

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНИХ НАУК ТА
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп'ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №3
з навчальної дисципліни «Алгоритми та структури даних»

Варіант 5

Виконав студент:

Омельніцький Андрій Миколайович

Група: КН-10236

Перевірив:

Старший викладач

Бульба Сергій Сергійович

Харків-2024

Зміст

1	Мета роботи	2
2	Завдання	2
3	Хід виконання	2
3.1	Підготовка до виконання завдання	3
3.2	Подання <i>short int</i>	5
3.3	Подання <i>double</i>	5
3.4	Подання <i>char</i>	5
3.5	Подання <i>Vec</i>	6
3.6	Подання <i>ParsedItem</i>	7
3.7	Подання <i>ParsedVec</i>	8
3.8	Парсинг <i>ParsedVec</i>	9
3.9	Відображення <i>ParsedVec</i>	11
3.10	Бітове подання <i>ParsedVec</i>	12
4	Висновки	15

1 Мета роботи

Отримати та закріпити знання про внутрішнє (комп'ютерне) подання числових типів даних у мовах програмування.

Теми для попередньої роботи:

- інформація та її подання в ЕОМ;
- адресація пам'яті;
- числові типи даних: цілі та дійсні;
- похибки подання дійсних чисел;
- статичні структури даних – масиви.

2 Завдання

Написати програму, яка виводить на екран внутрішнє (комп'ютерне) подання даних чотирьох типів. Типи даних обрати за табл. 3.1 згідно із своїм номером у журналі групи. Тип елементів масиву обрати за своїм розсудом.

За результатами роботи підготувати звіт з лабораторної роботи, де навести отримані результати та дати щодо них пояснення, зробити висновки.

№ З/П	integer	short int	long int	float	double	long double	char	[] <i>n</i> - вимірний
5		*			*		*	2

Рис. 1. Завдання за варіантом (5)

3 Хід виконання

Для виконання завдання було обрано мову Rust. Увесь код також додатково був розміщений в GitHub репозитарії: <https://github.com/blackgolyb/algos-labs>.

3.1 Підготовка до виконання завдання

Для початку напишемо утилітарні функції для відображення структур даних в бітовому вигляді та одразу використаємо їх для простих типів даних. Для цього напишемо:

- функцію для виводу байтів та бітів певного вказівника
- *trait ShowBytes*, який треба реалізувати для типів даних, які будуть підтримувати вивід в байтовому форматі
- макрос для зручної реалізації трейта *ShowBytes* для простих типів даних

Код програм алгоритма:

```

1 pub fn print_raw_bytes(byte: usize, size: usize) {
2     print_bytes(&byte as *const usize as *const u8, size);
3 }
4
5 pub fn print_byte(ptr: *const u8) {
6     let byte = unsafe { *ptr };
7     for j in 0..8 {
8         print!("{}", (byte >> (7 - j)) & 1);
9     }
10 }
11
12 pub fn print_bytes(ptr: *const u8, size: usize) {
13     for i in 0..(size - 1) {
14         print_byte(unsafe { ptr.add(i) });
15         print!("{}", "\n");
16     }
17
18     if size > 0 {
19         print_byte(unsafe { ptr.add(size - 1) });
20     }
21 }
22
23 pub fn show<T>(value: &T) {
24     let size = size_of:<T>();
25     let ptr = value as *const T as *const u8;
26     print_bytes(ptr, size);
27 }
28
29 pub trait ShowBytes {
30     fn show_bytes(&self);
31 }
32
33 #[macro_export]
34 macro_rules! impl_show_bytes {
35     ($impl_type:ident) => {
36         impl ShowBytes for $impl_type {
37             fn show_bytes(&self) {
38                 show:::<$impl_type>(&self);
39             }
40         }
41     };
42 }
43
44 impl_show_bytes!(i8);
45 impl_show_bytes!(i16);
46 impl_show_bytes!(i32);
47 impl_show_bytes!(i64);
48 impl_show_bytes!(i128);
49
50 impl_show_bytes!(u8);
51 impl_show_bytes!(u16);
52 impl_show_bytes!(u32);
53 impl_show_bytes!(u64);

```

```

54 impl_show_bytes!(u128);
55
56 impl_show_bytes!(f32);
57 impl_show_bytes!(f64);
58
59 impl_show_bytes!(usize);
60 impl_show_bytes!(bool);

```

Також реалізуємо функцію для зручного тестування працездатності виводу байтів

```

1 fn test_value<T: FromStr + Display + ShowBytes>(prompt: &str) {
2     let mut buffer = String::new();
3     print!("{}", prompt);
4     io::stdout().flush().unwrap();
5     io::stdin().read_line(&mut buffer).unwrap();
6     match buffer.trim().parse::<T>() {
7         Ok(value) => {
8             println!("Entered value: {}", value);
9             print!("Bin: ");
10            value.show_bytes();
11            println!();
12        }
13        Err(_) => eprintln!("Error parsing input: {}", buffer),
14    }
15 }

```

3.2 Подання *short int*

Через те, що в Rust трохи інші назви для типів, то *short int* називається *i16*. В минулому пункті ми вже реалізували логіку представлення байтів тому просто скористаємося нею

```
Enter short int value: 12
Entered value: 12
Bin: 00001100 00000000
```

Рис. 2. Приклад роботи для *short int*

3.3 Подання *double*

Через те, що в Rust інші назви для типів, то *double* називається *f64*. В минулому пункті ми вже реалізували логіку представлення байтів тому просто скористаємося нею

```
Enter double value: 12.34
Entered value: 12.34
Bin: 10101110 01000111 11100001 01111010 00010100 10101110 00101000 01000000
```

Рис. 3. Приклад роботи для *double*

3.4 Подання *char*

Напишемо реалізацію трейту *ShowBytes* для *char*

```
1 impl ShowBytes for char {
2     fn show_bytes(&self) {
3         let s = self.to_string();
4         let bytes = s.as_bytes();
5         for byte in bytes {
6             print_byte(byte);
7         }
8         println();
9     }
10 }
```

```
Enter char value: a
Entered value: a
Bin: 01100001
```

Рис. 4. Приклад роботи для *char*

3.5 Подання Vec

Для цього завдання було цікаво дослідити як в Rust працює стандартний *Vec*, який є динамічним масивом. А також було цікаво реалізувати можливість робити n-вимірний вектор.

Для початку давайте розглянемо як представляється *Vec* в пам'яті. Як можна побачити нижче, то *Vec* має такі поля: *len*, *cap*, *ptr*. Також додатково наведемо бінарне представлення усіх його полів, його адресу, розмір в байтах та бінарне представлення його вмісту.

```
addr: 0x7ffe520b0bd0 (11010000 00001011 00001011 01010010 11111110 01111111 00000000 00000000)
bytes size: 24
capacity: 4 (00000100 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
length: 3 (00000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
ptr: 10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000
bin: 00000100 00000000 00000000 00000000 00000000 00000000 00000000 10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000 00000011 00000000 0
0000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Inner: 00000001 00000000 00000000 00000000 | 00000010 00000000 00000000 00000000 | 00000011 00000000 00000000 00000000 | 00000000 00000000 00000000 00000000
Element 0: (10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000001 00000000 00000000 00000000
Element 1: (10000100 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000010 00000000 00000000 00000000
Element 2: (10001000 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000011 00000000 00000000 00000000
```

Рис. 5. Представлення *Vec* у пам'яті

Далі з цього малюнку можна побачити, що бінарне відображення працює правильно, бо бінарне представлення вмісту *Vec* співпадає з його бінарним представленням кожного елемента. Також можна побачити, що *Vec* виділив більше пам'яті ніж треба, що і відповідає значенню *capacity*. Не використана пам'ять позначена сірим кольором.

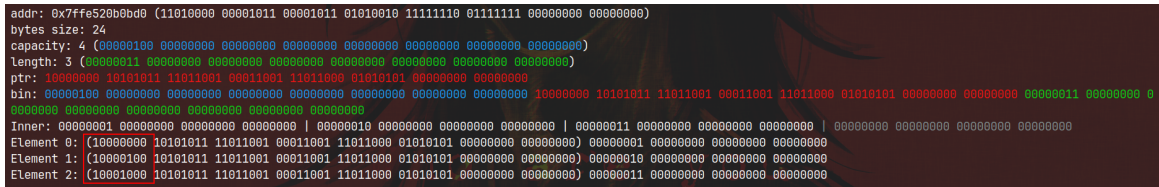
```
bytes size: 24
capacity: 4 (00000100 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
length: 3 (00000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
ptr: 10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000
bin: 00000100 00000000 00000000 00000000 00000000 00000000 00000000 10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000 00000011 00000000 0
0000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Inner: 00000001 00000000 00000000 00000000 | 00000010 00000000 00000000 00000000 | 00000011 00000000 00000000 00000000 | 00000000 00000000 00000000 00000000
Element 0: (10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000001 00000000 00000000 00000000
Element 1: (10000100 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000010 00000000 00000000 00000000
Element 2: (10001000 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000011 00000000 00000000 00000000
```

Рис. 6. Представлення *Vec* у пам'яті

Також можна побачити, що воно працює правильно, бо адреса першого елемента масиву співпадає з вказівником в *Vec*, а адреси елементів відрізняються один від одного на розмір елемента.

```
addr: 0x7ffe520b0bd0 (11010000 00001011 00001011 01010010 11111110 01111111 00000000 00000000)
bytes size: 24
capacity: 4 (00000100 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
length: 3 (00000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
ptr: 10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000
bin: 00000100 00000000 00000000 00000000 00000000 00000000 00000000 10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000 00000011 00000000 0
0000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Inner: 00000001 00000000 00000000 00000000 | 00000010 00000000 00000000 00000000 | 00000011 00000000 00000000 00000000 | 00000000 00000000 00000000 00000000
Element 0: (10000000 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000001 00000000 00000000 00000000
Element 1: (10000100 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000010 00000000 00000000 00000000
Element 2: (10001000 10101011 11011001 00011001 11011000 01010101 00000000 00000000) 00000011 00000000 00000000 00000000
```

Рис. 7. Представлення *Vec* у пам'яті

Рис. 8. Представлення *Vec* у пам'яті

3.6 Подання *ParsedItem*

Тепер ми можемо зробити таку структуру, яка буде багатовимірним масивом. Для цього можна зробити enum, який буде зберігати: або значення, або набір значень.

Напишемо реалізацію його та подивимося, як він виглядає з середини. *ParsedVec* розглянемо пізніше, тому поки можна вважати, що там звичайний *Vec*.

```

1  #[derive(Debug)]
2  pub enum ParsedItem<T> {
3      NestedVec(ParsedVec<T>),
4      Value(T),
5  }
6
7  fn test_item() {
8      let mut vec = Vec::new();
9      vec.push(1);
10     vec.push(2);
11     vec.push(3);
12     let p1: ParsedItem<u32> = ParsedItem::Value(10);
13     let p2: ParsedItem<u32> = ParsedItem::NestedVec(vec.into());
14     println!("ParsedItem<u32>: {u}{{}}u32:u{{}}\n", size_of:<ParsedItem<u32>>(), size_of:<u32>());
15
16     let size = size_of:<ParsedItem<u32>>();
17     print!("ParsedItem::Value:u");
18     print_bytes(&p1 as *const ParsedItem<u32> as *const u8, size);
19     println!("\n");
20     print!("ParsedItem::NestedVec:u");
21     print_bytes(&p2 as *const ParsedItem<u32> as *const u8, size);
22     println!("\n");
23 }

```

Як можна побачити, то розмір enum в Rust обирається за більшим типом який там є. А також додаткове місце, якщо воно потрібно на розміщення дискримінант за яким Rust визначає який тип зараз лежить в enum.

В нашому ж випадку Rust не потрібно додаткове місце в силу його внутрішніх оптимізацій. Тому наш enum буде займати всього 24 байти, що і відповідає розміру *Vec*, тому щоб подивитися що саме та як лежить в нашому enum ми можемо вивести бінарне представлення різних значень цього enum.

Тому для значення enum *Value* пропоную розглянути інвертований 0, щоб побачити де саме зберігається саме значення, а для значення enum *NestedVec* можемо взяти вектор з попереднього прикладу, щоб ми могли його легше перевірити


```

ParsedItem<u32>: 24  u32: 4
ParsedItem::Value: 00000000 00000000 00000000 00000000 00000000 00000000 10000000 11111111 11111111 11111111 00110101 01111111 00000000 00000000 00101
001 11001001 11010101 11001001 11011000 01010101 00000000 00000000
ParsedItem::NestedVec: 00000100 00000000 00000000 00000000 00000000 00000000 00000000 10100000 11001011 11010101 11001001 11011000 01010101 00000000 00000000 0
0000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000

```

Рис. 9. Представлення *Vec* у пам'яті

3.7 Подання *ParsedVec*

Тепер допишемо попередню структуру для того, щоб вона працювала правильно.

```

1  #[derive(Debug)]
2  pub enum ParsedItem<T> {
3      NestedVec(ParsedVec<T>),
4      Value(T),
5  }
6
7  #[derive(Debug)]
8  pub struct ParsedVec<T>(<Vec<ParsedItem<T>>>);
9
10 impl<T> From<Vec<T>> for ParsedVec<T> {
11     fn from(value: Vec<T>) -> Self {
12         let mut res = ParsedVec(Vec::new());
13         for v in value {
14             res.0.push(ParsedItem::Value(v))
15         }
16         res
17         // зроблено так щоб Rust не зміг тут оптимізувати копіювання вектора
18         // ParsedVec(value.into_iter().map(|v| ParsedItem::Value(v)).collect())
19     }
20 }

```

3.8 Парсинг *ParsedVec*

Тепер пропоную додати нашій структурі можливість парситися зі строки. Для цього напишемо парсер, який реалізує trait `FromStr` для нашої структури *ParsedVec*.

```

1  #[derive(Debug, PartialEq, Eq)]
2  pub struct ParseVecError;
3
4  impl<T: FromStr + Debug> FromStr for ParsedVec<T>
5  where
6      T::Err: std::fmt::Debug,
7  {
8      type Err = ParseVecError;
9
10     fn from_str(s: &str) -> Result<Self, Self::Err> {
11         let s = s
12             .trim()
13             .strip_prefix('(')
14             .and_then(|s| s.strip_suffix(''))
15             .and_then(|s| Some(s.trim()))
16             .ok_or(ParseVecError)?;
17
18         let mut items = Vec::new();
19
20         // for empty list/sublist []
21         if s.trim().is_empty() {
22             return Ok(ParsedVec(items));
23         }
24
25         let mut is_err = false;
26         for token in s.split(',') {
27             let parsed = token.trim().parse::<T>();
28             match parsed {
29                 Err(_) => {
30                     is_err = true;
31                     break;
32                 }
33                 Ok(val) => items.push(ParsedItem::Value(val)),
34             }
35         }
36
37         if !is_err {
38             return Ok(ParsedVec(items));
39         }
40
41         let mut items = Vec::new();
42         let mut depth = 0;
43         let mut start = 0;
44         let mut i = 0;
45         let mut is_comma = true;
46         let mut is_value = false;
47
48         while i < s.len() {
49             match &s[i..i] {
50                 "[" => {
51                     if depth == 0 {
52                         if !is_comma {
53                             return Err(ParseVecError);
54                         }
55                         is_comma = false;
56                         start = i;
57                     }
58                     depth += 1;
59                 }
60                 "]" => {
61                     if depth == 1 {
62                         let parsed = s[start..i].parse::<ParsedVec<T>>()?;
63                         items.push(ParsedItem::NestedVec(parsed));
64                     }
65                     depth -= 1;
66                 }
67                 "," if depth == 0 => {

```

```

68         if is_comma {
69             return Err(ParseVecError);
70         }
71         if is_value {
72             let parsed = s[start..i].trim().parse::<T>();
73             if parsed.is_err() {
74                 return Err(ParseVecError);
75             }
76             is_value = false;
77             items.push(ParsedItem::Value(parsed.unwrap()));
78         }
79         is_comma = true;
80     }
81     " " => (),
82     "\n" => (),
83     "\t" => (),
84     _ if depth == 0 && is_comma && !is_value => {
85         is_value = true;
86         is_comma = false;
87         start = i;
88     }
89     _ => (),
90 }
91 i += 1;
92 }
93
94 if is_value {
95     let parsed = s[start..i].trim().parse::<T>();
96     if parsed.is_err() {
97         return Err(ParseVecError);
98     }
99     items.push(ParsedItem::Value(parsed.unwrap()));
100 }
101
102 Ok(ParsedVec(items))
103 }
104 }

```

```

Enter array value: [1, [ 2 , 3], [ ], [], [4, [5 ,6]], [[7 , 8], 9] ,10]
Entered value: [1, [2, 3], [], [], [4, [5, 6]], [[7, 8], 9], 10]

```

Рис. 10. Приклад складного варіанту для парсингу

3.9 Відображення *ParsedVec*

Додамо можливість виводити нашу структуру *ParsedVec*. Для цього реалізуємо trait *Display* для нашої структури *ParsedVec*.

```

1  impl<T: Display> Display for ParsedItem<T> {
2      fn fmt(&self, f: &mut fmt::Formatter<'_,>) -> fmt::Result {
3          match self {
4              ParsedItem::Value(val) => write!(f, "{}", val),
5              ParsedItem::NestedVec(vec) => write!(f, "{}", vec),
6              _ => Ok(()),
7          }
8      }
9  }
10
11 impl<T: Display> Display for ParsedVec<T> {
12     fn fmt(&self, f: &mut fmt::Formatter<'_,>) -> fmt::Result {
13         let content: String = self
14             .0
15             .iter()
16             .map(|item| item.to_string())
17             .collect::<Vec<String>>()
18             .join(",\n");
19         write!(f, "[{}]", content)
20     }
21 }

```

3.10 Бітове подання *ParsedVec*

І на кінець реалізуємо наш trait `ShowBytes` для нашої структури *ParsedVec*. Для трохи кращої читаємості приберемо зайву інформацію.

```

1 fn print_layers(layers: &Vec<bool>) {
2     let s: String = layers
3         .iter()
4         .map(|l| match l {
5             true => "1",
6             false => "0",
7         })
8         .collect();
9     println!("{}", s);
10 }
11
12 fn show_bytes_vec<T: Display + ShowBytes>(vec: &Vec<T>, layers: &Vec<bool>) {
13     print_layers(layers);
14     println!("{}", {{}});
15     print_layers(layers);
16     println!("{}", |addr:u{:p}|, vec);
17     let addr = vec as *const Vec<T> as usize;
18     print_raw_bytes(addr, 8);
19     println!("{}", "");
20
21     print_layers(layers);
22     println!("{}", |capacity:u{:}|(x1b[34m", vec.capacity());
23     vec.capacity().show_bytes();
24     println!("{}", x1b[0m");
25
26     print_layers(layers);
27     println!("{}", |length:u{:}|(x1b[32m", vec.len());
28     vec.len().show_bytes();
29     println!("{}", x1b[0m");
30
31     print_layers(layers);
32     println!("{}", |ptr:u{:}|(x1b[31m");
33     (vec.as_ptr() as *const T).show_bytes();
34     println!("{}", x1b[0m");
35
36     print_layers(layers);
37     println!("{}", |{}|);
38 }
39
40 impl<T: ShowBytes + Display> ParsedItem<T> {
41     pub fn show_bytes_(&self, layers: &mut Vec<bool>) {
42         match self {
43             ParsedItem::Value(val) => {
44                 println!("{}", "Value:");
45                 print_layers(layers);
46                 println!("{}", {{}});
47                 print_layers(layers);
48                 println!("{}", |addr:u{:p}|, val);
49                 let addr = val as *const T as usize;
50                 print_raw_bytes(addr, 8);
51                 println!("{}", "");
52
53                 print_layers(layers);
54                 println!("{}", |bin:u{:}|);
55                 let ptr = val as *const T as usize as *const u8;
56                 let size = size_of::<ParsedItem<T>>();
57                 print_bytes(ptr, size);
58                 println!("{}", "");
59
60                 print_layers(layers);
61                 println!("{}", |value:u{:}|{val});
62
63                 print_layers(layers);
64                 println!("{}", |{}|);
65             }
66             ParsedItem::NestedVec(vec) => {
67                 vec.show_bytes_(layers);
68             }
69         }
70     }
71 }

```

```

69     }
70 }
71 }
72
73 impl<T: ShowBytes + Display> ShowBytes for ParsedItem<T> {
74     fn show_bytes(&self) {
75         self.show_bytes_(&mut Vec::new());
76     }
77 }
78
79 impl<T: ShowBytes + Display> ParsedVec<T> {
80     fn propagate_show_tree(&self, layers: &mut Vec<bool>) {
81         let len = self.0.len();
82         let mut inner_layers = layers.clone();
83         inner_layers.push(true);
84
85         for i in 0..(len - 1) {
86             print_layers(&inner_layers);
87             println!();
88             print_layers(layers);
89             print!("{}", " |-----");
90             self.0[i].show_bytes_(&mut inner_layers);
91         }
92
93         let level = inner_layers.len();
94         print_layers(&inner_layers);
95         inner_layers[level - 1] = false;
96         println!();
97         print_layers(layers);
98         print!("{}", " |-----");
99         self.0[len - 1].show_bytes_(&mut inner_layers);
100     }
101     pub fn show_bytes_(&self, layers: &mut Vec<bool>) {
102         if layers.is_empty() {
103             println!("{}", "Vec");
104         } else {
105             println!("{}", "NestedVec");
106         }
107
108         show_bytes_vec(&self.0, layers);
109
110         if self.0.is_empty() {
111             return;
112         }
113
114         self.propagate_show_tree(layers);
115     }
116 }
117
118 impl<T: ShowBytes + Display> ShowBytes for ParsedVec<T> {
119     fn show_bytes(&self) {
120         self.show_bytes_(&mut Vec::new());
121     }
122 }

```

Як приклад давайте наведемо двовимірний масив

```

Enter array value: [[1, 2], [ 3, 4] ]
Entered value: [[1, 2], [3, 4]]
Bin: Vec
{
  addr: 0x7fff11e1098 (10011000 00010000 00011110 00010001 11111111 01111111 00000000 00000000)
  capacity: 4 (00000100 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
  length: 2 (00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
  ptr: 00000000 10101100 00001101 00100000 11111101 01010101 00000000 00000000
}

NestedVec
{
  addr: 0x55fd20dac00 (00000000 10101100 00001101 00100000 11111101 01010101 00000000 00000000)
  capacity: 4 (00000100 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
  length: 2 (00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
  ptr: 10010000 10101011 00001101 00100000 11111101 01010101 00000000 00000000
}

Value:
{
  addr: 0x55fd20dab98 (10011000 10101011 00001101 00100000 11111101 01010101 00000000 00000000)
  bin: 00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000 11100010 10001010 00001101 00100000 11111101 01010101 00000000 00000000 00000000 00000000 00000000 00000000 10000000
  value: 1
}

Value:
{
  addr: 0x55fd20dabb0 (10110000 10101011 00001101 00100000 11111101 01010101 00000000 00000000)
  bin: 00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000 11100010 10001010 00001101 00100000 11111101 01010101 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  value: 2
}

NestedVec
{
  addr: 0x55fd20dac10 (00011000 10101100 00001101 00100000 11111101 01010101 00000000 00000000)
  capacity: 4 (00000100 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
  length: 2 (00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
  ptr: 01110000 10101100 00001101 00100000 11111101 01010101 00000000 00000000
}

Value:
{
  addr: 0x55fd20dac78 (01111000 10101100 00001101 00100000 11111101 01010101 00000000 00000000)
  bin: 00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000 11100010 10001010 00001101 00100000 11111101 01010101 00000000 00000000 00000000 00000000 00000000 00000000 10000000
  value: 3
}

Value:
{
  addr: 0x55fd20dac90 (10010000 10101100 00001101 00100000 11111101 01010101 00000000 00000000)
  bin: 00000100 00000000 00000000 00000000 00000000 00000000 00000000 00000000 11100010 10001010 00001101 00100000 11111101 01010101 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  value: 4
}

```

Рис. 11. Представлення *ParsedVec* у пам'яті

4 Висновки

В ході виконання лабораторної роботи було розглянуто подання різних типів даних в пам'яті, а також подання n-вимірного масиву в пам'яті.