatkowe opcje przekazywane do kompilatora w procesie tworzenia pliku wykonywalnego aplikacji. Za ich pomocą można m.in. zaostrzać kryteria, wg których określone działanie (lub zaniechanie działania, jak na listingu 8.) będzie traktowane jako błąd kompilacji. Niektóre z nich zostały przedstawione w rozdziale *2.3 Proces budowania aplikacji w językach C/C++*. Przykłady popularnych ***flag kompilatorów gcc*** i ***g++*** zestawiono w tabeli 3.

Tabela 3. Przykłady flag kompilatorów *gcc* (język *C*) i *g++* (język *C++*)

Flaga kompilatora	Opis
-Wall	Zwiększ liczbę zgłaszanych ostrzeżeń[2]
-Wextra	Zgłaszaj dodatkowe ostrzeżenia nieobsługiwane przez flagę **-Wall**[2]
-Werror	Traktuj każde ostrzeżenie jako błąd kompilacji
-Wunused	Ostrzegaj o nieużywanych zmiennych i funkcjach
-O	Określ stopień optymalizacji kodu przez kompilator (od 0 do 3)
-g	Zbuduj plik wykonywalny z symbolami debugowymi
-S	Zatrzymaj proces budowania pliku wykonywalnego przed asemblacją
-c	Zatrzymaj proces budowania pliku wykonywalnego przed konsolidacją
-o	Określ nazwę (ścieżkę do) pliku wynikowego

Aby zabezpieczyć się przed sytuacją z listingu 8. należy posłużyć się flagami **-Wall** oraz **-Werror**:

gcc -Wall -Werror main.c

Wówczas próba kompilacji zakończy się błędem **error: control reaches end of non-void function** [**-Werror=return-type**].

2.7. Operator *sizeof*

Zarówno język *C*, jak i *C++*, zawierają zbiór tzw. ***słów kluczowych***. Są to zarezerwowane symbole, o określonym znaczeniu, których nie można wykorzystać jako np. nazw zmiennych lub funkcji. Przykładami słów kluczowych mogą być, poznane już wcześniej, typy zmiennych (***int***, ***float***, ***double***, ...), ale również ***operatory*** (czyli +, -, *, ...), których reprezentantem jest ***operator sizeof***. ***Operator sizeof*** zwraca rozmiar (w bajtach) jaki dana zmienna zajmuje w pamięci komputera (listing 9). ***Operator sizeof*** może być wywołany zarówno na zmiennej, posługując się jej identyfikatorem

[2]Dokładny opis dostępny na stronie `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`

(nazwą) (tu: **sizeof(x)**), jak również na określonym typie zmiennej (tu: **sizeof(float)**). ***Operator sizeof*** jest jednym z najczęściej wykorzystywanych operatorów języków *C/C++*. Za jego pomocą można określać rozmiar zmiennych na różnych architekturach komputerów (większość typów zmiennych nie ma ściśle sprecyzowanego rozmiaru, patrz: tabela 1).

```
1 #include <stdio.h>
2
3 int x = 1;
4
5 int main() {
6     printf("Size of x: %d", sizeof(x));
7     printf("Size of float: %d", sizeof(float));
8     return 0;
9 }
```

Listing 9. Wywołanie operatora ***sizeof***

Niektóre z ***operatorów*** dostępnych w językach *C/C++* wymagają stosowania nawiasów do określania ich listy argumentów (***for***, ***while***, ***if***, ...), a inne nie (***return***, ***throw***, ...). W przypadku ***operatora sizeof*** stosowanie nawiasów jest dobrowolne, jeżeli przekazuje się zmienne z użyciem ich identyfikatorów (równoważnie ***sizeof x*** i ***sizeof(x)***), ale obligatoryjne w przypadku wywoływania ***operatora*** na typie zmiennej (***sizeof(float)***). Zarówno język *C*, jak i *C++*, dopuszczają stosowanie dodatkowych (redundantnych) nawiasów wokół wyrażeń, np. w celu ich „wizualnego grupowania". Zatem następujące zapisy są również poprawne:

```
1 int x = (((2)));
2 int y = (2 + (3 * 5));
3 return(0);
```

2.8. Komentarze

Dobrym nawykiem jest stosowanie wcięć i pozostawianie w kodzie źródłowym komentarzy, ułatwiających późniejszą analizę kodu i zwiększających jego czytelność. Komentarze to linie kodu, które zostaną zignorowane przez kompilator w procesie tłumaczenia kodu źródłowego na kod asemblera. Aby dodać komentarz należy rozpocząć linię od podwójnego ukośnika (//). Aby objąć komentarzem więcej niż jedną linię kodu, należy rozpocząć komentarz ciągiem /*, a zakończyć przez */ (listing 10).

```
1 // Zmienna typu calkowitego
2 int x = 4;
3
4 int main() {
5     /* Wyswietlanie wartosci zmiennej x
6         z uzyciem funkcji printf() */
7     printf("Value of x: %d", x);
8     return 0;
9 }
```

Listing 10. Przykłady komentarzy w językach *C/C++*

3. Program ćwiczenia

Do edycji kodu źródłowego wystarcza najprostszy notatnik w systemie operacyjnym. Warto jednak, by program edycji tekstu wspierał *kolorowanie składni* języka programowania, jak np. popularny *Notepad++*. Ułatwia to proces tworzenia kodu i zwiększa jego czytelność dla użytkownika (programisty). Proces kompilacji i konsolidacji kodu może być przeprowadzony z poziomu konsoli systemowej. Popularne, szczególnie przy większych projektach programistycznych, są tzw. *zintegrowane środowiska programistyczne* (*ang.* ***I****ntegrated* ***D****evelopment* ***E****nvironment, IDE*), zapewniające zarówno edytor kodu, jak i wbudowany kompilator, konsolidator, narzędzia statycznej lub dynamicznej analizy kodu, itp. Popularne IDE wykorzystywane komercyjnie do tworzenia aplikacji w językach *C* i *C++* to m.in. *Eclipse*, *CLion*, *QtCreator* czy *Visual Studio*. W ramach naszych zajęć będziemy wykorzystywać prosty edytor tekstu *Notepad++*.

Zadanie 1. Zbuduj i uruchom aplikację, która wypisze na konsolę twoje imię i nazwisko. Korzystając z edytora tekstu porównaj kod źródłowy aplikacji (***main.c***) z kodem źródłowym asemblera (***main.s***).

Zadanie 2. Utwórz plik nagłówkowy ***defs.h*** zawierający deklaracje dwóch zmiennych typu zmiennoprzecinkowego pojedynczej precyzji i funkcji wykonującej dodawanie dwóch liczb zmiennoprzecinkowych (*add()*). Utwórz odpowiadający mu plik źródłowy ***defs.c***, zawierający odpowiednie definicje tych zmiennych i funkcji. Wywołaj funkcję *add()* wewnątrz funkcji ***main()***, wykorzystując dwie zdefiniowane wcześniej zmienne i wypisz wynik dodawania na konsolę.

Zadanie 3. Sprawdź i wypisz na konsolę rozmiar różnych podstawowych typów zmiennych wykorzystując operator ***sizeof***.

4. Dodatek

4.1. Konwencje nazewnictwa

Wyróżnia się wiele konwencji nazewnictwa zmiennych i funkcji w językach *C/C++*. Do najczęściej stosowanych należą m.in. **camelCase** oraz **snake_case**. Obie konwencje zakładają, że nazwy stałych, zmiennych czy funkcji zapisujemy małą literą. Jeżeli nazwa składa się z więcej niż jednego słowa, to:

- w konwencji **camelCase** każdy następny człon nazwy rozpoczynamy wielką literą (np. *floatingPointValue*);
- w konwencji **snake_case** każdy następny człon nazwy rozpoczynamy małą literą i łączymy znakiem podkreślenia (np *floating_point_value*).

Identyfikatory (nazwy) zmiennych i funkcji w językach *C*, jak i *C++*, mogą składać się z liter, znaku podkreślenia i cyfr, przy czym pierwszy znak nie może być cyfrą. Zmienne i funkcje powinny mieć jednoznaczne nazwy, sugerujące ich przeznaczenie.

Wybór konwencji jest najczęściej kwestią upodobania. Należy jedynie pamiętać, żeby po wybraniu konwencji nazewnictwa trzymać się jej zasad i nie mieszać różnych konwencji wewnątrz jednego projektu aplikacji.

4.2. Liczby zmiennoprzecinkowe

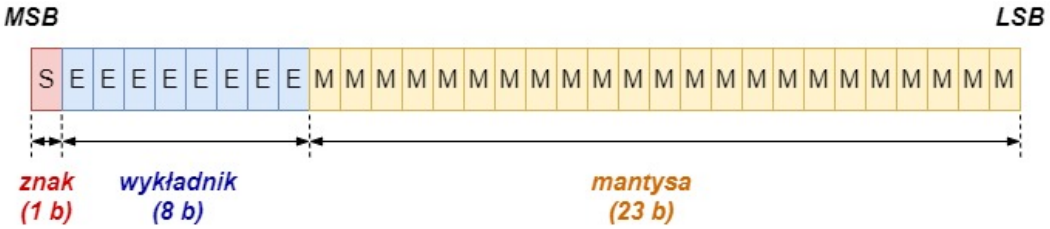
Zgodnie ze standardem IEEE 754 sprzętowa reprezentacja liczb zmiennoprzecinkowych składa się z trzech pól o określonej długości bitowej: ***znaku*** (S), ***wykładnika*** (E) i ***mantysy*** (M), które kodują wartość liczby (x) zgodnie z równaniem (1).

$$x = (-1)^S \cdot M \cdot 2^{E-bias} \tag{1}$$

Standard wyróżnia liczby pojedynczej (typ ***float*** w językach $C/C++$; typowo 32 bity) oraz podwójnej precyzji (typ ***double*** w językach $C/C++$; typowo 64 bity). Pierwszy bit stanowi ***znak***, którego wartość **0** koduje liczbę dodatnią ($(-1)^0$, patrz: równanie (1)), a **1** liczbę ujemną ($(-1)^1$). Kolejne bity stanowią odpowiednio ***wykładnik*** oraz ***mantysę***. ***Mantysa*** to liczba zawarta w przedziale **[1,2)**, przy czym część całkowita liczby nie jest kodowana (jest zawsze równa 1). Zapisywane jest jedynie jej rozszerzenie dziesiętne na odpowiednio **23** (w przypadku liczb zmiennoprzecinkowych pojedynczej precyzji) lub **52** bitach (w przypadku liczb zmiennoprzecinkowych podwójnej precyzji). ***Wykładnik*** zapisywany jest w ***kodzie spolaryzowanym***, tzn. jego wartość jest przesunięta o określoną stałą (***bias***): **127** dla liczb zmiennoprzecinkowych pojedynczej precyzji i **1023** dla liczb zmiennoprzecinkowych podwójnej precyzji [1]. Tabela 4. zawiera długości poszczególnych pól składających się na reprezentację liczby zmiennoprzecinkowej wg standardu IEEE 754. Na rys. 4.1. przedstawiono reprezentację liczby zmiennoprzecinkowej pojedynczej precyzji. Należy mieć na uwadze, że nie wszystkie mikroprocesory posiadają wsparcie sprzętowe dla obliczeń zmiennoprzecinkowych (*ang. floating-point unit, FPU*), np. mikroprocesory 8-bitowe. Arytmetyka zmiennoprzecinkowa jest na nich możliwa z wykorzystaniem implementacji software'owej (*ang. software floating-point*), jednakże wiąże się to ze zwiększonym wykorzystaniem pamięci. Co więcej, takie rozwiązania są z reguły wolniejsze niż implementacja sprzętowa.

Tabela 4. Rozmiar liczb zmiennoprzecinkowych w pamięci komputera [1]

Format	**Znak [b]**	**Wykładnik [b]**	**Mantysa [b]**
Pojedyncza precyzja	1	8	23
Podwójna precyzja	1	11	52



Rys. 4.1. Reprezentacja liczby zmiennoprzecinkowej pojedynczej precyzji [1]

Literatura

[1] IEEE. "IEEE-754, Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Jan. 2008).