



Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 9. Typowy układ pamięci programu i operacje na pamięci

Zagadnienia do opracowania:

- architektura harwardzka i architektura von Neumanna
- segmenty pamięci, ich przeznaczenie i ułożenie w przestrzeni adresowej komputera
- struktury
- aliasy typów
- operacje na pamięci w językach *C* i *C++*

Spis treści

1	Cel ćwiczenia	2
2	Wprowadzenie	2
2.1	Architektura komputerów a organizacja pamięci	2
2.2	Układ pamięci programu	4
2.2.1	Stos	8
2.2.2	Szereg	12
2.2.3	.bss	14
2.2.4	.data	16
2.2.5	Text (code)	19
2.3	Struktury	21
2.4	Operacje na pamięci	29
3	Program ćwiczenia	33
4	Dodatek	36
4.1	Kolejność bajtów	36
4.2	Organizacja danych w strukturach	37
4.3	Unie	41

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z typowym układem pamięci programu – przeznaczeniem i ułożeniem w przestrzeni adresowej komputera poszczególnych segmentów pamięci oraz podstawowymi operacjami na pamięci.

2. Wprowadzenie

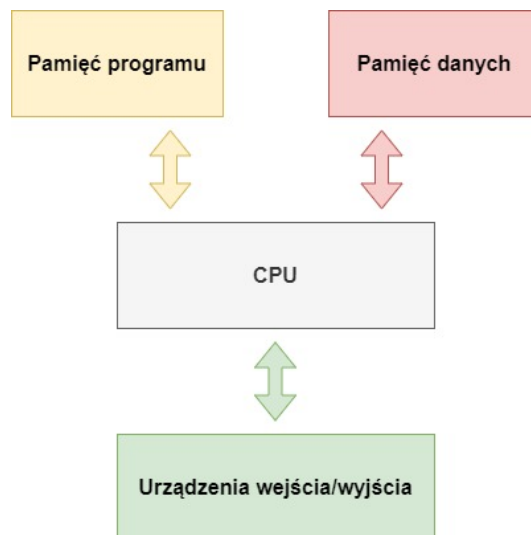
2.1. Architektura komputerów a organizacja pamięci

Architektura komputera ma wpływ na organizację pamięci i szybkość wykonywania operacji. Ze względu na sposób organizacji pamięci w komputerze można wyróżnić trzy podstawowe architektury:

- *harwardzką*;
- *von Neumanna (Princeton)*;
- *zmodyfikowaną harwardzką (mieszaną)*.

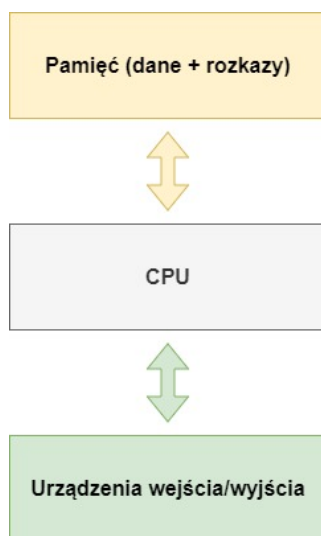
Architektura harwardzka charakteryzuje się rozdzieloną pamięcią programu (rozkazów) i pamięcią danych (rozdzielona przestrzeń adresowa rozkazów i danych). Dzięki temu procesor może jednocześnie pobierać dane i rozkazy, co skutkuje zwiększeniem szybkości wykonywanych operacji. Jednakże takie rozwiązanie uniemożliwia zmianę wykonywanego na komputerze programu (brak możliwości zapisu nowych instrukcji do pamięci programu) – komputer dostarczany jest ze stałym oprogramowaniem. **Architektura harwardzka** została schematycznie przedstawiona na rys. 2.1. **Architektura harwardzka** była stosowana w początkowych generacjach komputerów (np. *Harvard Mark I*), a obecnie wykorzystywana jest głównie w mikrokontrolerach jednokomórkowych oraz procesorach sygnałowych.

Zmodyfikowana architektura harwardzka (mieszana) charakteryzuje się rozdzieloną pamięcią programu i pamięcią danych, ale wykorzystuje wspólne magistrale: danych i adresową. Dzięki temu możliwe jest pobieranie rozkazów, analogicznie jak w przypadku danych.



Rys. 2.1. Model architektury harwardzkiej

Architekturę von Neumanna, zwaną również **architekturą Princeton**, wyróżnia wspólna pamięć dla instrukcji i danych. Możliwość wielokrotnego programowania komputera została zrealizowana kosztem jednoczesnego pobierania rozkazów i danych. Schemat **architektury von Neumanna** przedstawiono na rys. 2.2. **Architektura von Neumanna** jest powszechnie stosowana w konstrukcji komputerów uniwersalnych.



Rys. 2.2. Model architektury von Neumanna

Dalsza część tej instrukcji będzie odnosić się do komputera o ***architekturze von Neumanna***.

2.2. Układ pamięci programu

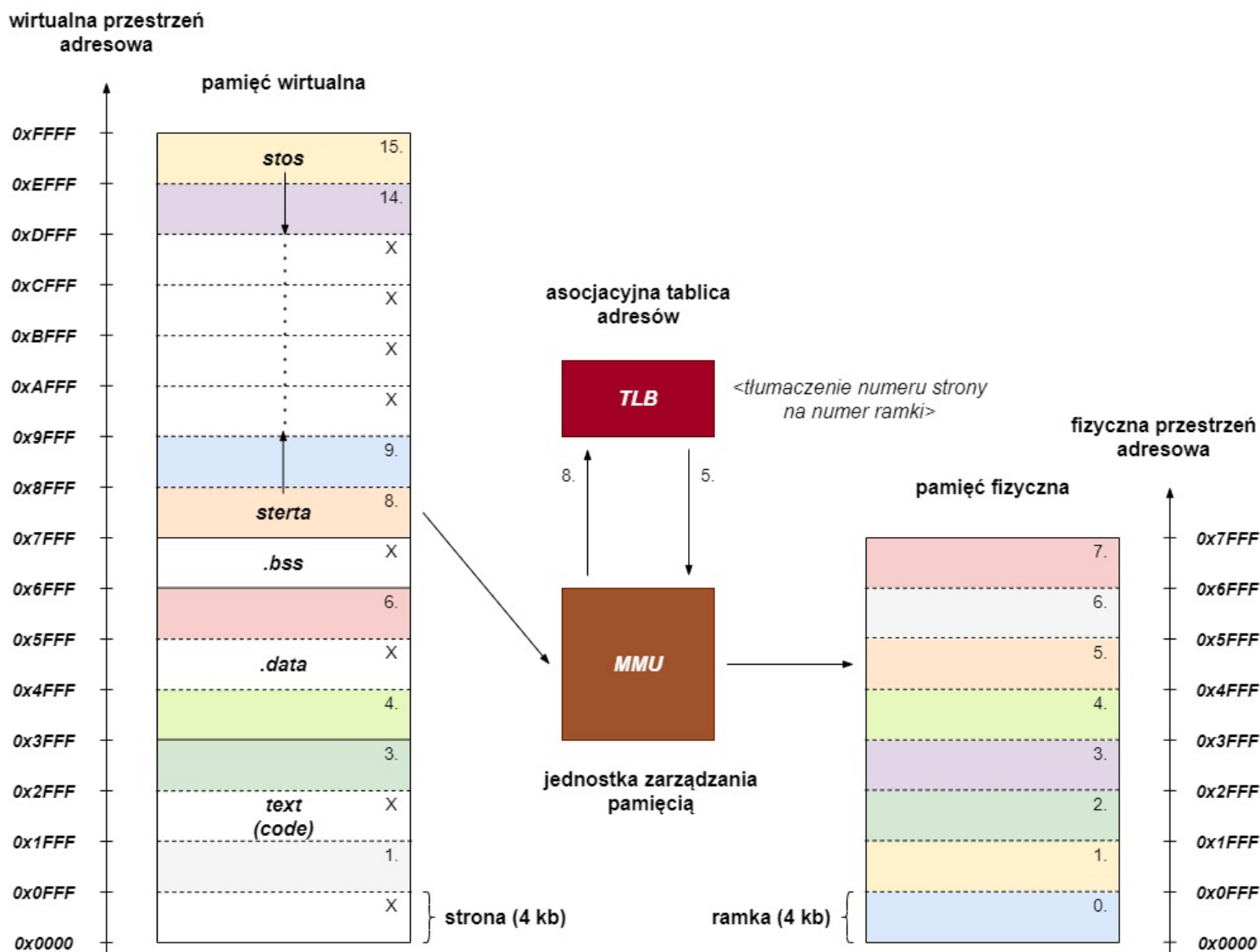
Standard języków *C/C++* nie określa jak powinien wyglądać układ pamięci programu, delegując odpowiedzialność za mapowanie zmiennych na odpowiednie segmenty pamięci do określonej implementacji kompilatora. Standard języka zakłada, że:

- w momencie deklaracji **zmiennej lokalnej** automatycznie alokowana jest potrzebna pamięć;
- pamięć zaalokowana za pomocą ***malloc()*** (***new***) może zostać zwolniona za pomocą ***free()*** (***delete***);
- w przypadku **zmiennych statycznych** pamięć alokowana jest przed wywołaniem funkcji ***main()***.

Jednakże na podstawie najpopularniejszych implementacji kompilatorów i architektur procesorów można podjąć próbę przedstawienia **typowego układu**

pamięci programu.

W pierwszej kolejności należy jednak uzmysłwić sobie, że program komputerowy **nie operuje na fizycznych adresach pamięci**. Układ elektroniczny nazywany **jednostką zarządzania pamięcią** (*ang. memory management unit, MMU*) przydziela programowi jego własną **wirtualną przestrzeń adresową** (*ang. virtual address space, VAS*) – **adresy logiczne**. Pamięć ta jest podzielona na bloki o stałej długości zwane **stronami** (*ang. virtual pages*). **Strony** stanowią ciągłe zakresy adresów pamięci, a ich rozmiar wynosi typowo od 512 B do 1 GB. **Pamięć fizyczna** podzielona jest analogicznie na bloki zwane **ramkami** (*ang. page frames*), których rozmiar jest równy rozmiarowi **stron**. **Strony pamięci wirtualnej** są mapowane na **ramki pamięci fizycznej** przez **MMU**. W wyniku tego pamięć, która z punktu widzenia aplikacji jest ciągła (np. w przypadku zmiennych tablicowych), w rzeczywistości może być podzielona na różne sektory pamięci **RAM**. Translację adresów logicznych na fizyczne schematycznie przedstawiono na rys. 2.3. W tym przykładzie rozmiar **stron**, a więc i **ramek**, wynosi 4 kb (4096 b). Pierwsza **strona** zawiera adresy w przedziale 0x0000 – 0x0FFF (4095), druga 0x1000 (4096) – 0x1FFF (8191), itd. **Wirtualna przestrzeń adresowa** ma rozmiar 64 kb (16 **stron**), natomiast **fizyczna przestrzeń adresowa** ma rozmiar 32 kb (8 **ramek**). Oznacza to, że **nie wszystkie strony** muszą być jednocześnie zmapowane na **ramki**, aby uruchomić program [9]. Mapowanie **pamięci wirtualnej** na **pamięć fizyczną** z rys. 2.3. przedstawiono w tabeli 1. Rzecz jasna, w rzeczywistych systemach operacyjnych przestrzenie adresowe są dużo większe niż w opisanym przykładzie (rzędu Gb).



Rys. 2.3. Translacja adresów logicznych na fizyczne przez jednostkę zarządzania pamięcią (MMU)

Tabela 1. Mapowanie stron pamięci wirtualnej na ramki pamięci fizycznej
(na podstawie rys. 2.3.)

Nr strony	Zakres adresów wirtualnych	Nr ramki	Zakres adresów fizycznych
0	0x0000 – 0x0FFF	–	–
1	0x1000 – 0x1FFF	6	0x6000 – 0x6FFF
2	0x2000 – 0x2FFF	–	–
3	0x3000 – 0x3FFF	2	0x2000 – 0x2FFF
4	0x4000 – 0x4FFF	4	0x4000 – 0x4FFF
5	0x5000 – 0x5FFF	–	–
6	0x6000 – 0x6FFF	7	0x7000 – 0x7FFF
7	0x7000 – 0x7FFF	–	–
8	0x8000 – 0x8FFF	5	0x5000 – 0x5FFF
9	0x9000 – 0x9FFF	0	0x0000 – 0x0FFF
10	0xA000 – 0xAFFF	–	–
11	0xB000 – 0xBFFF	–	–
12	0xC000 – 0xCFFF	–	–
13	0xD000 – 0xDFFF	–	–
14	0xE000 – 0xEFFF	3	0x3000 – 0x3FFF
15	0xF000 – 0xFFFF	1	0x1000 – 0x1FFF

Kiedy program komputerowy odwołuje się do *adresu pamięci wirtualnej*, *MMU* przeprowadza jego tłumaczenie z wykorzystaniem asocjacyjnej tablicy adresów *TLB* (*ang. translation lookaside buffer*). Jeżeli program odwoła się do *wirtualnego obszaru pamięci*, który nie ma mapowania w pamięci fizycznej (symbol **X** na rys. 2.3.), to system operacyjny dostaje sygnał *page fault*. W wyniku tego przeprowadzane jest mapowanie zadanej *strony w pamięci wirtualnej* na aktualnie niewykorzystywaną *ramkę w pamięci fizycznej*, a przerwana instrukcja wykonywana jest ponownie. Dla każdej *strony MMU* przechowuje informacje czy istnieje mapowanie na *pamięć fizyczną* oraz czy jest ona w użyciu (nastąpiło odwołanie do

adresu leżącego w zakresie **strony**). Służą do tego odpowiednio **bit obecności** (*ang. present bit*) oraz **bit użycia** (*ang. accessed bit*).

Pamięć cyfrową, udostępnianą programowi przez komputer (lub mikrokontroler), można podzielić na ulotną **pamięć o dostępie swobodnym** (*ang. random-access memory, RAM*) oraz nieulotną **pamięć tylko do odczytu** (*ang. read-only memory, ROM*). W przypadku komputerów osobistych (*PC*) cała wirtualna przestrzeń adresowa leży najczęściej w obrębie pamięci **RAM**; niektóre segmenty mogą być dodatkowo oznaczone jako tylko do odczytu. W przypadku mikrokontrolerów popularne jest umieszczanie skompilowanego kodu maszynowego i wyrażeń stałych (segmenty **.rodata** i **text**) wewnątrz pamięci **ROM**, którą stanowi najczęściej pamięć **Flash** lub **EEPROM** (*ang. electrically erasable programmable ROM*) – pamięć kasowalna elektrycznie. Pamięć **RAM** na mikrokontrolerach ma często ograniczony rozmiar w stosunku do pamięci tylko do odczytu. Z tego względu **bardzo istotna jest optymalizacja kodu przez stosowanie specyfikatora *const*** z wyrażeniami, których wartość jest stała przez cały czas trwania programu, ponieważ mogą być wówczas przechowywane w pamięci **ROM**.

2.2.1. Stos

Stos (*ang. stack*) to segment pamięci zrealizowany w postaci **liniowej struktury danych**, w której elementy są dodawane i usuwane z jej wierzchołka. Adres ostatnio dodanej zmiennej wskazywany jest przez **wskaźnik wierzchołka stosu** (*ang. stack pointer*). Zmienne alokowane na **stosie** są zwalniane automatycznie, bez ingerencji programisty. Na **stosie** odkładane są:

- zmienne lokalne alokowane statycznie (*ale nie zmienne statyczne!*),
- adresy powrotu z funkcji,
- kopie argumentów wywołania funkcji (w tym argumenty funkcji *main()*),

-
- wartości zwracane z funkcji.

Ostatni element umieszczony na *stosie* jest zarazem pierwszym elementem, który zostanie ze *stosu* ściągnięty. Stanowi to realizację bufora **LI-FO** (*ang. Last In, First Out*). Zostało to zaprezentowane na listingu 1. Zakres wartości umieszczonych na *stosie* w ramach wywołania pojedynczej funkcji (w tym również adres powrotu z funkcji) nazywa się *ramką stosu* (*ang. stack frame*).

```
1 #include <iostream>
2
3 void displayStackFrame(int a, float b, short c) {
4     int x = 2;
5     float y = 5.5;
6     short z = 112;
7     std::cout << "Print variables from stack frame"
8     << std::endl;
9     std::cout << &x << std::endl;
10    std::cout << &y << std::endl;
11    std::cout << &z << std::endl;
12    std::cout << &a << std::endl;
13    std::cout << &b << std::endl;
14    std::cout << &c << std::endl;
15 }
16
17 void displayStackFrame2(int a, float b, short c) {
18     int x = 3;
19     float y = 2.0;
20     short z = 0;
21     std::cout << "Print variables from stack frame"
22     << std::endl;
23     std::cout << &x << std::endl;
```

```

22     std::cout << &y << std::endl;
23     std::cout << &z << std::endl;
24     std::cout << &a << std::endl;
25     std::cout << &b << std::endl;
26     std::cout << &c << std::endl;
27 }
28
29 int main() {
30     displayStackFrame(0, 2.0, 5);
31     displayStackFrame2(3, 5.5, 1);
32     return 0;
33 }

```

Listing 1. Stos jako bufor LIFO

Wynik wywołania programu z listingu 1. może wyglądać następująco:

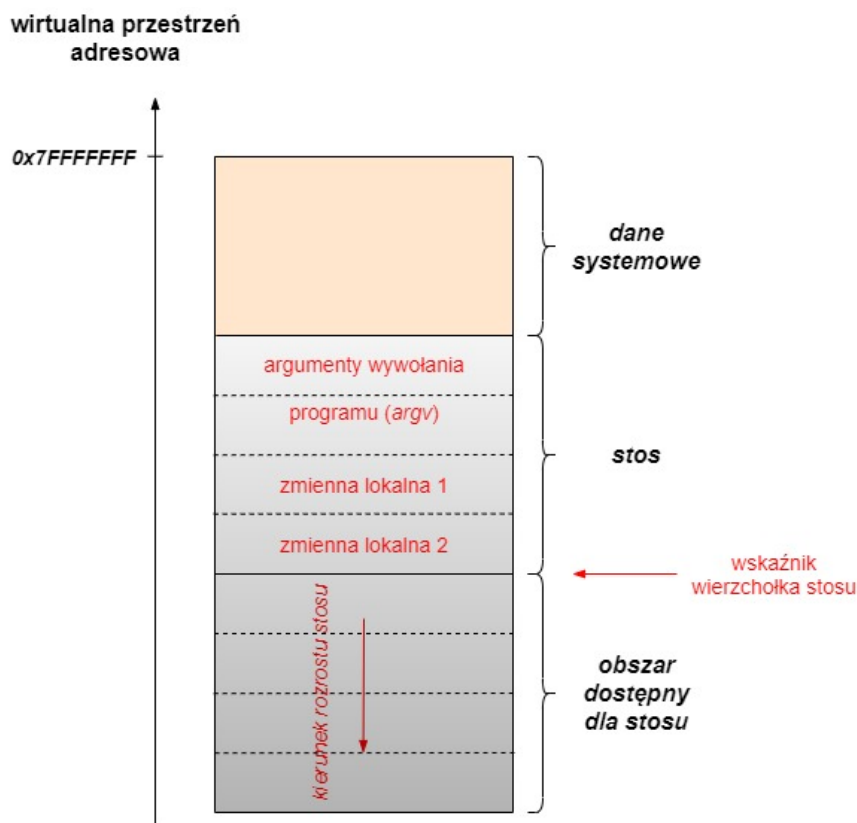
```

1 Print variables from stack frame
2 0x7ffffb1c
3 0x7ffffb18
4 0x7ffffb16
5 0x7ffffb0c
6 0x7ffffb08
7 0x7ffffb06
8 Print variables from stack frame
9 0x7ffffb1c
10 0x7ffffb18
11 0x7ffffb16
12 0x7ffffb0c
13 0x7ffffb08
14 0x7ffffb06

```

Widać stąd, że po wywołaniu funkcji ***displayStackFrame()*** na stosie została zaalokowana pamięć dla trzech zmiennych lokalnych, kolejno pod adresami: 0x7ffffb1c, 0x7ffffb18 i 0x7ffffb16 oraz dla trzech kopii argumentów wywołania funkcji (adresy: 0x7ffffb0c, 0x7ffffb08 i 0x7ffffb06). Po wyjściu z funkcji pamięć została zwolniona (***wskaźnik wierzchołka stosu*** został przesunięty z powrotem przed adres 0x7ffffb1c). Dzięki temu po wywołaniu ***displayStackFrame2()*** zmienne lokalne oraz kopie argumentów w obrębie tej funkcji alokowane są pod tymi samymi adresami pamięci.

Najpopularniejsza implementacja stosu zakłada jego lokalizację w **górnym zakresie wirtualnej przestrzeni adresowej**. Rozmiar pamięci ***stosu*** jest znany na etapie kompilacji. Kolejne elementy dodawane na ***stos*** powodują jego rozrost w **kierunku niższych adresów**. Zostało to schematycznie przedstawione na rys. 2.4. W tym przypadku przestrzeń adresowa rozciąga się między adresami 0x00000000 a 0x7FFFFFFF (dziesiętnie 2147483647 – maksymalna wartość, jaką może przyjmować czterobajtowa zmienna typu *signed int*). W pierwszej kolejności na ***stosie*** alokowane są **argumenty wywołania programu** – argumenty, z którymi została wywołana funkcja *main()* (*int argc, char ** argv*). Następnie alokowane są kolejno deklarowane zmienne lokalne, kopie argumentów wywołania innych funkcji, ich wartości zwracane i adresy powrotu. Możliwe jest **przepiętnienie stosu** (*ang. stack overflow*) w wyniku przekroczenia obszaru pamięci przeznaczonego na rozrost ***stosu***. Może do niego dojść np. w wyniku zbyt długiego ciągu rekurencyjnych wywołań funkcji lub deklaracji lokalnej tablicy zmiennych o zbyt dużym rozmiarze. W przypadku programów uruchamianych w ramach systemu operacyjnego na szczycie przestrzeni adresowej (nad ***stosem***) umiejscowione są **dane systemowe** [8].

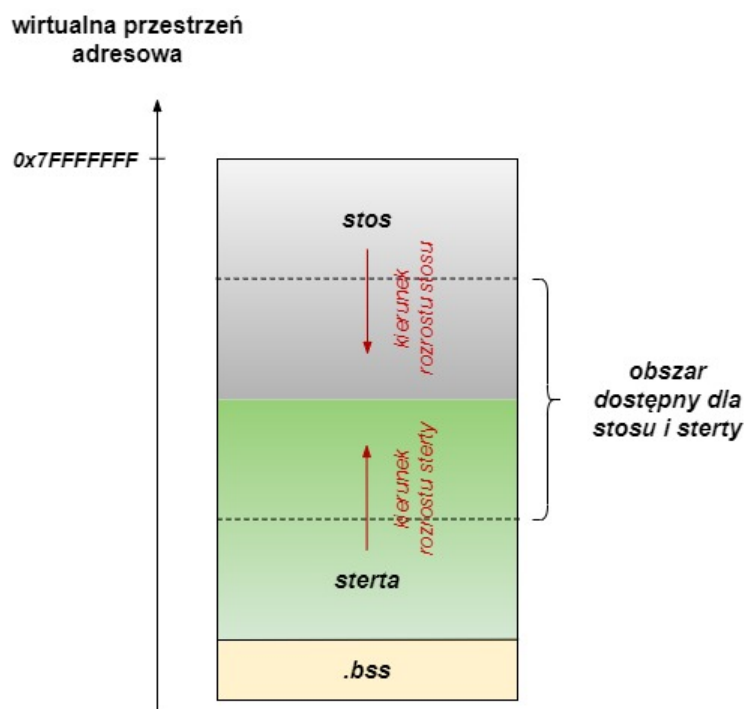


Rys. 2.4. Najpopularniejsze położenie stosu w przestrzeni adresowej komputera (mikrokontrolera)

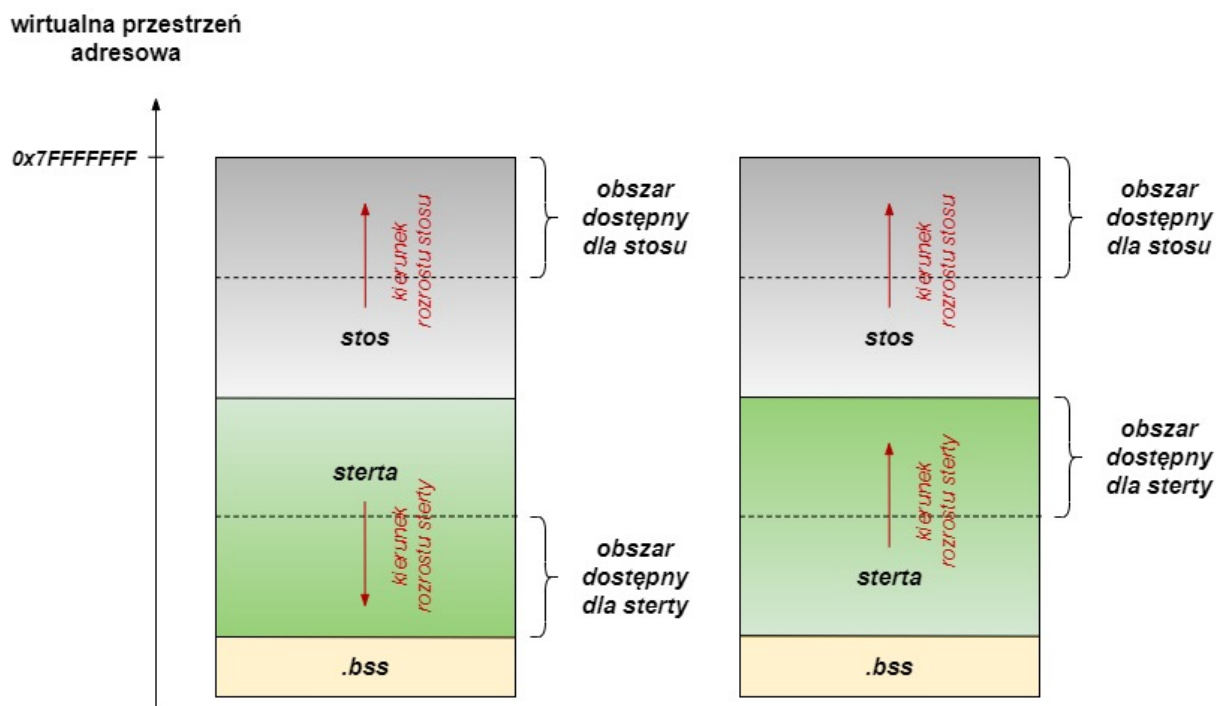
2.2.2. Sterta

Sterta (ang. *heap*) to segment pamięci przechowujący **zmienne alokowane dynamicznie**, stąd rozmiar pamięci alokowanej na **stercie** nie jest znany na etapie kompilacji programu. Dane alokowane na **stercie** nie są uporządkowane, ani czyszczone w sposób automatyczny (między wywołaniami funkcji), jak ma to miejsce w przypadku **stosu**. Możliwa jest fragmentacja pamięci **sterty**. Konstrukcja **sterty** powoduje, że operacje na zmiennych zaalokowanych na niej są z reguły **wolniejsze** niż w przypadku zmiennych zaalokowanych na **stosie** (zmienne alokowane na **stercie** znajdują się dalej od pamięci podręcznej procesora – **cache**). **Najczęściej sterta** zaczyna się na końcu przestrzeni adresowej segmentu **.bss** i rozrasta się w **kierunku**

wyższych adresów (w kierunku *stosu*). *Szterta* współdzieli dostępny (na rozrost) obszar pamięci ze *stosem*. W związku z tym teoretycznie możliwe jest **nałożenie się segmentów *stosu* i *sterty*** w wyniku jej przepełnienia (*ang. heap overflow*). Względne ułożenie *stosu* i *sterty* w przestrzeni adresowej komputera (mikrokontrolera) zostało schematycznie przedstawione na rys. 2.5. Takie ułożenie obu segmentów pamięci ma swoje uzasadnienie ze względów optymalizacyjnych. Inne kombinacje położenia *stosu* i *sterty* znacznie ograniczałyby dostępną pamięć dla obu segmentów. Szczególnie znaczenie ma to w przypadku mikrokontrolerów, gdzie dostępny rozmiar pamięci **RAM** jest zazwyczaj dużo mniejszy w porównaniu do rozmiaru pamięci **ROM**. Inne (nieoptymalne) przykłady ułożenia *stosu* i *sterty* przedstawiono na rys. 2.6. Co więcej, inne rozwiązania wiążą się z niebezpieczeństwem nadpisania pozostałych segmentów pamięci przez przepełniony *stos* lub *stertę*.



Rys. 2.5. Najpopularniejsze położenie sterty w przestrzeni adresowej komputera (mikrokontrolera)



Rys. 2.6. Nieoptymalne położenie stosu i sterty

2.2.3. .bss

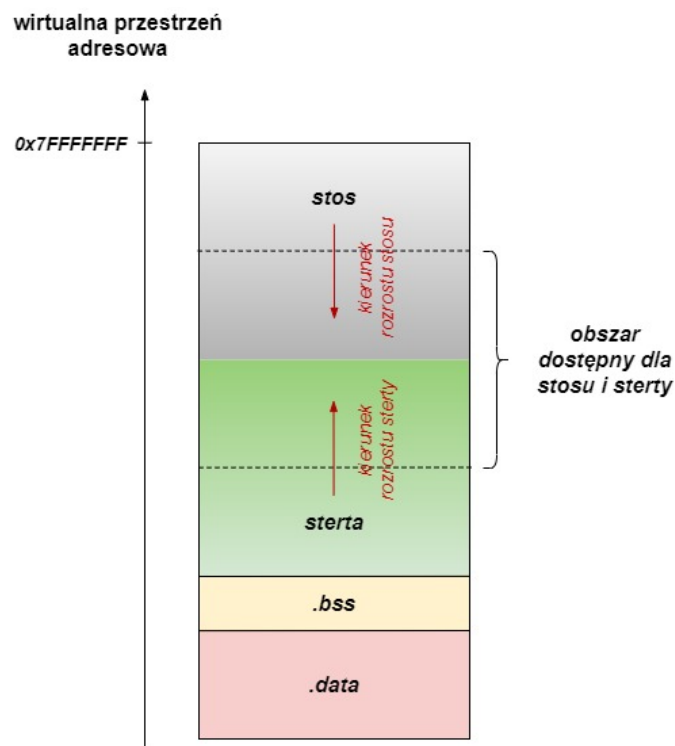
Segment **.bss** przechowuje **niezainicjalizowane zmienne globalne i statyczne oraz zmienne globalne i statyczne jawnie zainicjalizowane wartością 0**. Zgodnie ze standardem **ANSI C** globalne i statyczne zmienne są **automatycznie inicjalizowane wartością 0 przed wywołaniem funkcji *main()***. [Uwaga: w przypadku mikrokontrolerów jednowątkowych małych mocy spotykanym zabiegiem jest rezygnacja ze wsparcia standardu ANSI C. Wówczas nie jest przeprowadzana automatyczna inicjalizacja zmiennych globalnych i statycznych wartością 0. Umożliwia to skrócenie czasu wykonania programu.] Nazwa **.bss** wywodzi się od instrukcji języka assemblera, która rozwijana była jako *block started by symbol* [6]. Często plik obiektowy (rozszerzenie *.o*) nie przechowuje obrazu zmiennych z segmentu **.bss**, a jedynie jego rozmiar. Wynika to z faktu, że wszystkie zmienne przydzielone do segmentu **.bss** są niezainicjalizowane lub przyjmują wartość 0.

Na listingu 2. przedstawiono przykłady takich zmiennych.

```
1 // Niezainicjalizowana zmienna globalna
2 float x;
3 // Zmienna globalna zainicjalizowana do 0
4 int y = 0;
5 // Niezainicjalizowana statyczna zmienna globalna
6 static short z;
7 // Statyczna zmienna globalna zainicjalizowana do 0
8 static unsigned int w = 0;
9 // Zmienne umieszczone w anonimowej przestrzeni nazw
   sa rownowazne statycznym zmiennym globalnym
10 namespace {
11     // Zmienna niezainicjalizowana
12     double a;
13     // Zmienna zainicjalizowana do 0
14     long b = 0;
15 }
16
17 int main() {
18     // Niezainicjalizowana statyczna zmienna lokalna
19     static long long c;
20     // Statyczna zmienna lokalna zainicjalizowana do
   0
21     static unsigned short d = 0;
22     return 0;
23 }
```

Listing 2. Zmienne alokowane w segmencie .bss

Segment **.bss** położony jest między segmentem **.data** a **stertą** (rys. 2.7).



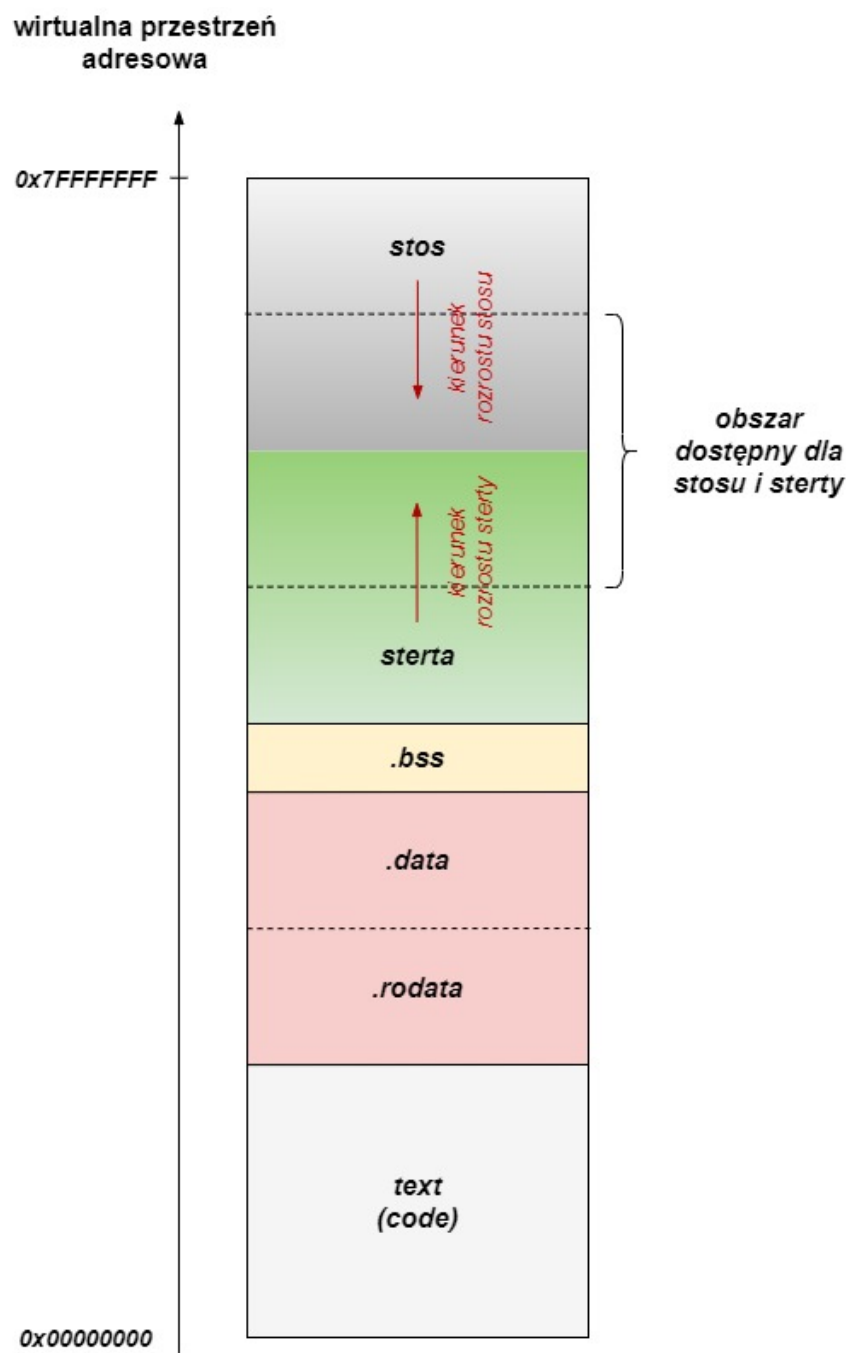
Rys. 2.7. Segment `.bss` w przestrzeni adresowej komputera (mikrokontrolera)

2.2.4. `.data`

Segment **`.data`** przechowuje **globalne i statyczne zmienne jawnie zainicjalizowane** przez programistę (wartością różną od 0). W segmencie **`.data`** można wydzielić obszar pamięci, w którym alokowane są zmienne tylko do odczytu – **`.rodata`** (*read-only data*). Alokowane są tam stałe. Segment **`.data`** położony jest w przestrzeni adresowej komputera między segmentem **`.bss`** a segmentem **`text`** (rys. 2.8). Rozmiar segmentu **`.data`** (a więc i **`.rodata`**) nie zmienia się w trakcie działania programu. Na listingu 3. przedstawiono przykłady zmiennych alokowanych w segmencie **`.data`**.

```
1 // Zainicjalizowana zmienna globalna
2 float x = 3.5;
3 // Zainicjalizowane statyczne zmienne globalne
4 static short y = 100;
5 namespace {
6     double z = 5.5;
7 }
8 // Globalna stala (.rodata)
9 const int a = 111;
10 // Statyczna globalna stala (.rodata)
11 static const float b = 4.0;
12
13 int main() {
14     // Zainicjalizowana statyczna zmienna lokalna
15     static long w = 666;
16     // Lokalna stala (.rodata)
17     const unsigned short c = 1;
18     return 0;
19 }
```

Listing 3. Zmienne alokowane w segmencie .data



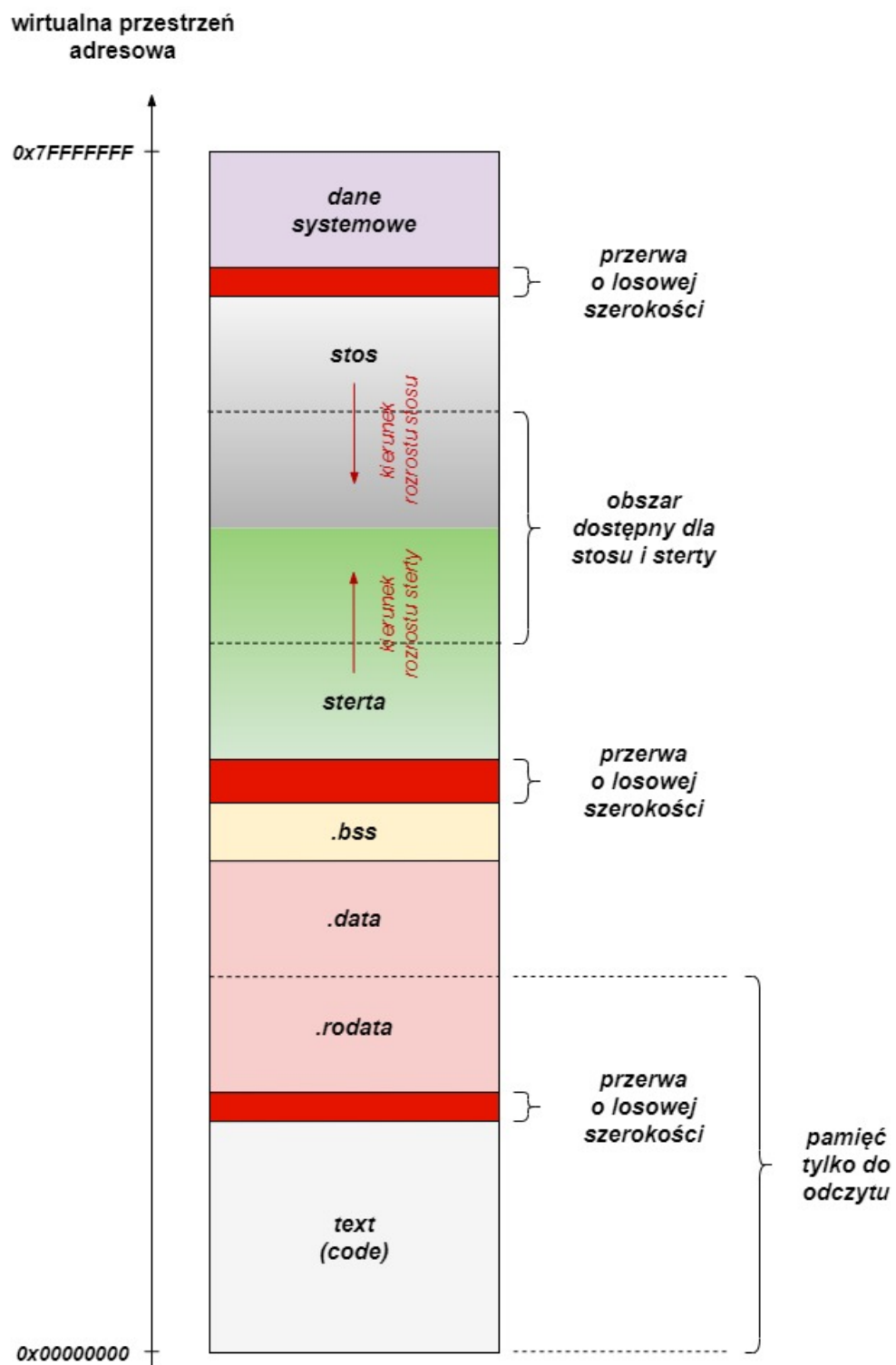
Rys. 2.8. Segment **.data** w przestrzeni adresowej komputera (mikrokontrolera)

2.2.5. Text (code)

Skompilowany (wykonywalny) kod maszynowy programu przechowywany jest w segmencie *text*, znanym również z tego względu pod nazwą *code*. Segment *text* położony jest najczęściej w **dolnym zakresie wirtualnej przestrzeni adresowej**, poniżej segmentu *.data*. Jego rozmiar jest stały. W większości przypadków segment *text* jest obszarem pamięci **tylko do odczytu**, aby zapobiec przed nieintencjonalną modyfikacją jego zawartości w trakcie działania programu. W segmencie *text* położone są również adresy funkcji zadeklarowanych w kodzie programu. Na rys. 2.9. przedstawiono typowy układ pamięci programu. Niektóre segmenty pamięci są dodatkowo rozdzielone przerwami o losowej szerokości. Ma to na celu utrudnienie manipulacji z zewnątrz w wykonywanym kodzie, dzięki wprowadzeniu elementu losowości w rozmieszczeniu segmentów pamięci [8].

Rozmiar segmentów *.bss*, *.data* oraz *text* skompilowanego programu może być sprawdzony z poziomu konsoli systemowej, za pomocą polecenia *size*. Wywołanie polecenia *size app.exe* może dać np. następujący rezultat:

```
1 C:\> size app.exe
2      text      data      bss      dec      hex      filename
3      3425      2344      448      6217      1849      app.exe
```



Rys. 2.9. Typowy układ pamięci programu

Programiści (w szczególności pracujący z mikrokontrolerami) często stosują pewne charakterystyczne wartości zmiennych do oznaczania skrajnych komórek zaalokowanego obszaru pamięci. Popularność w tym zastosowaniu zyskała konwencja **hexspeak**, polegająca na tworzeniu angielskich słów z wykorzystaniem liczb systemu heksadecymalnego, np. `0xDEADBEEF`, `0xCAFEFEFE`, `0xDEADC0DE` czy `0xFACEFEED`. Takie bariery na granicy obszarów pamięci (*ang. memory fences*) są łatwe do rozpoznania przez programistę w trakcie debugowania kodu, a ich naruszenie (nadpisanie) może oznaczać niepoprawne działanie programu.

2.3. Struktury

Tablice stanowią przykład zmiennych, które za pomocą pojedynczego identyfikatora (nazwy) umożliwiają odwołanie do wielu wartości jednego typu, położonych w ciągłym, liniowym obszarze pamięci komputera (mikrokontrolera). Zarówno język *C*, jak i *C++*, przewidują specjalny typ danych umożliwiający grupowanie **zmiennych różnych typów, ułożonych kolejno w ciągłym, liniowym obszarze pamięci**, pod wspólnym identyfikatorem. Zmienne takie nazywane są **strukturami**. Przykładem może być **struktura Student** przechowująca dane studenta – jego imię, nazwisko oraz numer indeksu (*struktury to pierwszy krok w kierunku klas, a więc i paradygmatu programowania obiektowego*). Deklarację i instancjonowanie struktury przedstawiono na listingu 4. Struktura **Student** została zadeklarowana z wykorzystaniem słowa kluczowego **struct**. Wewnątrz nawiasów klamrowych umieszczono kolejne składowe zmiennej strukturalnej (dwie tablice *char* oraz zmienną typu *unsigned int*), zwane **polami struktury**. Deklaracja struktury **zakończona jest średnikiem**. Wewnątrz funkcji *main()* utworzono instancję (egzemplarz) struktury **Student** o nazwie **wemifStudent**. Warto zwrócić uwagę, że typ instancjonowanej zmiennej zawiera słowo kluczowe **struct** (tu: *struct Student*). W języku *C++* zrezygnowano z tego zapisu – nazwa struktury jest pełnoprawnym typem zmiennej. Kolejne **polo** zmiennej **wemifStudent** są inicjalizowane z wykorzystaniem **listy inicja-**

lizacyjnej, zawierającej wartości kolejnych składowych *struktury* (umieszczone po przecinku wewnątrz nawiasów klamrowych). Do poszczególnych *pól struktury* można się później odwoływać za pomocą *operatora dostępu do składowych* – *operator.* oraz nazwy określonego *pola*.

```
1 #include <stdio.h>
2 // Struktura Student
3 struct Student {
4     char name[10];
5     char surname[20];
6     unsigned int index;
7 };
8
9 int main() {
10     // wemifStudent jest instancja struktury Student
11     // w C++ dozwolone: Student wemifStudent = ...
12     struct Student wemifStudent = { // Lista
13         inicjalizacyjna
14         "Adam",          // wartosc pola name
15         "Kowalski",      // wartosc pola surname
16         251439           // wartosc pola index
17     };
18     // Odwołanie do pol struktury
19     printf("Student's name: %s\n", wemifStudent.name);
20     printf("Student's surname: %s\n", wemifStudent.surname);
21     printf("Student's index: %d\n", wemifStudent.index);
22     return 0;
23 }
```

Listing 4. Deklaracja i instancjonowanie struktury

Aby uniknąć każdorazowego powtarzania słowa kluczowego ***struct*** można posłużyć się **mechanizmem aliasów typów**. Realizowane jest to za pośrednictwem słowa kluczowego ***typedef***. Przykład przedstawiono na listingu 5., gdzie zadeklarowana została struktura ***Car***, grupująca markę i model samochodu. Wykorzystując specyfikator ***typedef*** utworzono alias ***Car_t*** na typ ***struct Car***. Dzięki temu możliwe było utworzenie instancji **struktury** z użyciem krótszego identyfikatora typu.

```
1 // Struktura Car
2 struct Car {
3     char brand[15];
4     char model[20];
5 };
6
7 // Car_t jest aliasem typu struct Car
8 typedef struct Car Car_t;
9
10 // Instancja struktury Car
11 Car_t car = {
12     "Ford",
13     "Mondeo"
14 };
```

Listing 5. Aliasy typów w języku *C*

Język *C++* wprowadził **równoważny mechanizm aliasów typów** z wykorzystaniem słowa kluczowego ***using***. Przykład przedstawiono na listingu 6. [Uwaga: w związku z kompatybilnością wsteczną języka *C++* z językiem *C* możliwe jest definiowanie aliasów typów zarówno za pomocą słowa kluczowego ***typedef***, jak i ***using***.]

```
1 // Struktura Car
2 struct Car {
3     char brand[15];
4     char model[20];
5 };
6
7 // Car_t jest aliasem typu Car
8 using Car_t = Car;
9
10 // Instancja struktury Car
11 Car_t car = {
12     "Ford",
13     "Mondeo"
14 };
```

Listing 6. Aliasy typów w języku *C++*

Zarówno język *C*, jak i *C++*, umożliwiają tworzenie **nienazwanych typów strukturalnych**. Anonimowa struktura może mieć swoją (jedyną) instancję utworzoną w miejscu deklaracji jej typu. Ich zastosowanie w praktyce sprowadza się do struktur **zagnieżdżonych** w innych strukturach (*ang. nested structures*):

```
1 #include <stdio.h>
2 // Polaczenie deklaracji struktury Driver z
   deklaracja aliasu Driver_t
3 typedef struct Driver {
4     char name[10];
5     unsigned short age;
6     // Zagniezdzona anonimowa struktura
7     struct {
8         char brand[15];
9         char model[20];
10    } car; // car jest instancja anonimowej
   struktury i polem struktury Driver
11 } Driver_t; // Driver_t jest aliasem na typ struct
   Driver
12
13 int main() {
14     Driver_t driver = { // Lista inicjalizacyjna
15         "Andrzej",
16         54,
17         { // Lista inicjalizacyjna pola car
18             "Toyota",
19             "Corolla"
20         }
21     };
22     printf("Driver %s (age %d) has a red %s %s",
23         driver.name, driver.age,
24         driver.car.brand, driver.car.model
25     );
26     return 0;
27 }
```

Struktury mogą stanowić zarówno argumenty, jak i wartości zwracane z funkcji. Na listingu 7. przedstawiono program obliczający odległość na płaszczyźnie między dwoma punktami. Współrzędne punktów zapisywane są w strukturze *Point* i przekazywane jako argumenty funkcji *calculateDistance()*.

```
1 #include <math.h>
2 #include <stdio.h>
3
4 typedef struct Point {
5     double x;
6     double y;
7 } Point_t;
8
9 double calculateDistance(Point_t first, Point_t
    second) {
10     const double xDist = second.x - first.x;
11     const double yDist = second.y - first.y;
12     return sqrt(xDist * xDist + yDist * yDist);
13 }
14
15 int main() {
16     Point_t first = {1.0, 2.0};
17     Point_t second = {3.0, 5.0};
18     printf("Distance between two points is equal %f",
        calculateDistance(first, second));
19     return 0;
20 }
```

Listing 7. Struktura jako argument funkcji

Możliwe jest tworzenie wskaźników na *strukтуры*. Przykład przedstawiono na listingu 8. Zostały w nim uwzględnione **dwa równoważne sposoby odwołania do pól struktury za pośrednictwem wskaźnika**. Pierwszy, intuicyjny, przez **wywołanie operatora dereferencji na wskaźniku i zastosowanie operatora dostępu do składowej**: `(*beer).price`. Druga metoda polega na wykorzystaniu **operatora dostępu do składowej przez wskaźnik**: `beer->price`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct Beer {
5     float price;
6     float alcoholContent;
7 } Beer_t;
8
9 int main() {
10     Beer_t * beer = malloc(sizeof(Beer_t));
11     (*beer).price = 5.99;
12     (*beer).alcoholContent = 6.7;
13
14     printf("Beer has %.2f%% alcohol content and costs\n%.2f zł", beer->alcoholContent, beer->price);
15     free(beer);
16     return 0;
17 }
```

Listing 8. Wskaźnik na zmienną strukturalną

Na listingu 9. przedstawiono program wyświetlający rozmiar struktury i jej poszczególnych pól (w bajtach). Program skompilowany i uruchomiony na 64-bitowym systemie operacyjnym może dać następujący rezultat:

size of name equals 8 B
size of age equals 2 B
size of index equals 8 B
size of student equals 24 B

Jak widać, rozmiar zmiennej *student* nie jest równy prostej sumie rozmiarów poszczególnych zmiennych ($24 \neq 8 + 2 + 8$). Nierówność ta wynika z ułożenia *struktury* w pamięci komputera (mikroprocesora). Kolejne *pola struktury* ułożone są w sposób ciągły. Co więcej, każde *pole* wyrównane jest do rozmiaru *słowa maszynowego*, czyli podstawowego rozmiaru porcji danych, na którym operuje procesor. Rozmiar *słowa maszynowego* jest zależny od architektury procesora. Dla systemu 64-bitowego, rozmiar *słowa maszynowego* wynosi 8 B. Rozmiar zmiennej strukturalnej może być zatem równy wielokrotności *słowa maszynowego*. Ułożenie pól struktury z lisingu 9. w pamięci komputera przedstawiono na rys. 2.10.

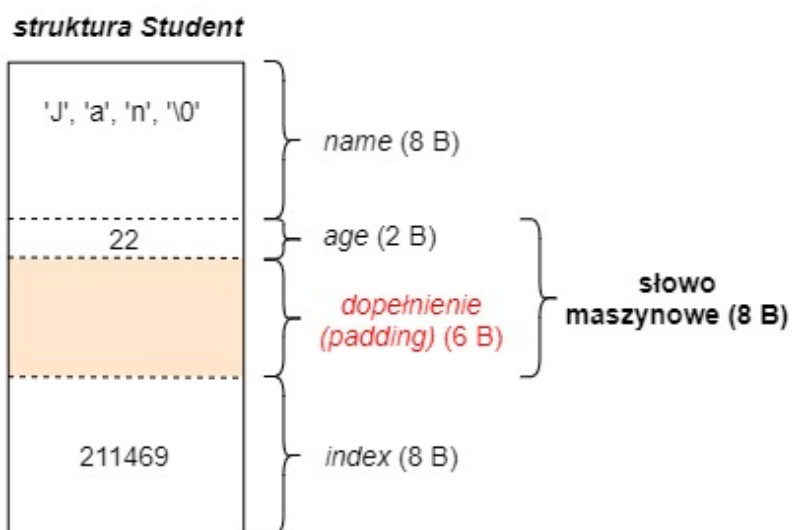
```
1 #include <stdio.h>
2
3 typedef struct Student {
4     char name[8];
5     unsigned short age;
6     unsigned long index;
7 } Student_t;
8
9 int main() {
10     Student_t student = {
11         "Jan",
12         22,
13         211469
14     };
```

```

15 printf("size of name equals %d B\n", sizeof(
    student.name));
16 printf("size of age equals %d B\n", sizeof(student
    .age));
17 printf("size of index equals %d B\n", sizeof(
    student.index));
18 printf("size of student equals %d B\n", sizeof(
    student));
19 return 0;
20 }

```

Listing 9. Rozmiar instancji struktury a rozmiary jej pól



Rys. 2.10. Ułożenie pól struktury w pamięci komputera

2.4. Operacje na pamięci

Biblioteka standardowa języka *C* (oraz *C++*) dostarcza gotowe funkcje, zadeklarowane w nagłówku ***string.h*** (*cstring*), realizujące podstawowe operacje na pamięci. Nagłówki funkcji przedstawiono na listingu 10.

```
1 void * memcpy (void * destination, const void *  
    source, size_t num);  
2 void * memmove (void * destination, const void *  
    source, size_t num);  
3 void * memset (void * ptr, int value, size_t num);  
4 int memcmp (const void * ptr1, const void * ptr2,  
    size_t num);  
5 void * memchr (const void * ptr, int value, size_t  
    num);
```

Listing 10. Nagłówki funkcji realizujących podstawowe operacje na pamięci

Funkcje *memcpy()* i *memmove()* realizują operację kopiowania danych o rozmiarze *num* bajtów z bloku pamięci wskazywanego przez *source* pod adres *destination*. Różnica między tymi funkcjami polega na tym, że *memcpy()* kopiuje dane bezpośrednio z *source* do *destination*, a *memmove()* pośrednio kopiuje dane do tymczasowego bufora. W związku z tym bloki *source* i *destination* nie mogą się nakładać w przypadku funkcji *memcpy()*, natomiast w przypadku *memmove()* dozwolone jest ich nakładanie się. Zastosowanie typu wskaźnikowego *void ** umożliwia generyczną obsługę różnych typów danych zaalokowanych pod adresami *source* i *destination*. Zarówno *memcpy()*, jak i *memmove()*, zwracają adres *destination* [4][5]. Na listingu 11. przedstawiono przykładową implementację funkcji *memcpy()* oraz przykład jej zastosowania. Wskaźniki *source* i *destination* są w pierwszej kolejności rzutowane na typ *unsigned char **, w celu kopiowania danych bajt po bajcie (typ *char* zajmuje zawsze rozmiar 1 B [patrz: Tabela 1, Ćw. 2]). Następnie, w pętli *for*, kopiowane są kolejne bajty bloku *source* bezpośrednio do bloku *destination*.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 void * memcpy (void * destination, const void *
    source, size_t num) {
5     unsigned char * pSrc = (unsigned char *)source;
6     unsigned char * pDst = (unsigned char *)
    destination;
7     for (unsigned int i = 0; i < num; ++i)
8         pDst[i] = pSrc[i];
9     return destination;
10 }
11
12 int main() {
13     char buffIn[] = "Ala ma kota";
14     const unsigned int buffLen = strlen(buffIn) + 1;
15     char buffOut[buffLen];
16     memcpy(buffOut, buffIn, buffLen);
17     // Wypisze "Ala ma kota"
18     printf("%s", buffOut);
19     return 0;
20 }

```

Listing 11. Przykładowa implementacja i zastosowanie funkcji *memcpy()*

Funkcja *memset()* inicjalizuje pierwsze *num* bajtów bloku pamięci wskazywanego przez *ptr* wartością *value*. Funkcja zwraca adres *ptr* [7]. Przykładowe użycie funkcji *memset()* przedstawiono na listingu 12.


```
1 char buffer[] = "Ala ma kota";
2 memset(buffer, '.', 3);
3 // Wypisze "... ma kota"
4 printf("%s", buffer);
```

Listing 12. Przykład użycia funkcji *memset()*

Funkcja *memcmp()* porównuje pierwsze *num* bajtów zawartości bloków pamięci wskazywanych przez *ptr1* i *ptr2*. Funkcja zwraca:

- 0 – jeżeli zawartość bloków pamięci jest identyczna
- wartość ujemną – jeżeli pierwszy różniący się bajt danych ma wartość większą w *ptr1* niż w *ptr2*
- wartość dodatnią – w przeciwnym wypadku [3].

Przykład użycia funkcji *memcmp()* przedstawiono na listingu 13.

```
1 char firstBuffer[] = "Hello, world!";
2 char secondBuffer[] = "Hello, world! Goodbye!";
3 if (memcmp(firstBuffer, secondBuffer, strlen(
4     firstBuffer)) == 0)
5     // Wypisze "Buffers' contents are equal"
6     printf("Buffers' contents are equal");
```

Listing 13. Przykład użycia funkcji *memcmp()*

Funkcja *memchr* przeszukuje pierwsze *num* bajtów bloku wskazywanego przez *ptr* pod kątem występowania wartości *value*. Funkcja zwraca adres, pod którym po raz pierwszy wystąpiła wartość *value* w zadanym bloku pamięci. Jeżeli zadana wartość nie została odnaleziona, funkcja zwraca *NULL* [2]. Przykład użycia funkcji *memchr()* przedstawiono na listingu 14.

```
1 char buffer[] = "Ala ma kota";
2 char * found = (char *)memchr(buffer, 'k', strlen(
    buffer));
3 if (found != NULL)
4     // Wypisze "Letter 'k' found at position 7 in
    buffer"
5     printf("Letter 'k' found at position %ld in
    buffer", found - buffer);
```

Listing 14. Przykład użycia funkcji *memchr()*

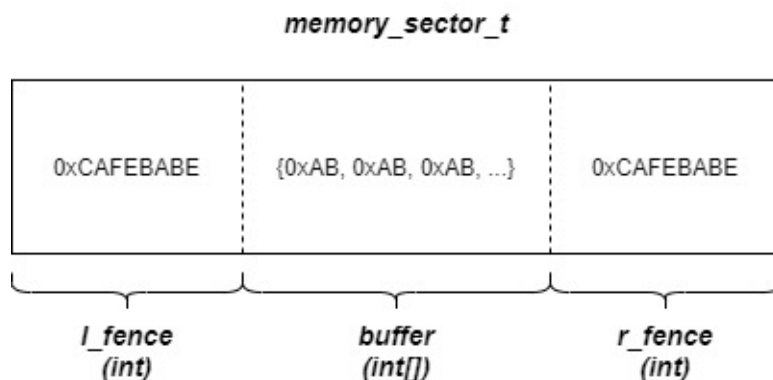
3. Program ćwiczenia

Zadanie 1. Plik *triangle.h* zawiera deklaracje struktury *Point_t*, przechowującej współrzędne punktu na płaszczyźnie, oraz funkcji *bool isRightTriangle(Point_t points[])*, która sprawdza czy trzy zadane punkty na płaszczyźnie tworzą trójkąt prostokątny. Punkty przekazywane są w postaci tablicy struktur *Point_t*. Napisz definicję funkcji *isRightTriangle()* i umieść ją w pliku *triangle.c*, a następnie przetestuj działanie zaimplementowanej funkcji. Współrzędne *x*, *y* punktów pobierz z klawiatury. Dla uproszczenia możesz przyjąć, że działanie funkcji jest *niezdefiniowane* w przypadku, gdy tablica punktów ma rozmiar różny niż 3.

Zadanie 2. Plik *memory_sector.h* zawiera deklarację struktury *memory_sector_t*, symulującej liniowy obszar pamięci. Struktura składa się z trzech pól: *l_fence* i *r_fence* – przeznaczonych na inicjalizację lewej i prawej bariery pamięci, oraz tablicy *buffer* o losowym rozmiarze. Typ struktury jest deklarowany przez wywołanie makra *INITIALIZE_MEMORY_SECTOR()* (z każdą kompilacją programu losowany jest inny rozmiar składowej *buffer*). Plik *memory_operations.h* zawiera deklaracje funkcji:

-
- *int * get_left_fence_address(memory_sector_t *)* – zwracającej adres lewej bariery pamięci (*l_fence*) instancji struktury *memory_sector_t*;
 - *int * get_right_fence_address(memory_sector_t *)* – zwracającej adres prawej bariery pamięci (*r_fence*) instancji struktury *memory_sector_t*;
 - *size_t get_memory_buffer_size(memory_sector_t *)* – zwracającej rozmiar bufora (*buffer*) instancji struktury *memory_sector_t*;
 - *void initialize_memory(memory_sector_t *)* – inicjalizującej instancję struktury *memory_sector_t*, tj. inicjalizującej *l_fence* i *r_fence* wartością *FENCE_INITIALIZER* i wypełniającą tablicę *buffer* wartościami *BUFFER_INITIALIZER*;
 - *address_status_t validate_address(memory_sector_t * memory_sector, const int * address)* – sprawdzającą czy zadany adres (*address*) leży wewnątrz obszaru pamięci zaalokowanego na *memory_sector*. Funkcja zwraca jedną z wartości typu wyliczeniowego *address_status_t*: *NO_ERROR* – jeżeli zadany adres należy do struktury, albo *ADDRESS_OUT_OF_RANGE* w przeciwnym wypadku;
 - *address_status_t read_memory(memory_sector_t *, const int * address, int * buffer)* – zapisującą zawartość komórki pamięci leżącej pod adresem *address* do zmiennej pod adresem *buffer*, jeżeli *address* należy do struktury *memory_sector_t*. Funkcja zwraca *NO_ERROR* – jeżeli odczyt danych się powiódł, albo wartość *ADDRESS_OUT_OF_RANGE* w przeciwnym wypadku;
 - *address_status_t write_memory(memory_sector_t *, int * address, int value)* – zapisującą wartość *value* pod adres *address*, jeżeli *address* należy do struktury *memory_sector_t*. Funkcja zwraca *NO_ERROR* – jeżeli zapis danych się powiódł, albo wartość *ADDRESS_OUT_OF_RANGE* w przeciwnym wypadku.

W funkcji *main()* (plik *main.cpp*) napisano zestaw testów funkcji z pliku nagłówkowego *memory_operations.h*, wykorzystując makro *assert*. Celem zadania jest napisanie definicji funkcji z nagłówka *memory_operations.h* (i umieszczenie ich w pliku źródłowym *memory_operations.cpp*) tak, aby po kompilacji i uruchomieniu programu wyświetlił się napis **All tests passed!**. Strukturę *memory_sector_t* schematycznie przedstawiono na rys. 3.1. [Uwaga: nie możesz modyfikować żadnego z plików: *memory_sector.h*, *memory_operations.h*, *main.cpp*]



Rys. 3.1. Schemat struktury *memory_sector_t*

Zadanie 3. Zadanie polega na rozszerzeniu zestawu funkcji **Zadania 2.** o funkcję *address_status_t copy_memory(memory_sector_t *, int * destination, int * source, size_t size)*, która kopiuje dane o rozmiarze *size* leżące pod adresem *source* pod adres *destination*, jeżeli adres *destination* należy do struktury *memory_sector_t*. Funkcja zwraca enumerator *NO_ERROR* – jeżeli zapis danych się powiódł, albo *ADDRESS_OUT_OF_RANGE* w przeciwnym wypadku. W ramach zadania:

- dodaj deklarację funkcji *copy_memory()* do pliku *memory_sector.h*;
- dodaj definicję funkcji do pliku *memory_sector.c* (skorzystaj z funkcji *memcpy()* lub *memmove()* biblioteki standardowej);
- rozszerz funkcję *main()* o testy nowo zaimplementowanej funkcji.

4. Dodatek

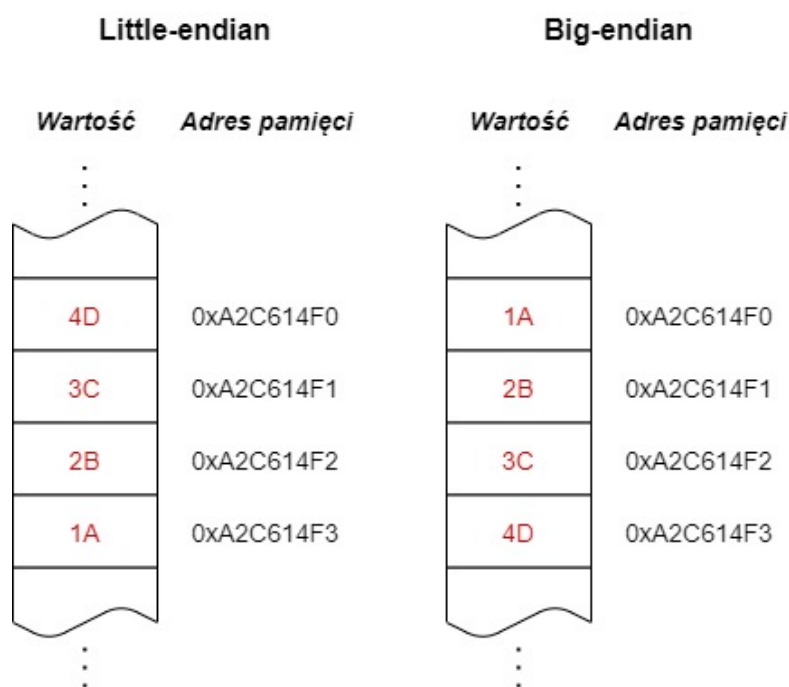
Zadanie (dla chętnych) Samodzielnie zweryfikuj układ pamięci programu na swoim komputerze. W tym celu utwórz po kilka egzemplarzy funkcji oraz zmiennych różnych rodzajów (alokowanych statycznie i dynamicznie, lokalnych, globalnych i statycznych) i na podstawie ich adresów sprawdź, w jakich obszarach przestrzeni adresowej położone są poszczególne segmenty pamięci oraz jaki jest kierunek rozrostu stosu i sterty. Adres funkcji w pamięci komputera można wyświetlić w sposób następujący:

```
1 #include <stdio.h>
2
3 void sayHello() {
4     printf("Hello, world!\n");
5 }
6
7 int main() {
8     sayHello();
9     printf("Function address: %p", sayHello);
10    return 0;
11 }
```

4.1. Kolejność bajtów

Istnieją dwie metody uporządkowania liczb składających się z wielu (co najmniej dwóch) bajtów podczas ich zapisu do pamięci komputera. Kolejność w jakiej bajty organizowane są w kolejnych komórkach pamięci (*ang. endianness*) nie jest ustalona przez standardy języków *C/C++* i zależy od konkretnego procesora, na którym wykonywany jest kod programu. W pierwszym podejściu najmniej znaczący bajt (*ang. least significant byte, LSB*) zapisywany jest jako pierwszy. Taka kolejność bajtów nosi nazwę **little-endian** (*LE*).

Odwrotny przypadek stanowi forma **big-endian** (BE), w której pierwszy zapisywany jest najbardziej znaczący bajt (*ang. most significant byte, MSB*). Obie metody organizacji bajtów w pamięci komputera dla liczby **0x1A2B3C4D** przedstawiono schematycznie na rys. 4.1. Analogiczna sytuacja, ale w przypadku kolejności bitów, ma miejsce dla transmisji danych za pomocą różnych protokołów komunikacyjnych.



Rys. 4.1. Organizacja bajtów liczby podczas zapisu do pamięci komputera

4.2. Organizacja danych w strukturach

Ponieważ kolejne składowe **struktur** są wyrównywane do rozmiaru **słowa maszynowego** procesora, ich odpowiednie ułożenie może prowadzić do optymalizacji rozmiaru pamięci wymaganej do utworzenia instancji **struktur**. Jest to szczególnie istotne w przypadku **struktur** o znacznych rozmiarach na mikrokontrolerach o ograniczonym rozmiarze pamięci. Na listingu 15. przedstawiono dwie deklaracje tej samej **struktury** z uwzględnieniem różnego ułożenia **pól struktury** (przykład dla architektury 64-

bitowej). Dla losowego (najgorszego) przypadku ułożenia *pól struktury* (*UnorganizedStruct*) na trzy składowe struktury przypada dodatkowe dopełnienie przydzielonej pamięci do rozmiaru równego wielokrotności rozmiaru *słowa maszynowego* (ang. *padding*). W przypadku zorganizowanego ułożenia *pól struktury* (*OrganizedStruct*) można uniknąć zbędnego dopełniania pamięci, przez co wypadkowy rozmiar *struktury* jest niemal dwukrotnie mniejszy (24 B a 40 B) – równy sumie algebraicznej rozmiarów poszczególnych *pól struktury*. Schematycznie zostało to zobrazowane na rys. 4.2.

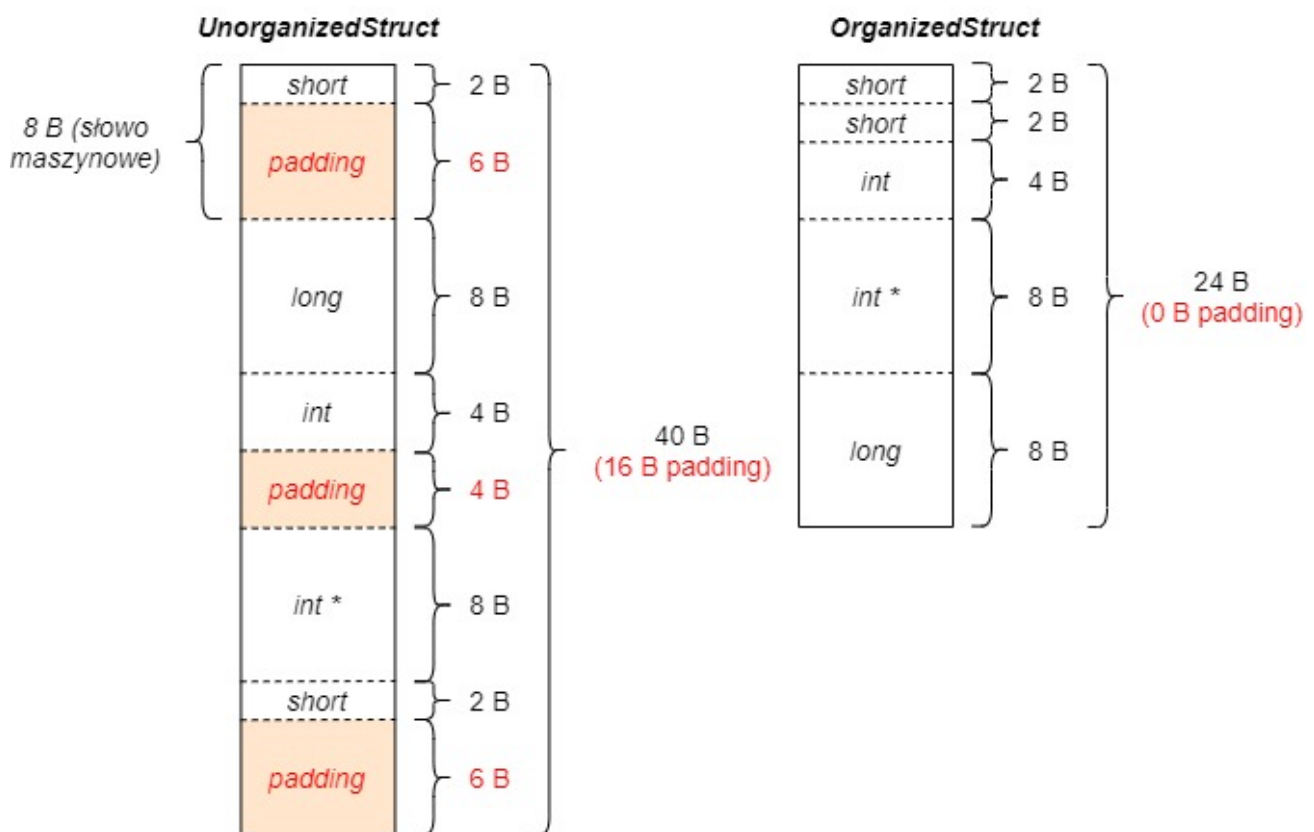
```
1 #include <iostream>
2
3 struct UnorganizedStruct {
4     short x; // 2 B (+ 6 B padding)
5     long z; // 8 B
6     int w; // 4 B (+ 4 B padding)
7     int * ptr; // 8 B
8     short y; // 2 B (+ 6 B padding)
9 };
10
11 struct OrganizedStruct {
12     short x; // 2 B
13     short y; // 2 B
14     int w; // 4 B
15     int * ptr; // 8 B
16     long z; // 8 B
17 };
18
19 int main() {
20     // 40 B
21     std::cout << "Size of unorganized struct: " <<
        sizeof(UnorganizedStruct) << " B" << std::endl;
```

```

22 // 24 B
23 std::cout << "Size of organized struct: " <<
    sizeof(OrganizedStruct) << " B" << std::endl;
24 return 0;
25 }

```

Listing 15. Ułożenie pól struktury a jej rozmiar



Rys. 4.2. Organizacja pól struktury a jej wynikowy rozmiar

Maksymalną długość dopełnienia rozmiaru pamięci dla poszczególnych pól struktury (ale również *unii* czy *klas*) można modyfikować z wykorzystaniem *dyrektywy preprocesora* **#pragma pack**. Ustawiana długość dopełnienia musi być potęgą liczby **2**. Dyrektywa **#pragma pack** nie ustawia dopełnienia **większego niż rozmiar słowa maszynowego**. Na listingu 16.

przedstawiono przykład zmniejszenia dopełnienia pól struktury z listingu 15. Ustawienie maksymalnej długości dopełnienia na 2 B poskutkowałoby osiągnięciem minimalnego możliwego rozmiaru struktury ***UnorganizedStruct*** (24 B - brak dopełnień dla poszczególnych pól struktury).

```
1 #include <iostream>
2
3 #pragma pack(4)
4 struct UnorganizedStruct {
5     short x; // 2 B (+ 2 B padding)
6     long z; // 8 B
7     int w; // 4 B
8     int * ptr; // 8 B
9     short y; // 2 B (+ 2 B padding)
10 };
11
12 int main() {
13     // 28 B
14     std::cout << "Size of unorganized packed struct: "
15         << sizeof(UnorganizedStruct) << " B" << std::
16         endl;
17     return 0;
18 }
```

Listing 16. Wykorzystanie dyrektywy preprocesora *#pragma once*

4.3. Unie

Innym, podobnym w składni do *struktur*, typem zmiennych są *unie*, deklarowane z wykorzystaniem słowa kluczowego *union*. *Unia* może odzwierciedlać różne typy danych, ale tylko jeden naraz. Wszystkie *pola unii* współdzielą jeden obszar pamięci, stąd rozmiar *unii* jest równy rozmiarowi największej ze składowych. Przykład deklaracji i zastosowania *unii* przedstawiono na listingu 17.

```
1 #include <stdio.h>
2 // Identyfikacja przez numer indeksu albo PESEL
3 typedef union StudentId {
4     unsigned int index;
5     unsigned long pesel;
6 } StudentId_t;
7
8 int main() {
9     // StudentId_t to alias na typ union StudentId
10    StudentId_t id;
11    // Unia pełni rolę zmiennej typu unsigned int
12    id.index = 253107;
13    printf("Index: %d\n", id.index);
14    // Unia pełni rolę zmiennej typu unsigned long
15    // Utrata danych zapisanych w id.index
16    id.pesel = 98171272137;
17    printf("PESEL: %ld\n", id.pesel);
18    // Równe sizeof(id.pesel)
19    printf("Union size: %d\n", sizeof(StudentId_t));
20    return 0;
21 }
```

Listing 17. Deklaracja i zastosowanie unii

Unie bywają stosowane w programowaniu mikrokontrolerów w celu **oszczędności pamięci** (gdy funkcja wykorzystuje kilka typów zmiennych, ale nigdy więcej niż jednego z nich jednocześnie). Przykład przedstawiono na listingu 18. Struktura **Variant** umożliwia przechowywanie zmiennej jednego z trzech typów: *int*, *long* albo *float*. Typ wyliczeniowy **MemberType** wskazuje jaki typ zmiennej jest aktualnie przechowywany w strukturze **Variant**. Dzięki takiemu rozwiązaniu rozmiar struktury jest równy sumie rozmiarów **MemberType** oraz **long** (największy rozmiar spośród *int*, *long*, *float*), powiększonej o wyrównanie do długości słowa maszynowego. Wynikowy rozmiar jest mniejszy niż suma rozmiarów *int*, *long* oraz *float*. Funkcje **initWithFloat()** oraz **initWithLong()** umożliwiają inicjalizację instancji struktury **Variant** odpowiednio wartością typu *float* albo *long*. [Uwaga: w języku C nie ma możliwości przeciążania funkcji, dlatego funkcje inicjalizujące nie mogą posiadać wspólnej nazwy] Funkcja **printVariant()** wyświetla wartość aktualnie przechowywaną w strukturze **Variant** w zależności od enumeratora **type**. Taką implementację można określić mianem *pseudo-polimorfizmu*, znanego z *programowania obiektowego* (np. w języku C++). Działanie funkcji może być różne w zależności od typu zmiennej aktualnie przechowywanej w strukturze **Variant**.

```
1 #include <stdio.h>
2
3 enum MemberType {
4     INT,
5     LONG,
6     FLOAT
7 };
8
9 // Przechowuje zmienna jednego z typow: int, long
   albo float
10 typedef struct Variant {
```

```
11     MemberType type;
12     union {
13         int intMember;
14         long longMember;
15         float floatMember;
16     };
17 } Variant_t;
18
19 // Inicjalizacja variant zmienna typu float
20 void initWithFloat(Variant_t * variant, float
    initializer) {
21     if (variant != NULL) {
22         variant->type = FLOAT;
23         variant->floatMember = initializer;
24     }
25 }
26
27 // Inicjalizacja variant zmienna typu long
28 void initWithLong(Variant_t * variant, long
    initializer) {
29     if (variant != NULL) {
30         variant->type = LONG;
31         variant->longMember = initializer;
32     }
33 }
34
35 void printVariant(Variant_t * variant) {
36     if (variant != NULL) {
37         switch(variant->type) {
38             case INT:
```

```

39         printf("Variant member value: %d\n",
variant->intMember);
40         break;
41     case LONG:
42         printf("Variant member value: %ld\n"
, variant->longMember);
43         break;
44     case FLOAT:
45         printf("Variant member value: %f\n",
variant->floatMember);
46         break;
47     default:
48         printf("Invalid member type\n");
49     }
50 }
51 }
52
53 int main() {
54     Variant_t variant;
55     initWithFloat(&variant, 4.5f);
56     printVariant(&variant);
57     initWithLong(&variant, 21);
58     printVariant(&variant);
59     return 0;
60 }

```

Listing 18. Pseudo-polimorfizm w języku *C* z wykorzystaniem unii

Inny przykład praktycznego wykorzystania *unii* w języku *C* przedstawiono na listingu 19. Ośmiobitowy rejestr ***Watchdog Timer Control Register*** (*WDTCR*) mikrokontrolera ATmega8535 służy do konfiguracji układu *watchdog*, którego zadaniem jest wykrywanie nieprawidłowego działania sys-

temu i automatyczne przywracanie mikrokontrolera do właściwego stanu. Zawartość rejestru przedstawiono na rys. 4.3.:

- bity 0...2 (**WDP0...WDP2**) – **Watchdog Timer Prescaler**; służą skalowaniu częstotliwości oscylatora układu *watchdog*;
- bit 3 (**WDE**) – **Watchdog Enable**; włączenie (1) albo wyłączenie (0) układu *watchdog*;
- bit 4 (**WDCE**) – **Watchdog Change Enable**; włączenie możliwości zmiany stanu (bitu **WDE**) układu *watchdog* (możliwość automatycznego wyłączenia układu *watchdog*);
- bity 5...7 – zarezerwowane (nieużywane).

Bit	7	6	5	4	3	2	1	0
WDTCR	-	-	-	WDCE	WDE	WDP2	WDP1	WDP0
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R/W

Rys. 4.3. Rejestr *Watchdog Timer Control Register* (**WDTCR**) mikrokontrolera ATmega8535 [1]

Unia **WDTCR** stanowi programową realizację rejestru **WDTCR** mikrokontrolera ATmega8535. Pole **reg** umożliwia jednoczesny zapis całego rejestru (bity 0...7), natomiast pola **WDP0**, **WDP1**, **WDP2**, **WDE**, **WDCE** i **reserved** umożliwiają zapis pojedynczych bitów. W danym momencie rejestr może przechowywać wartość zapisaną albo pod zmienną **reg** albo wewnątrz zagnieżdżonej struktury. Zapis : *n* obok pól struktury ogranicza wartość określonego pola do *n* bitów (tzw. *pola bitowe*), np. `uint8_t WDP0 : 1` ogranicza pole **WDP0** do wartości 0 albo 1 (1 bit).

```

1  #include <stdint.h>
2
3  typedef union {
4      struct {
5          // Pola bitowe
6          uint8_t WDP0 : 1;
7          uint8_t WDP1 : 1;
8          uint8_t WDP2 : 1;
9          uint8_t WDE : 1;
10         uint8_t WDCE : 1;
11         uint8_t reserved : 3;
12     };
13     // 8-bitowa zmienna calkowita
14     uint8_t reg;
15 } WDTCR;
16
17 int main() {
18     WDTCR watchdogRegister;
19     // WDP1 = 1, WDE = 1, WDCE = 1
20     // Zapis calego rejestru
21     watchdogRegister.reg = (1 << 1) | (1 << 3) | (1
22 << 4);
23     // Zapis pojedynczych bitow
24     watchdogRegister.WDP1 = 1;
25     watchdogRegister.WDE = 1;
26     watchdogRegister.WDCE = 1;
27     return 0;
28 }

```

Listing 19. Zapisywanie wartości rejestru z wykorzystaniem unii

Literatura

- [1] *ATmega8535(L) - Complete Datasheet*. URL: <https://ww1.microchip.com/downloads/en/DeviceDoc/doc2502.pdf>.
- [2] *memchr*. URL: <http://www.cplusplus.com/reference/cstring/memchr/>.
- [3] *memcmp*. URL: <http://www.cplusplus.com/reference/cstring/memcmp/>.
- [4] *memcpy*. URL: <http://www.cplusplus.com/reference/cstring/memcpy/>.
- [5] *memmove*. URL: <http://www.cplusplus.com/reference/cstring/memmove/>.
- [6] *Memory Layout of C Programs*. URL: <https://www.geeksforgeeks.org/memory-layout-of-c-program/>.
- [7] *memset*. URL: <http://www.cplusplus.com/reference/cstring/memset/>.
- [8] *Program w C++ i jego pamięć*. URL: <http://simplemesimpleit.pl/index.php/10-c/9-c-blok-obszary-rodzaje-pamieci-stos-sterata-static-bss-data-code-text>.
- [9] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. Prentice Hall Press, 2014.