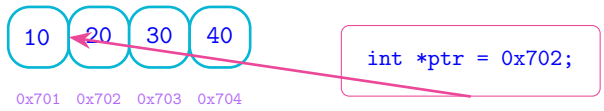
//ANSI C, C99, C11, C23

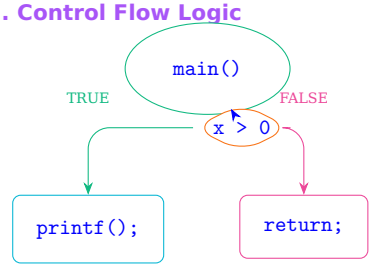
printf("პროგრამირების ენა C");

საფუძვლები, სტრუქტურები და მეხსიერება

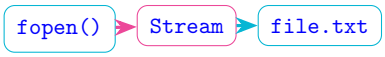
1. Pointers & Memory Layout



2. Control Flow Logic

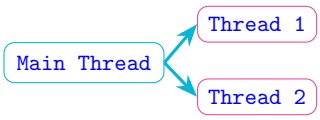


3. File I/O & Streams



4. Compilation Pipeline

5. High Performance C




```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    // C Course Interactive Guide
    printf("Welcome to C Programming!\n");
    return 0;
}
```

ლევან შოშიაშვილი

პროგრამირების ენა C

თბილისი
2016,2026

სარჩევი

განახლებული გამოცემის წინასიტყვაობა IX

წინასიტყვაობა XI

შესავალი XIII

C* და *C++ . . . . . **xiii**

***C++* უპირატესობანი** . . . . . **xiii**

ობიექტზე ორიენტირებულობა და მონაცემებზე ორიენტირებულობა . . . . . **xiv**

პროგრამული კოდი XV

დავალება XIX

1 C ენის ლექსიკური ელემენტები, ცვლადები და ფუნქციები 1

1.1 ლექსიკური ელემენტები, ცვლადები და ტიპები **2**

1.1.1 ცვლადის ატრიბუტები და დასახელების წესები 2

1.1.2 ბაზური მონაცემთა ტიპები და დიაპაზონები 2

1.2 ცვლადების დეკლარაცია და ინიციალიზაცია **3**

1.3 ფორმატირებული გამოტანა: ფუნქცია printf **3**

1.3.1 სპეციალური მმართველი სიმბოლოები (Escape Sequences) 3

1.3.2 მაგალითი: მმართველი სიმბოლოების გამოყენება 4

1.4 printf ფუნქციის ფორმატირება **4**

1.4.1 განსხვავება %d -სა და %i -ს შორის 5

1.4.2 სიგანე და სიზუსტე (Width & Precision) 5

1.4.3 დროშები (Flags) 5

1.5 ნამდვილი რიცხვების ფორმატები: %f, %e და %g **8**

1.6 sizeof ოპერატორი **9**

1.6.1 sizeof — ოპერატორი თუ ფუნქცია? 10

1.7 მათემატიკური ოპერაციები და არითმეტიკა **10**

1.7.1 ზრდის (Increment) და შემცირების (Decrement) ოპერატორები 11

1.8 ფუნქცია scanf — მონაცემების წაკითხვა კლავიატურიდან 11
1.8.1 მისამართის აღების ოპერატორი (&) 11
1.9 scanf-ის ფორმატის სტრინგში სიმბოლოების მართვა 12
1.10 ფორმატის სპეციფიკატორები და მოდიფიკატორები 13
1.11 %c სპეციფიკატორის თავისებურება scanf()-ში 14
1.12 C99 და თანამედროვე სტანდარტების სპეციფიკატორები 15
1.13 მცოცავი ტიპის ცვლადები თექვსმეტობით ფორმატში 15
1.13.1 printf-ში გამოყენება 16
1.13.2 scanf-ში გამოყენება 16
1.13.3 რატომ არის ათობითი (%f) პრობლემური? 17
1.13.4 თექვსმეტობითი (%a) ჩანაწერის უპირატესობა. 17

2 გადაწყვეტილების მიღება და ციკლები 21
2.1 შედარების (პირობითი) ოპერატორები (<, >, <=, >=, ==, !=) 21
2.1.1 ლოგიკური ოპერატორები (&&, ||, !) 22
2.2 'if' და 'if ... else' ბრძანება 22
2.3 ბრძანება switch ... case 27
2.4 ციკლები 32
2.4.1 for ციკლი. 32
2.4.2 while ციკლი 35
2.4.3 ბრძანება break და continue 38
2.5 ჩამონათვალი (დანომრილი) ტიპი enum 40

3 მნიშვნელობა, მისამართი, მითითება, ფუნქციები 45
3.1 მნიშვნელობა და მისამართი 45
3.2 მიმთითებელი 46
3.2.1 NULL მიმთითებელი 47
3.3 პრეპროცესორის დირექტივები 48
3.3.1 დირექტივა #define 49
3.3.2 არგუმენტიანი #define მაკროები და მათი საფრთხეები 50
3.4 ფუნქციები 51
3.4.1 const მოდიფიკატორი მიმთითებლთან. 54
3.5 მონაცემების გადაცემა მიმთითებლით 55
3.6 მიმთითებელი ფუნქციაზე (Function Pointer). 56
3.6.1 ფუნქციის მიმთითებლის გამოცხადების სინტაქსი 56
3.6.2 ფუნქციაში ფუნქციის მიმთითებლის გადაცემა და გამოძახება. 57
3.7 დეკლარაცია/იმპლემენტაცია სხვადასხვა ფაილებში. 58

3.8 extern "C" **62**
3.9 C პროექტის შექმნა **63**

4 C ენის სტანდარტები და იმპლემენტაცია 65
4.1 ახალი საკვანძო სიტყვები (Keywords) C99-დან C23-მდე **69**
4.1.1 C99 საკვანძო სიტყვები 69
4.1.2 C11 საკვანძო სიტყვები 69
4.1.3 C23 საკვანძო სიტყვები 69
4.2 `math.h` და `complex.h` სათავო ფაილების გაფართოებები **70**
4.2.1 Floating-point კლასიფიკაცია და მაკროები 70
4.2.2 Type-Generic Math (<tgmath.h>) 70
4.3 მაგალითი: კომპლექსური რიცხვი **70**
4.4 Compile-Time კონსტანტები C23-ში: constexpr, const და #define **72**
4.4.1 C23 სტანდარტი და შემოღების მიზეზები 72
4.4.2 #define მაკროსთან შედარება 72
4.4.3 const საკვანძო სიტყვასთან შედარება 74
4.4.4 `constexpr` C და C++-ში: ყოფაქცევა და განსხვავებები 75
4.4.5 constexpr-ის გამოყენება C23-ში 76
4.5 C11 `_Generic`, ფუნქციების გადატვირთვა (Function Overloading) C-ში და მაკროები **77**
4.5.1 `_Generic`-ის და `static inline` ფუნქციების ტანდემი 78
4.5.2 მაკროს ისტორიული პრობლემების გადაჭრა 78
4.5.3 C და C++ Overloading-ის შედარება 79

5 ფაილში ჩაწერა და წაკითხვა 81
5.1 ფაილის ცნება და მოდელი C-ში **81**
5.2 ტექსტური და ბინარული რეჟიმი **82**
5.2.1 ტექსტურ და ბინარულ ფორმატებს შორის განსხვავება 82
5.2.2 სტანდარტული ნაკადები (Streams) 83
5.3 ფაილის გახსნა და დახურვა: `fopen` და `fclose` **83**
5.4 ასო-ასო ჩაწერა და წაკითხვა: `fgetc()` და `fputc()` **84**
5.4.1 მაგალითი 1: სიმბოლოების წაკითხვა და ეკრანზე გამოტანა 84
5.4.2 ფაილის დასასრულის შემოწმება 85
5.5 სტრიქონების ჩაწერა და წაკითხვა: `fgets()` და `fputs()` **86**
5.5.1 მაგალითი 2: ფაილის წაკითხვა სტრიქონ-სტრიქონ 86
5.6 ფორმატირებული შეტანა/გამოტანა: `fprintf()` და `fscanf()` **87**
5.6.1 მაგალითი 3: ბაზური ტიპების ჩაწერა ფაილში (`fprintf`) 87

5.6.2 მაგალითი 4: ბაზური ტიპების წაკითხვა ფაილიდან (`fscanf`) 88
5.7 ბინარული შეტანა/გამოტანა: `fwrite()` და `fread()` 89
5.8 ფაილში პოზიციონირება: `fseek()`, `ftell()` და `rewind()` 89
5.8.1 მაგალითი 6: ფაილის ზომის დადგენა ბაიტებში 90
5.9 შეჯამება: ოპერაციები ბაზურ ტიპებზე 90
5.10 სავარჯიშოები 91

6 მასივი 93
6.1 ერთგანზომილებიანი მასივი 94
6.1.1 ერთგანზომილებიანი მასივის ინიციალიზაცია 94
6.1.2 დანიშნული ინიციალიზატორები (Designated Initializers) და C23-ის სიახლეები 96
6.1.3 მასივის ელემენტებზე მიმართვა 97
6.1.4 მიმთითებელი და მასივი 99
6.1.5 მაგალითი: მთვლელების მასივი 102
6.1.6 მაგალითი: მთელი მონაცემების ჰისტოგრამა 103
6.1.7 მაგალითი: შემთხვევითი რიცხვების გენერატორი 104
6.1.8 მაგალითი: ფიბონაჩის რიცხვები 105
6.1.9 მაგალითი: რიცხვი სხვადასხვა სისტემაში 106
6.2 სიმბოლოების მასივი (სტრიქონები / Strings) 107
6.3 სიმბოლოების მასივი `'0'`-ის გარეშე 110
6.3.1 სინტაქსის დეტალური განმარტება 110
6.4 სტრიქონის წაკითხვა ფაილიდან 112
6.5 მასივის გადაცემა ფუნქციაში 112
6.5.1 მაგალითი: ერთგანზომილებიანი მასივის გადაცემა 113
6.5.2 მაგალითი: მასივის დალაგება 116
6.6 მრავალგანზომილებიანი მასივები 117
6.6.1 მრავალგანზომილებიანი მასივის ინიციალიზაცია 118
6.6.2 მაგალითი: ორგანზომილებიანი მასივების შეკრება 119
6.6.3 2D მასივის გადაცემა ფუნქციაში 120
6.6.4 C99-მდე არსებული ალტერნატიული მეთოდები 2D მასივის გადასაცემად 122
6.7 1D მასივის ჩაწერა/წაკითხვა ფაილიდან 125
6.8 სტატიკური (სტეკ) 3D მასივი 126
6.9 ცვლადი ზომის მასივები (VLA) C99/C11 სტანდარტებში 128
6.9.1 VLA-ის ისტორია და C11 ცვლილებები 128
6.9.2 რატომ გახდა VLA არასავალდებულო? 129

6.9.3 VLA მხარდაჭერის შემოწმება კოდში 129
6.10 ბრძანებათა სტრიქონის არგუმენტები: `argc` და `argv` 129
6.10.1 პრაქტიკული მაგალითები: ბრძანებათა სტრიქონის არგუმენტები 130
6.11 სავარჯიშოები 133
6.12 2D და 3D სტეკ მასივების ჩაწერა და წაკითხვა 134
6.13 მასივის ფორმატირებული ჩაწერა/წაკითხვა 136
6.13.1 კოდის დეტალური განმარტება ხაზების მიხედვით 139
6.13.2 წაკითხვა სიმბოლომდე — `%99[^=]` 140
6.13.3 `\r\n` და კროსპლატფორმულობა `fscanf()`-ში 141
6.14 სავარჯიშოები 142

7 შედგენილი ტიპები: struct და union 145
7.1 სტრუქტურები – struct 145
7.2 სტრუქტურის ცვლადების ინიციალიზაცია 148
7.3 მაგალითი: სტრუქტურების მასივის დალაგება 149
7.4 ჩადგმული სტრუქტურები (Nested Structs) 151
7.5 სტრუქტურები და ფუნქციები. სტრუქტურის მიმთითებელი 152
7.6 typedef და struct 153
7.6.1 სტრუქტურის ფორმატირებული ბეჭდვა 153
7.7 ბიტური ველები (Bit-fields) სტრუქტურაში 154
7.8 სტრუქტურები და დინამიკური მეხსიერება 155
7.9 გაერთიანება – union 155
7.10 სპეციალური მაგალითები: Vector3D და RGB/Color Union 157
7.10.1 მაგალითი : Vector3D (სახელით და მასივით წვდომა) 157
7.10.2 მაგალითი : RGBA Color (32-ბიტიანი Hex რიცხვი vs 8-ბიტიანი არხები) . 158
7.11 სტრუქტურების და იუნიონების ფაილში ჩაწერა და წაკითხვა 159
7.11.1 ტექსტურ ფორმატში ჩაწერა და წაკითხვა (`fprintf` / `fscanf`) 159
7.11.2 ბინარულ ფორმატში ჩაწერა და წაკითხვა (`fwrite` / `fread`) 160

8 ცვლადები და მეხსიერება 163
8.1 მეხსიერების განაწილება C ენაში:სეგმენტები და მათი მახასიათებლები . 163
8.1.1 Text / Code Segment (კოდის სეგმენტი) 163
8.1.2 Data Segment (ინიციალიზებული მონაცემები) 164
8.1.3 BSS Segment (არაინიციალიზებული მონაცემები) 164
8.1.4 Heap (გროვა) 164
8.1.5 Stack (სტეკი) 164
8.2 ავტომატური ცვლადები და ხილვადობის არე 166

8.3 გლობალური და ლოკალური ცვლადები 167
8.4 გლობალური ცვლადების ხილვადობის არე და `static` ცვლადები 168
8.4.1 მრავალფაილიანი პროექტის მაგალითი 168
8.5 სტატიკური (`static`) ფუნქციები C ენაში. 170
8.6 `inline` და `static inline`-ის განსხვავება C და C++ ენებში. 173
8.7 ცვლადის კვალიფიკატორი `volatile` 178
8.8 ტიპის გარდაქმნა — კასტირება (Type Casting) 178
8.9 `void*` მიმთითებელი (Generic Pointer) 178
8.10 მეხსიერების დინამიური მართვა 179
8.10.1 `realloc()` ფუნქციის პრაქტიკული მაგალითი 180
8.11 დინამიური 2D და 3D მასივები (Flat Array Approach) 181
8.12 3D მასივის სტრუქტურა და ინკაფსულაცია 181
8.13 `goto` ოპერატორი და ნიშნულები (Labels). 182

9 ბიტური ოპერაციები და ბიტური ველები 185
9.1 ბიტური ლოგიკური ოპერაციები (AND, OR, XOR, NOT) 186
9.2 მარცხნივ და მარჯვნივ წანაცვლების ოპერაციები 187
9.3 ბიტური ოპერაციების პრიორიტეტი (Precedence) 188
9.4 ბიტური ოპერაციების მექანიზმი ორობით (0 და 1) ენაზე 189
9.4.1 ბიტური ნიღაბი 189
9.5 ბიტური ნიღბები (Bit Masks): 8, 16 და 32-ბიტიანი სისტემები 191
9.5.1 8, 16 და 32-ბიტიანი ნიღბების განსაზღვრა 191
9.5.2 ბიტური ოპერაციების მაგალითი 192
9.5.3 უნივერსალური ბიტური მაკროსები (8, 16, 32, 64 ბიტისთვის) 193
9.6 პრაქტიკული გამოყენება: აპარატურული რეგისტრებისა და LED-ების მართვა 194
9.6.1 ლაბორატორიული მაგალითები: LED-ების მართვის სცენარები 194
9.7 ბიტური ველები (Bit Fields) 197
9.7.1 8, 16 და 32-ბიტიანი სტრუქტურის მაგალითები 197
9.7.2 უსახელო ველები (Unnamed Bit Fields) და მეხსიერების გასწორება . . . 198
9.7.3 K&R კრიტიკული გაფრთხილება: `signed` vs `unsigned` ბიტური ველები . . 199
9.7.4 Unions + Bit Fields: აპარატურული რეგისტრების იდეალური მოდელი . 200
9.7.5 ბიტური ველების შეზღუდვები და არაპორტაბელურობა 202

10 მონაცემების დამუშავება და მაღალი წარმადობის გამოთვლები 203
10.1 FISO სენსორით გაზომილი მონაცემების წაკითხვა. 203
10.2 მასივების სტრუქტურა თუ სტრუქტურების მასივი (AoS vs SoA) 209

10.3 Fortran vs C/C++: წარმადობა, IEEE 754 და მეხსიერების არქიტექტურა 210
10.3.1 `restrict` საკვანძო სიტყვა და Pointer Aliasing 210
10.3.2 IEEE 754 სტანდარტი და კომპლექსური გამრავლება. 211
10.3.3 მათემატიკური მატრიცა და მეხსიერების ლეიოუტი: Row-Major vs Column-Major 211
10.3.4 წარმადობა (Performance) და მატრიცების გამრავლება 213
10.3.5 მაგალითი: OpenGL (Column-Major) მატრიცის ჩალაგება, ინიციალიზაცია და დაბეჭდვა C-ში 215
10.4 Fortran და C კოდები, ჩალაგება და წარმადობა. 217
10.5 GFLOPS-ის გამოთვლის ფორმულის დეტალური ანალიზი 225
10.6 ახალი მათემატიკური ფუნქციები 226
10.7 C დროის ფუნქციები vs `C++ std::chrono` 226

განახლებული გამოცემის წინასიტყვაობა

— თანამედროვე სტანდარტების დაფარვა: კურსი არ შემოიფარგლება მხოლოდ ANSI C-ით და მოიცავს C99, C11 და C23 სტანდარტების სიახლეებს, როგორიცაა size_t, uint64_t, designated initializers და tgmath.h.

— ჩაშენებული სისტემების (Embedded) აქცენტი: ბიტურ ოპერაციებთან, GPIO რეგისტრების მართვასთან, LED-ების კონტროლთან და ბიტურ ველებთან (Bit Fields) მუშაობა იდეალურად შეესაბამება ინჟინერიის სტუდენტების საჭიროებებს.

— დაემატა ახალი თავი მაღალი წარმადობის გამოთვლები (HPC): AoS vs SoA არქიტექტურა, Fortran vs C შედარება, restrict საკვანძო სიტყვა, მეხსიერების განლაგება (Row-Major) და GFLOPS-ის ანალიზი.

— მეხსიერების მართვა: დინამიური 2D/3D მასივებისა და goto-ს უსაფრთხო გამოყენების სცენარების (მაგ. რესურსების გასუფთავება) განხილვა სტუდენტებს უყალიბებს დაბალი დონის პროგრამირების სწორ ჩვევებს.

— განახლდა კოდის ფორმატირება და ვიზუალიზაცია, რაც აადვილებს ციფრული დოკუმენტის კითხვადობას.

ლევან შოშიაშვილი
28 სექ. 2026წ.

წინასიტყვაობა

C პროგრამირების ენის ცოდნა აუცილებელია ელექტრული და ელექტრონული ინჟინერიის სპეციალობის სტუდენტებისათვის, რომლებიც სწავლობენ მიკროკონტროლიორების და ჩაშენებული სისტემების პროგრამირებას.

სამწუხაროდ, ქართველი სტუდენტები არ არიან განებივრებული სასწავლო-სამეცნიერო ლიტერატურით ქართულ ენაზე. ეს განსაკუთრებით ეხება პროგრამირების თანამედროვე ენებს.

თუკი *C++* ენისათვის შეიძლება მოიძებნოს რამდენიმე სახელმძღვანელო თუ ლექციათა კურსი ქართულ ენაზე, *C* პროგრამირების ენისათვის ასეთი რამ არ არსებობს.

წინამდებარე დოკუმენტი წარმოადგენს ლექციათა კურსს წაკითხულს ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტში ზუსტ და საბუნებისმეტყველო ფაკულტეტზე ელექტრული და ელექტრონული ინჟინერიის დეპარტამენტის სტუდენტებისათვის.

უნდა აღინიშნოს, რომ ესაა ლექციათა კურსი და არა წიგნი *C* ენაში. რაც ნიშნავს, რომ საკითხები რაც განხილვის მიღმა არის დატოვებული განიხილება პრაქტიკულ მეცადინეობებზე. პროგრამირების ენის სწავლება განუყოფელია პროგრამული კოდის წერის პროცესთან.

ამ სალექციო კურსში მოყვანილი მასალა საკმარისია დამწყებთათვის, რომ დამოუკიდებლად გაეცნონ *C* ენაში არსებულ წარმოდგენებს და პროგრამირების ტექნიკას, დამოუკიდებლად ამოხსნან სალექციო მასალაში მოყვანილი ამოცანები[1].

მოცემული კურსის შესწავლა საკმარისია, რომ შემდგომში გაუადვილდეთ „მიკროკონტროლიორების დაპროგრამების“ და „მოდელირება და ვიზუალიზაცია“ კურსების შესწავლა და ჰქონდეთ მყარი საფუძველი დამოუკიდებლად გაიღრმავონ ცოდნა მოცემულ სფეროში.

აღსანიშნავია, რომ ქართული ტერმინოლოგია მაღალ ტექნოლოგიურ დარგებში ნაკნულია. ზოგიერთი ტერმინის ქართული შესატყვისი ან არ გამოიყენება, ან საერთოდ არ არსებობს. ჩვენ ვცდილობდით ტერმინოლოგია ყოფილიყო როგორც ქართულად გამართული, ასევე გასაგები[2].

ლევან შოშიაშვილი
2016წ.

[1] დოკუმენტის განახლებული ვერსია, პროგრამული კოდები და კურსის ვიდეომასალა შეგიძლიათ იხილოთ საიტზე `http://lshoshia.science.tsu.ge`. მოსაზრებები, შენიშვნები შეგიძლიათ გაგვიზიაროთ `shoshia at hotmail dot com`

[2] დაიხ, სამწუხაროდ ზოგიერთ ავტორთან გვხვდება ისეთი ქართული ტერმინებიც რომელთა მნიშვნელობაც გაუგებარია.

შესავალი

C და *C++*

Unix სისტემის შექმნასთან ერთად აუცილებელი შეიქმნა ახალი პროგრამული ენის შექმნა, რომელიც სამუშაოდ და ასათვისებლად უნდა ყოფილიყო გაცილებით იოლი ვიდრე ასემბლერი. ნაკლებად უნდა ყოფილიყო დამოკიდებული მანქანის მოწყობილობებზე. ამ ენაზე შესაძლებელი უნდა ყოფილიყო სისტემური პროგრამირება. დაწერილი პროგრამული კოდი უნდა ყოფილიყო გადატანადი ერთი სისტემიდან მეორეზე. ასე შეიქმნა პროგრამირების ენა *C*.

C++ განვითარდა 80-იან წლებში AT&T ბელლ ლაბორატორიაში. თავდაპირველად. საჭირო იყო პრეკომპილატორი, რომელსაც *C++* კოდი გადაჰყავდა *C* კოდში, ხოლო შემდეგ ხდებოდა ამ კოდის კომპილაცია. *C++* კოდი, რომელსაც კითხულობდა პრეკომპილატორი მოთავსებული იყო ფაილებში .cpp, .C ან .cc გაფართოებით. დღეს, თანამედროვე *C++* კომპილატორებს გადაჰყავთ *C++* კოდი მანქანურ კოდებში პრეკომპილაციის გარეშე. ვინაიდან *C++* წარმოადგენს *C* განვითარებას, *C++* კომპილატორებს შეუძლიათ *C* კოდის კომპილირებაც. თანამედროვე *C* უკვე აღარ წარმოადგენს *C++*-ის ქვესიმრავლეს. ეს ორი ენა ვითარდება ცალ-ცალკე. ცალ-ცალკეა *C* და *C++* ენების სტანდარტიზაციის კომისიები. რაც არის *C* სტანდარტი, მაგალითად gcc კომპილატორისთვის ის არის *C++* ნაწილი მიკროსოფტის კომპილატორისთვის.

C++ უპირატესობანი

ხშირად საუბრობენ *C++* უპირატესობაზე მოჰყავთ რა სხვდასხვა არგუმენტები, მაგალითად ის, რომ *C* ძირითადად სისტემური პროგრამირების ენაა ხოლო *C++* უფრო ზოგადი პროგრამირების ენა, რაც ობიექტური შეფასება არაა. *C* ზეც შეიძლება დაიწეროს ზოგადი მოხმარების პროგრამები, თუ ბიბლიოთეკები. ამის მაგალითია gtk პროგრამირების ბიბლიოთეკა, Gnome დესკტოპ გარემო, Gimp – ფოტოშოპის მსგავსი გრაფიკული რედაქტორი, სხვადასხვა დონის მათემატიკური ბიბლიოთეკები: gsl– გნუ სამცნიერო ბიბლიოთეკა, gts – ტრიანგულაციის ბიბლიოთეკა.

C ზე კარგად დაწერილი პროგრამა თითქმის არ ჩამოუვარდება FORTRAN პროგრამებს, თუმცა FORTRAN კვლავ რჩება სამეცნიერო გამოთვლებისთვის ყველაზე სწრაფ ენად. *C++* მთავარი უპირატესობაა მონაცემების „დამალვის“ შესაძლებლობა, ახალი ტიპების და ობიექტების უფრო ადვილად შექმნა, მაბლონური პროგრამირება, პოლიმორფიზმი და ობიექტორიენტირებულობა. ყველა სხვა „უპირატესობა“ რასაც მიაწერენ *C++* შესაძლებელია *C* ფარგლებშიც.

ობიექტზე ორიენტირებულობა და მონაცემებზე ორიენტირებულობა

ისტორიულად $C{+}{+}$ წარმოადგენს „$C+$ ობიექტზე ორიენტირებულიბის კონცეფცია“[1]. ობიექტის იდეა C შეტანილ იქნე იმ დროისათვის ცნობილი ობიექტზე ორიენტირებული ენებიდან SmallTalk და Simula . პროცედურული ენების შემთხვევაში, როგორებიცაა, მაგალითად Pascal, C, ხდება პრობლემის იდენტიფიცირება. შემდეგ ეს პრობლემა იყოფა რამდენიმე ქვეპრობლემად. შემოგვყავს თითოეული ქვეპრობლემისათვის დამახასიათებელი ცვლადები და ფუნქციები, რომლებიც შეგვიძლია გავაერთიანოთ ბიბლიოთეკაში შემდგომი გამოყენებისათვის.

ობიექტზე ორიენტირებული მიდგომისას კი, ჩვენ გამოვყოფთ ცალკეულ ობიექტებს. აღვწერთ მათ თვისებებს. ასვე აღვწერთ ამ ობიექტთა შორის ურთიერთმიმართებას. თითოეული ობიექტი არის სრულყოფილი იმ გაგებით, რომ მან იცის ყველაფერი თავის შესახებ. ასევე მან იცის რა დამოკიდებულებაში არიან მასთან სხვა ობიექტები.

როცა გვაქვს მონაცემთა დიდი რაოდენობა, აღიწეროს თითოეული ერთეული როგორც ობიექტი არაეფექტურია როგორც მეხსიერების ასევე პროგრამის სწრაფი შესრულების თვალსაზრისით. ობიექტზე ორიენტირებული ენა, მაგალითად როგორიცაა $C{+}{+}$ თუ მაღალ ეფექტურია მაგალითად მომხმარებლის გრაფიკული ინტერფეისის შესაქმნელად, გამოთვლითი ამოცანებისთვის გაცილებით ეფექტურია პროცედურული მიდგომა.

[1] $C{+}{+}$ შექმნის ისტორია ინტერვიუ სტრაუსტრუპთან(Bjarne Stroustrup)

პროგრამული კოდი

1.1 printf ფორმატი . . . 4
1.2 ფორმატირების მაგალითები C-ში . . . 7
1.3 დამრგვალება, სწორება, ნიშნის ბეჭდვა . . . 7
1.4 sizeof ოპერატორის გამოყენება . . . 10
1.5 scanf ფუნქციის სტრუქტურა . . . 11
1.6 შეკრების მარტივი პროგრამა scanf-ის გამოყენებით . . . 12
1.7 სრული ფორმატირებული ბეჭდვა C-ში . . . 13
1.8 სიმბოლოს უსაფრთხო წაკითხვა scanf-ით . . . 14
1.9 C99 პორტაბელური ტიპების ბეჭდვა . . . 15
1.10 `printf`, `double` ტიპი და თექვსმეტობითი ჩანაწერი . . . 16
1.11 `scanf`, `double` ტიპი და თექვსმეტობითი ჩანაწერი . . . 17
2.1 სამ რიცხვს შორის უდიდესი . . . 24
2.2 printf მაგალითი და შედარების ოპერატორები . . . 26
2.3 ელემენტარული კალკულატორი . . . 29
2.4 switch case მაგალითი . . . 30
2.5 კვადრატული განტოლება . . . 31
2.6 n რიცხვის ჯამი . . . 33
2.7 გამრავლების ცხრილი . . . 33
2.8 რიცხვის გამყოფი . . . 34
2.9 საერთო გამყოფი . . . 34
2.10 ორმაგი ციკლი . . . 35
2.11 ნიშნები . . . 37
2.12 თვეები . . . 42
3.1 მნიშვნელობა და მისამართი . . . 45
3.2 მიმთითებელი . . . 46
3.3 ერთი მიმთითებლის მრავალჯერადი გამოყენება . . . 48
3.4 მაკროს გვერდითი ეფექტები . . . 50
3.5 კვადრატული განტოლება . . . 51
3.6 ფუნქცია არგუმენტის გადაცემა მნიშვნელობით . . . 53
3.7 `const` მიმთითებლებსა და მითითებებთან . . . 54
3.8 არგუმენტის გადაცემა მიმთითებლით (C) . . . 55
3.9 ფუნქციის მიმთითებელი არგუმენტად . . . 57
3.10 myheader.h დეკლარაციის ფაილი . . . 59
3.11 myheader.c ფუნქციის იმპლემენტაცია . . . 60
3.12 main.c პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას. . . . 61
4.1 c11_generic_example . . . 70

4.2 constexpr-ის გამოყენება C23-ში . . . 76
4.3 _Generic-ის გამოყენება ფუნქციების გადატვირთვისთვის C11-ში . . . 78
5.1 ფაილის წაკითხვა და სიმბოლოების დათვლა . . . 84
5.2 ფაილის წაკითხვა სტრიქონ-სტრიქონ fgets-ის გამოყენებით . . . 86
5.3 ბაზური ტიპების ჩაწერა ფაილში . . . 87
5.4 ბაზური ტიპების წაკითხვა ფაილიდან . . . 88
5.5 ფაილის ზომის გამოთვლა fseek და ftell-ით . . . 90
6.1 სიმბოლოების მასივი '\0'-ის გარეშე . . . 111
6.2 სტრიქონ-სტრიქონ წაკითხვა ფაილიდან . . . 112
6.3 ფაილში ჩაწერა და წაკითხვა . . . 125
6.4 VLA-ის მხარდაჭერის შემოწმება . . . 129
6.5 არგუმენტების გადაცემა . . . 130
6.6 ორი რიცხვის შეკრება . . . 131
6.7 2D და 3D სტეკ მასივების ჩაწერა და წაკითხვა . . . 134
6.8 ორგანზომილებიანი მასივის ფაილში ჩაწერა და შერეული ფორმატით წაკითხვა 136
7.1 struct Vector3D მაგალითი . . . 153
7.2 ბიტური ველების მაგალითი . . . 154
7.3 დინამიკური სტრუქტურა . . . 155
7.4 union Vector3D მაგალითი . . . 157
7.5 rgba ფერი union . . . 158
8.1 memory_layout_example.c . . . 165
8.2 ავტომატური ცვლადები და ბლოკური ხილვადობა . . . 166
8.3 გლობალური და ლოკალური ცვლადების გამოყენება . . . 167
8.4 main.c პროექტის მთავარი ფაილი . . . 168
8.5 util.h სათაო ფაილი . . . 169
8.6 util.c იმპლემენტაციის ფაილი . . . 170
8.7 დამხმარე მოდული . . . 171
8.8 მთავარი პროგრამა . . . 172
8.9 math_utils.h . . . 173
8.10 math_utils.c . . . 174
8.11 void* მიმთითებლის გამოყენების მაგალითი . . . 179
8.12 დინამიური ბუფერის გაფართოება realloc-ის მეშვეობით . . . 180
8.13 3D დინამიური მასივის სტრუქტურული რეალიზაცია . . . 181
8.14 ჩაშენებული ციკლებიდან გამოსვლა goto-თი . . . 182
8.15 რესურსების გასუფთავება (Cleanup Pattern) C ენაში . . . 183
9.1 16 და 32-ბიტიანი ნიღბების განსაზღვრა enum და წანაცვლებით . . . 191
9.2 32-ბიტიანი დროშების მართვა . . . 192
9.3 32-ბიტიანი GPIO რეგისტრის ჯგუფური მართვა . . . 195
9.4 8, 16 და 32-ბიტიანი ბიტური ველები . . . 197
9.5 უსახელო ველების და :0-ის გამოყენება . . . 198
9.6 Unions და Bit Fields-ის კომბინაცია რეგისტრებისთვის . . . 200

10.1 მონაცემების წაკითხვა 203
10.2 c_row_col_comparison 212
10.3 c_matrix_mult_slow 214
10.4 c_matrix_mult_fast 214
10.5 Fortran-ის ოპტიმალური ციკლების თანმიმდევრობა 214
10.6 c_opengl_matrix_example 215
10.7 C მასივები restrict ოპციით 217
10.8 F90 მასივები 223

დავალება

1.1 ზრდის/შემცირების ოპერატორების ანალიზი 11
1.2 სინტაქსი, ცვლადები და შეყვანა-გამოყვანა 18
2.1 კალკულატორი 30
2.2 ტემპერატურა 31
2.3 კვადრატული განტოლება 31
2.4 ორმაგი for ციკლი 35
2.5 არაუარყოფითი რიცხვების საშუალო 39
2.6 ნაკიანი წელი 43
3.1 კალკულატორი 58
4.1 კვადრატული განტოლება 71
5.1 ფაილების კოპირება 91
5.2 სტრიქონებისა და სიტყვების დათვლა 91
5.3 სტუდენტების მონაცემთა ბაზა 91
6.1 char მასივი 95
6.2 მასივის ინდექსირება 103
6.3 რეიტინგების შოუ და ჰისტოგრამა 104
6.4 ფიბონაჩის რიცხვები 106
6.5 თვლის სისტემები 107
6.6 კვადრატული განტოლება მასივით 116
6.7 მასივის წაკითხვა/ბეჭდვა 117
6.8 მასივის დალაგება 117
6.9 3D მასივის ინდექსაცია 133
6.10 ბრძანებათა სტრიქონით ფაილების ასლი (File Copy via argc/argv) 133
6.11 2D მასივის ტრანსპონირება მიმთითებლებით 134
6.12 ბრძანებათა სტრიქონის კალკულატორი 142
6.13 ფაილის თავსართის ანალიზი და მატრიცის ტრანსპონირება 142
6.14 3D მასივის ბინარული ჩაწერა და წაკითხვა 143
7.1 სტრუქტურები და გაერთიანებები (პრაქტიკული ამოცანები) 161
8.1 დინამიური 2D მასივის სტრუქტურა 183
8.2 მეხსიერების გაჟონვის პოვნა 184
8.3 სტატიკური ცვლადების ტრეკერი 184
10.1 რთული ფორმატის ფაილის წაკითხვა 208
10.2 კონვერტირებული ფაილის წაკითხვა 209
10.3 Row-Major ↔ Column-Major 217

თავი 1

C ენის ლექსიკური ელემენტები, ცვლადები და ფუნქციები

1.1 ლექსიკური ელემენტები, ცვლადები და ტიპები 2
1.1.1 ცვლადის ატრიბუტები და დასახელების წესები 2
1.1.2 ბაზური მონაცემთა ტიპები და დიაპაზონები 2
1.2 ცვლადების დეკლარაცია და ინიციალიზაცია 3
1.3 ფორმატირებული გამოტანა: ფუნქცია printf 3
1.3.1 სპეციალური მმართველი სიმბოლოები (Escape Sequences) 3
1.3.2 მაგალითი: მმართველი სიმბოლოების გამოყენება 4
1.4 printf ფუნქციის ფორმატირება 4
1.4.1 განსხვავება %d -სა და %i -ს შორის 5
1.4.2 სიგანე და სიზუსტე (Width & Precision) 5
1.4.3 დროშები (Flags) 5
1.5 ნამდვილი რიცხვების ფორმატები: %f, %e და %g 8
1.6 sizeof ოპერატორი 9
1.6.1 sizeof — ოპერატორი თუ ფუნქცია? 10
1.7 მათემატიკური ოპერაციები და არითმეტიკა 10
1.7.1 ზრდის (Increment) და შემცირების (Decrement) ოპერატორები . . 11
1.8 ფუნქცია scanf — მონაცემების წაკითხვა კლავიატურიდან 11
1.8.1 მისამართის აღების ოპერატორი (&) 11
1.9 scanf-ის ფორმატის სტრინგში სიმბოლოების მართვა 12
1.10 ფორმატის სპეციფიკატორები და მოდიფიკატორები 13
1.11 %c სპეციფიკატორის თავისებურება scanf()-ში 14
1.12 C99 და თანამედროვე სტანდარტების სპეციფიკატორები 15
1.13 მცოცავი ტიპის ცვლადები თექვსმეტობით ფორმატში 15
1.13.1 printf-ში გამოყენება 16
1.13.2 scanf-ში გამოყენება 16
1.13.3 რატომ არის ათობითი (%f) პრობლემური? 17
1.13.4 თექვსმეტობითი (%a) ჩანაწერის უპირატესობა 17

1.1 ლექსიკური ელემენტები, ცვლადები და ტიპები

C ენის პროგრამა შედგება მცირე ენობრივი ერთეულებისგან — **ლექსიკური ელემენტებისგან** (Tokens). ესენია:

- **ლათინური ანბანის ასოები:** `A...Z`, `a...z` და ქვედა ტირე `_`;
- **ციფრები:** `0...9`;
- **ოპერატორები:** `+`, `-`, `*`, `/`, `%`, `++`, `--`, `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `&&`, `||`, `!`, `&`, `|`, `^`, `~`, `<<`, `>>`, `<<=`, `>>=`, `&=`, `^=`, `|=`, `?`, `:`, `.`, `->`;
- **პუნქტუაციისა და გამოყოფის ნიშნები:** `;`, `,`, `(`, `)`, `{`, `}`, `[`, `]`, `#`, `##`, `...`.

1.1.1 ცვლადის ატრიბუტები და დასახელების წესები

ცვლადი არის მეხსიერების დასახელებული არე, რომელშიც ინახება მონაცემი. თითოეულ ცვლადს C ენაში გააჩნია ოთხი ძირითადი ატრიბუტი:

1. **სახელი (Identifier):** რითაც მივმართავთ ცვლადს კოდში;
2. **ტიპი (Data Type):** განსაზღვრავს მონაცემის ფორმატს და მეხსიერების ზომას;
3. **მნიშვნელობა (Value):** უშუალოდ შენახული მონაცემი;
4. **მისამართი (Address):** ადგილი ოპერატიულ მეხსიერებაში (RAM).

მნიშვნელოვანია: C ენა არის რეგისტრზე დამოკიდებული (*Case-sensitive*). ეს ნიშნავს, რომ ცვლადები `Object`, `object` და `OBJECT` კომპილატორისთვის სამი სრულიად სხვადასხვა ცვლადია.

ცვლადის სახელი:

- სახელი შეიძლება შეიცავდეს ინგლისურ ასოებს, ციფრებსა და ქვედა ტირეს (_);
- სახელი **არ უნდა იწყებოდეს ციფრით**;
- სახელი არ უნდა ემთხვეოდეს C ენის რეზერვირებულ საკვანძო სიტყვებს (`int`, `return`, `if` და ა.შ.).

```c
int x = 4;    /* სწორია */
int x2 = 4;   /* სწორია */
int _x = 4;   /* სწორია */
int 2x = 4;   /* შეცდომაა! სახელი იწყება ციფრით */
```

1.1.2 ბაზური მონაცემთა ტიპები და დიაპაზონები

C ენის ბაზური მონაცემთა ტიპებია: მთელი რიცხვები (`int`), ათწილადები/მცოცავწერტილიანი რიცხვები (`float`, `double`), სიმბოლოები (`char`) და ლოგიკური ტიპი (`bool`).

ყოველ ტიპს გააჩნია ფიქსირებული ზომა ბაიტებში. ეს ზომა შეიძლება განსხვავდებოდეს ოპერაციული სისტემისა და კომპილატორის არქიტექტურის მიხედვით (მაგ. 32-ბიტიან vs 64-ბიტიან სისტემებზე).

თუ მთელი ტიპის ცვლადის ზომა მეხსიერებაში არის N ბაიტი (ანუ $8N$ ბიტი), მაშინ:

ნიშნიანი (signed) ცვლადის დიაპაზონია: $[-2^{8N-1},\ 2^{8N-1}-1]$

უნიშნო (unsigned) ცვლადის დიაპაზონია: $[0,\ 2^{8N}-1]$

მაგალითად: 1-ბაიტიანი ($N = 1$) `signed char`-ის დიაპაზონია $[-2^7, 2^7 - 1] = [-128, 127]$, ხოლო `unsigned char`-ის დიაპაზონია $[0, 2^8 - 1] = [0, 255]$.

1.2 ცვლადების დეკლარაცია და ინიციალიზაცია

ცვლადის გამოყენებამდე აუცილებელია მისი **დეკლარირება** (გამოცხადება)[1]:

```c
bool a;           /* ლოგიკური ცვლადი (C99 სტანდარტიდან: <stdbool.h>) */
int i;            /* მთელი რიცხვი */
unsigned int j;   /* უნიშნო მთელი რიცხვი (მხოლოდ >= 0) */
float x;          /* მცოცავწერტილიანი ათწილადი (Single Precision) */
double t;         /* ორმაგი სიზუსტის ათწილადი (Double Precision) */
char character;   /* სიმბოლური ცვლადი */
```

თუ ცვლადს მხოლოდ გამოვაცხადებთ და მასში მნიშვნელობას არ ჩავწერთ, მეხსიერების იმ არეში დარჩება **„ნაგავი მნიშვნელობა“ (Garbage Value)** — ანუ ის შემთხვევითი ბიტები, რომლებიც მანამდე ეწერა ოპერატიულ მეხსიერებაში. ასეთი ცვლადის წაკითხვამ შეიძლება გამოიწვიოს არაპროგნოზირებადი შედეგი ან კომპილატორის გაფრთხილება (Warning).

პროგრამირების საუკეთესო პრაქტიკაა ცვლადის **ინიციალიზაცია** (საწყისი მნიშვნელობის მინიჭება) მისი გამოცხადებისთანავე:

```c
bool a = true;          /* ლოგიკური ცვლადი */
int i = 0;              /* მთელი რიცხვი */
float x = 0.0f;         /* ათწილადი */
double t = 0.0;         /* ორმაგი სიზუსტის ათწილადი */
char character = '\0'; /* ნულ-სიმბოლო (Null Character) */
```

1.3 ფორმატირებული გამოტანა: ფუნქცია printf

ეკრანზე ტექსტისა და ცვლადების გამოსატანად გამოიყენება სტანდარტული ბიბლიოთეკის stdio.h ფუნქცია `printf()`.

1.3.1 სპეციალური მმართველი სიმბოლოები (Escape Sequences)

ტექსტში სპეციალური სიმბოლოების (მაგ. ახალ სტრიქონზე გადასვლა, ტაბულაცია) ჩასასმელად გამოიყენება \ (Backslash) პრეფიქსი:

[1] აქ საუბარია ავტომატურ (სტატიკურად დეკლარირებულ) ცვლადებზე, რომლებიც იქმნება ფუნქციის გამოძახებისას და ნადგურდება მისი დასრულებისას. ე.წ. დინამიურ ცვლადებს განვიხილავთ თავში „ცვლადები და მეხსიერება“.

ცხრილი 1.1 ძირითადი Escape-სიმბოლოები C ენაში

ჩანაწერი	სახელწოდება	აღწერა / ფუნქცია
\n	Newline	ახალ სტრიქონზე გადასვლა
\t	Horizontal Tab	ჰორიზონტალური ტაბულაცია (წანაცვლება)
\\	Backslash	უკუღმა დახრილი ხაზის (\) დაბეჭდვა
\"	Double Quote	ორმაგი ბრჭყალის (") დაბეჭდვა ტექსტში
\'	Single Quote	ერთმაგი ბრჭყალის (') დაბეჭდვა
\a	Alert / Bell	სისტემური განგაშის ხმოვანი სიგნალი
\b	Backspace	კურსორის ერთ პოზიციით უკან დაბრუნება
\xdd	Hex Character	სიმბოლოს დაბეჭდვა მისი თექვსმეტობითი კოდით (მაგ. \x41 არის 'A')

1.3.2 მაგალითი: მმართველი სიმბოლოების გამოყენება

```c
#include <stdio.h>
int main(void)
{
 printf("Testing...\n..1\n...2\n....3\n");
 printf("char \'A\'=\t teqvbsmetobiti \\x41 \x41 \n");
 printf("char \'a\'=\t teqvbsmetobiti \\x61 \x61 \n");
 return 0;
}
```

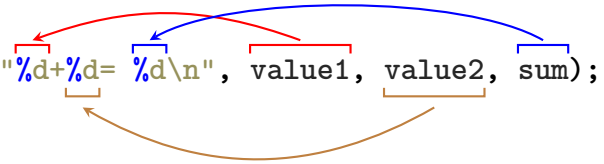
1.4 printf ფუნქციის ფორმატირება

როდესაც printf()-ით ცვლადის მნიშვნელობის დაბეჭდვა გვსურს, ფორმატის სტრინგში ჩავსვამთ **ფორმატის სპეციფიკატორს** (%d, %f და ა.შ.), ხოლო მის შემდეგ არგუმენტად გადავცემთ შესაბამის ცვლადებს.

კოდი 1.1 printf ფორმატი

```c
#include <stdio.h>
int main(void)
{/* ცვლადების მნიშვნელობების ჩასმა %d სპეციფიკატორების ადგილას */
 int value1, value2, sum;
 value1 = 50;
```

```
6    value2 = 25;
7    sum = value1 + value2;
8
9     printf ("%d+%d= %d\n", value1, value2, sum);
10
11   return 0;
12  }
```



მნიშვნელოვანი წესები:

1. ფორმატის სტრიქონში რამდენიმე % ნიშანიც არის, ზუსტად იმავე რაოდენობისა და შესაბამისი ტიპის არგუმენტები უნდა მიეწოდოს printf-ს მძიმით გამოყოფილი სახით;
2. არგუმენტების ჩასმა ხდება მარცხნიდან მარჯვნივ მიმდევრობით.

1.4.1 განსხვავება %d -სა და %i -ს შორის

- printf()-ში ორივე სპეციფიკატორი სრულიად იდენტურად მუშაობს (ბეჭდავს ათობით ნიშნიან მთელ რიცხვს).
- scanf()-ში მათ შორის არის მნიშვნელოვანი განსხვავება:
  - %d კითხულობს რიცხვს **მხოლოდ ათობით სისტემაში**;
  - %i ავტომატურად აღიქვამს რიცხვის სისტემას: თუ რიცხვი იწყება 0x-ით, წაიკითხავს როგორც თექვსმეტობითს, ხოლო თუ იწყება 0-ით — როგორც რვაობითს (მაგ. 033 წაიკითხება როგორც 27_{10}).

1.4.2 სიგანე და სიზუსტე (Width & Precision)

სიზუსტე (.precision): აკონტროლებს ციფრების რაოდენობას მძიმის შემდეგ ათწილადებში, ან მაქსიმალურ სიგრძეს სტრიქონებში.

- %.2f → დაბეჭდავს 2 ციფრს მძიმის შემდეგ (მაგ: 3.14).
- %.5s → დაბეჭდავს სტრიქონის მხოლოდ პირველ 5 სიმბოლოს.

სიგანე (width): განსაზღვრავს მინიმალურ ადგილს, რაც ველისთვის უნდა გამოიყოს.

- %10d → გამოყოფს 10 სიმბოლოს ადგილს და რიცხვს გაასწორებს მარჯვნივ.

1.4.3 დროშები (Flags)

- (მარცხნივ გასწორება): გამოყოფილი სიგანის ფარგლებში ტექსტს/რიცხვს წევს მარცხნივ (მაგ: %-10d).

+ (ნიშნის ჩვენება): დადებით რიცხვებსაც უწერს + ნიშანს (მაგ: %+d).

- printf("%d", 42); → 42
- printf("%+d", 42); → +42
- printf("%+d", -42); → -42 (უარყოფით რიცხვებზე ჩვეულებრივად მინუსი რჩება)

ეს განსაკუთრებით გამოსადეგია ფინანსურ, მათემატიკურ ან ტემპერატურის აღმრიცხველ პროგრამებში, სადაც ცვლილების მიმართულების (დადებითი/უარყოფითი) მკაფიოდ ჩვენებაა საჭირო.

`0` **(ნულებით შევსება):** ცარიელ ადგილებს ავსებს ნულებით (მაგ: `%05d`).

`#` **(ალტერნატიული ფორმატი):** ცვლის გამოსახვის სტანდარტულ სახეს მონაცემთა ტიპის მიხედვით:

- **16-ობით / 8-ობით რიცხვებში** (`%x`, `%X`, `%o`): ნულისგან განსხვავებული მნიშვნელობებისთვის ავტომატურად ამატებს შესაბამის პრეფიქსს (`0x`, `0X` ან `0`).
- **ნამდვილ რიცხვებში** (`%f`, `%e`): აიძულებს ათწილადის წერტილის (`.`) ყოველთვის გამოჩენას, იმ შემთხვევაშიც კი, როცა წერტილის შემდეგ ნულოვანი სიზუსტეა მითითებული (`%.0f`).
- **კომპაქტური ჩაწერის ნამდვილ რიცხვებში** (`%g`, `%G`): აიძულებს ათწილადის წერტილის გამოჩენას და არ აცილებს ბოლოში არსებულ ნულებს (*trailing zeros*).

C კოდის მაგალითი:

```c
#include <stdio.h>

int main(void) {
 int num = 255;
 double val = 12.0;

 /* 1. თექვსმეტობითი და რვაობითი სისტემა */
 printf("standartuli hex: %x\n", num);     /* ff   */
 printf("alternatiuli hex: %#x\n", num);   /* 0xff */
 printf("alternatiuli HEX: %#X\n", num);   /* 0XFF */
 printf("alternatiuli oct: %#o\n", num);   /* 0377 */

 /* 2. ნამდვილი რიცხვები (სიზუსტით 0) */
 printf("standartuli float: %.0f\n", val);  /* 12  */
 printf("alternatiuli float: %#.0f\n", val); /* 12. */

 /* 3. კომპაქტური ჩაწერის ნამდვილი რიცხვები (%g / %G) */
 printf("standartuli g: %g\n", val);       /* 12 */
 printf("alternatiuli g: %#g\n", val);     /* 12.0000 */

 return 0;
}
```

- ველის სიგანისა ან ათწილადის სიზუსტის კოდში მყარად გაწერის ნაცვლად, შეგიძლიათ გამოვიყენოთ ვარსკვლავი (`*`), რათა პარამეტრები ცვლადებიდან ავიღოთ:
  - `int width = 8, prec = 2;`
  - `printf("%*.*f", width, prec, 3.14159);` → ბეჭდავს `  3.14` (სიგანე 8, სიზუსტე 2).

ქვემოთ მოყვანილია C ენის პროგრამა სხვადასხვა ფორმატირებისათვის:

კოდი 1.2 ფორმატირების მაგალითები C-ში

```c
#include <stdio.h>
    int main() {
 int pos = 42;
 int neg = -42;
 double pi = 3.1415926535;

 printf("=== 1. %+ alami ===\n");
 printf("chveulebrivi: %d\n", pos);
 printf("+ nishnit:      %+d\n", pos);
 printf("uarkofiti:   %+d\n\n", neg);

 printf("=== 2. scoreba da nulebi ===\n");
 printf("nulebit shevseba: %05d\n", pos);
 printf("marjvniv (default): |%10d|\n", pos);
 printf("marckhniv (%%-):       |%-10d|\n\n", pos);

 printf("=== 3. alternatiuli forma (%%#) ===\n");
 printf("Hex prefiqsit: %#x\n\n", 255);

 printf("=== 4. dinamiuri sizuste (%%*) ===\n");
 int width = 10;
 int precision = 3;
 printf("dinamiuri  float: |%*.*f|\n", width, precision, pi);
 return 0;
}
```

დამრგვალების, სწორების და ნიშნის ბეჭდვის მაგალითი:

კოდი 1.3 დამრგვალება, სწორება, ნიშნის ბეჭდვა

```c
#include <stdio.h>
   int main()
    {
  int num = 7;
  double pi = 3.14159265;
  char text[] = "C Programming";
  // სიგანისა და ნულების მაგალითი
  printf("ID: %04d\n", num);              // შედეგი: ID: 0007
  // ათწილადის დამრგვალება
  printf("Pi: %.2f\n", pi);                // შედეგი: Pi: 3.14
```

```c
    // მარცხნივ გასწორება და სიგანე
  printf("|%-15s|\n", text);              // შედეგი: |C Programming    |
  // დადებითი ნიშნის იძულებითი ჩვენება
  printf("Temperatura: %+d C\n", 25);     // შედეგი: ტემპერატურა: +25 C
  return 0;
}
```

1.5 ნამდვილი რიცხვების ფორმატები: %f, %e და %g

C ენაში მცოცავწერტილიანი (ნამდვილი) რიცხვების გამოტანისას გამოიყენება სამი ძირითადი ფორმატის სპეციფიკატორი:

ათწილადი ფორმატი (%f / %F): რიცხვს ბეჭდავს სტანდარტულ ათწილად ფორმატში. სიზუსტის მითითებისას (%.Nf) N აღნიშნავს **ციფრების რაოდენობას მძიმის შემდეგ**.
ექსპონენციალური / სამეცნიერო ფორმატი (%e / %E): რიცხვს წარადგენს $m \times 10^n$ სახით (მაგალითად: 1.234500e+02). სიზუსტით (%.Ne) განისაზღვრება **ციფრების რაოდენობა მძიმის შემდეგ** ექსპონენციალურ ჩანაწერში.
ზოგადი / კომპაქტური ფორმატი (%g / %G): ავტომატურად ირჩევს უმოკლეს ჩანაწერს %f-სა და %e-ს შორის.

- აცილებს ბოლოში არსებულ ზედმეტ ნულებს (*trailing zeros*) და ათწილადის წერტილს.
- **მთავარი განსხვავება:** %.Ng ფორმატში N ნიშნავს **მნიშვნელოვანი ციფრების** (*significant digits*) საერთო რაოდენობას და არა ციფრთა რაოდენობას მძიმის შემდეგ.

C კოდის მაგალითი (სიზუსტე და მნიშვნელოვანი ციფრები):

```c
#include <stdio.h>

int main(void) {
  /* 1. ძირითადი განსხვავება %f, %e, %g-ს შორის */
  double val1 = 123.45;
  printf("--- 1. chveulebrivi ricxvi (123.45) ---\n");
  printf("%%f : %f\n", val1);   /* 123.450000 (6 cifri mdzimis shemdeg) */
  printf("%%e : %e\n", val1);   /* 1.234500e+02 */
  printf("%%g : %g\n\n", val1); /* 123.45 (moshorebulia zedmeti nulebi) */
  /* 2. მნიშვნელოვანი ციფრების (Significant Digits) მაგალითი */
  double val2 = 12.3456;
  printf("--- 2. sizuste val2 = 12.3456 ---\n");
  printf("%%.2f : %.2f\n", val2); /* 12.35 (2 cifri mdzimis shemdeg) */
  printf("%%.2e : %.2e\n", val2); /* 1.23e+01 (1 cifri mdzimemde + 2
  ↪ mdzimis shemdeg = 3 mnishvnelovani cifri) */
  printf("%%.2g : %.2g\n\n", val2);/* 12 (sul 2 mnishvnelovani cifri!) */
```

```c
    printf("%%.4f : %.4f\n", val2); /* 12.3456 (4 cifri mdzimis shemdeg) */
    printf("%%.4g : %.4g\n\n", val2);/* 12.35 (sul 4 mnishvnelovani cifri!)
    ↪  */
    /* 3. მცირე რიცხვები და წამყვანი ნულები */
    double val3 = 0.00123456;
    printf("--- 3. mcire ricxvi val3 = 0.00123456 ---\n");
    printf("%%.2f : %.2f\n", val3); /* 0.00 (mdzimis shemdeg 2 cifri -
    ↪ dakargulia informacia) */
    printf("%%.2g : %.2g\n", val3); /* 0.0012 (nulebi ar itvleba, sul 2
    ↪ mnishvnelovani cifri: 1 da 2) */
    printf("%%.4g : %.4g\n", val3); /* 0.001235 (sul 4 mnishvnelovani cifri
    ↪ : 1, 2, 3, 5) */
    return 0;
}
```

პროგრამის შედეგი:

```
c:\users\username>     --- 1. chveulebrivi ricxvi (123.45) ---
c:\users\username>     %f : 123.450000
c:\users\username>     %e : 1.234500e+02
c:\users\username>     %g : 123.45
c:\users\username>
c:\users\username>     --- 2. sizuste val2 = 12.3456 ---
c:\users\username>     %.2f : 12.35
c:\users\username>     %.2e : 1.23e+01
c:\users\username>     %.2g : 12
c:\users\username>
c:\users\username>     %.4f : 12.3456
c:\users\username>     %.4g : 12.35
c:\users\username>
c:\users\username>     --- 3. mcire ricxvi val3 = 0.00123456 ---
c:\users\username>     %.2f : 0.00
c:\users\username>     %.2g : 0.0012
c:\users\username>     %.4g : 0.001235
```

1.6 sizeof ოპერატორი

`sizeof` არის C ენის **კომპილაციის დროის ოპერატორი**, რომელიც აბრუნებს ცვლადის ან ტიპის ზომას ბაიტებში (`size_t` ტიპის სახით).

კოდი 1.4 sizeof ოპერატორის გამოყენება

```c
#include <stdio.h>

int main(void)
{
 int a = 10;
 printf("int zoma: %zu byte\n", sizeof(int));
 printf("ცვლად a-ს zoma: %zu byte\n", sizeof(a));
 printf("double zoma: %zu byte\n", sizeof(double));
 return 0;
}
```

1.6.1 sizeof — ოპერატორი თუ ფუნქცია?

`sizeof` **ოპერატორია** (Operator) და არა ფუნქცია(ხშირად ის ფუნქცია ჰგონიათ იმის გამო, რომ გვერდით ფრჩხილები ახლავს ხოლმე (`sizeof(int)`), თუმცა ეს უბრალოდ სინტაქსური წესია.):

1. **ტიპის არგუმენტად გადაცემა:** C ენაში ფუნქციას არგუმენტად მონაცემთა ტიპს (მაგ. `int`, `double`, `char`) ვერ გადასცემ — ფუნქცია იღებს მხოლოდ მნიშვნელობებს ან ცვლადებს. `sizeof`-ს კი გადავცემთ ტიპს : `sizeof(double)`.
2. **ფრჩხილები სავალდებულო არ არის:** ცვლადის ზომის გაგებისას ფრჩხილების დაწერა საერთოდ არ არის საჭირო! ჩანაწერი `sizeof a` სრულიად ვალიდურია. ფუნქციას კი ფრჩხილების გარეშე ვერასდროს გამოიძახებ.
3. **კომპილაციის დრო (Compile-time evaluation):** `sizeof`-ის მნიშვნელობას კომპილატორი ითვლის **კომპილაციის ეტაპზევე** (სანამ პროგრამა გაეშვება). მანქანურ კოდში ფუნქციის გამოძახება საერთოდ არ ხდება — კომპილატორი `sizeof(int)`-ის ადგილას პირდაპირ ჩასვამს კონკრეტულ რიცხვს (მაგ. 4).
4. **მისამართის აღება შეუძლებელია:** ფუნქციას გააჩნია თავისი მისამართი ოპერატიულ მეხსიერებაში და შეგიძლია შექმნა მიმთითებელი ფუნქციაზე. `sizeof`-ზე მისამართის აღება (`&sizeof`) შეუძლებელია.

შენიშვნა: ერთადერთი გამონაკლისი, როდესაც `sizeof` კომპილაციისას კი არა, პროგრამის მსვლელობისას (Runtime) გამოითვლება, არის C99-ში დამატებული ცვლადი ზომის მასივები (VLA — Variable Length Arrays). თუმცა ამ შემთხვევაშიც ის მაინც ოპერატორად რჩება.

1.7 მათემატიკური ოპერაციები და არითმეტიკა

C ენაში გამოიყენება სტანდარტული არითმეტიკული ოპერატორები: `+`, `-`, `*`, `/` და `%` (ნაშთიანი გაყოფა).

ყურადღება — მთელი რიცხვების გაყოფა: ორი მთელი რიცხვის გაყოფისას (`int / int`) შედეგიც ყოველთვის **მთელი რიცხვია** (ათწილადი ნაწილი უბრალოდ იგნორირდება). მაგალითად:

- 5 / 2 იძლევა 2-ს და არა 2.5-ს!
- ათწილადი შედეგის მისაღებად ერთ-ერთი ოპერანდი მაინც უნდა იყოს ნამდვილი რიცხვი: 5.0 / 2 იძლევა 2.5-ს.

ნაშთის ოპერატორი (%) გამოიყენება მხოლოდ მთელი რიცხვისთვის (მაგ. 7 % 3 აბრუნებს 1-ს).

1.7.1 ზრდის (Increment) და შემცირების (Decrement) ოპერატორები

ცვლადის 1-ით გასაზრდელად ან შესამცირებლად გამოიყენება ++ და -- ოპერატორები. არსებობს მათი ორი ფორმა:

პრეფიქსული (++a): ჯერ იზრდება ცვლადის მნიშვნელობა 1-ით, შემდეგ სრულდება გამოსახულება;

პოსტფიქსული (a++): ჯერ გამოიყენება ცვლადის მიმდინარე მნიშვნელობა გამოსახულებაში, ხოლო ოპერაციის დასრულების შემდეგ იზრდება 1-ით.

დავალება 1.1: ზრდის/შემცირების ოპერატორების ანალიზი

რა დაიბეჭდება ეკრანზე შემდეგი კოდის შესრულებისას?

```
1  int a, b, c = 0;
2  a = ++c;
3  b = c++;
4  printf("%d, %d, %d", a, b, ++c);
```

1.8 ფუნქცია scanf — მონაცემების წაკითხვა კლავიატურიდან

კლავიატურიდან მონაცემების წასაკითხად გამოიყენება scanf() ფუნქცია.

კოდი 1.5 scanf ფუნქციის სტრუქტურა

```
1
2  scanf("%d,%d,%d\n",&value1,&value2,&value3);
3
```

1.8.1 მისამართის აღების ოპერატორი (&)

scanf() ფუნქციაში ცვლადის წინ აუცილებლად უნდა დავწეროთ **მისამართის აღების ოპერატორი &** (Ampersand).

&value1 ნიშნავს: „გადაეცი scanf-ს value1 ცვლადის **მეხსიერების მისამართი**, რათა მან წაკითხული მნიშვნელობა პირდაპირ იმ მეხსიერების უჯრაში ჩაწეროს“.

კოდი 1.6 შეკრების მარტივი პროგრამა scanf-ის გამოყენებით

```c
#include <stdio.h>
int main(void)
{
 int num1, num2, sum;
 printf("Enter first integer: ");
 scanf("%d", &num1);

 printf("Enter second integer: ");
 scanf("%d", &num2);

 sum = num1 + num2;
 printf("Sum is %d\n", sum);
 return 0;
}
```

1.9 scanf-ის ფორმატის სტრინგში სიმბოლოების მართვა

scanf-ში ფორმატის ჩანაწერში სიმბოლოების (მძიმე, ხარე) ჩაწერა განსაზღვრავს მომხმარებლისგან შეყვანის მკაცრ ფორმატს.

1. %d%d%d — თავისუფალი შეყვანა (ხარე, Tab, Enter)

%d, %f და %s სპეციფიკატორები ავტომატურად იგნორირებას უკეთებენ ნებისმიერ ჰაეროვან/ცარიელ სიმბოლოს (Whitespace: ხარე, Tab, ახალი სტრიქონი).

მისაღები შეყვანა:

ხარე: 10 20 30
Enter-ით: 10 → Enter → 20 → Enter → 30
Tab-ით: 10 20 30

შეცდომა: თუ მომხმარებელი აკრეფს 10,20,30, პირველი %d წაიკითხავს 10-ს, წააწყდება მძიმეს (,) და გაჩერდება. მეორე %d ელოდება ციფრს, მაგრამ ხვდება მძიმეს — კითხვა ჩაიშლება (Matching Failure).

2. %d,%d,%d — მკაცრი შეყვანა მძიმით

ფორმატის ჩანაწერში მძიმის ჩაწერა ნიშნავს, რომ მომხმარებელმა სავალდებულოდ მძიმე უნდა ჩაწეროს რიცხვებს შორის.

მისაღები შეყვანა: 10,20,30 (ან 10, 20, 30, რადგან %d მძიმის შემდგომ ხარეს მაინც გამოტოვებს).

შეცდომა: თუ მომხმარებელი აკრეფს 10 20 30 (მძიმის გარეშე), პირველი %d წაიკითხავს 10-ს, შემდეგ scanf ელოდება მძიმეს, მაგრამ ხვდება ხარეს და კითხვას წყვეტს.

ერთადერთი რამ, რასაც scanf-ში უნდა ვერიდოთ არის ბოლოში ცარიელი ადგილის (ხარე) ან „\n“-ის ჩასმა:

```c
// არასწორია! პროგრამა "გაიჭედება" ბოლოში:
scanf("%lf %lf %lf ", &a, &b, &c);
scanf("%lf %lf %lf\n", &a, &b, &c);
```

1.10 ფორმატის სპეციფიკატორები და მოდიფიკატორები

C ენაში `printf` და `scanf` ფუნქციების ფორმატირების სტრიქონი ექვემდებარება შემდეგ ზოგად იერარქიას:

% [ალმები] [სიგანე] [.სიზუსტე] [სიგრძის მოდიფიკატორი] სპეციფიკატორი

კონკრეტული მაგალითი: `%+015.4Lf`

განვიხილოთ რთული ფორმატის სტრიქონი: `"%+015.4Lf"`

1. **`%`:** სპეციფიკატორის დასაწყისის აღმნიშვნელი სიმბოლო;
2. **`+0` (ალმები / Flags):**
   `+` — აიძულებს დადებით რიცხვსაც წაუმძღვაროს პლუსის ნიშანი (+);
   `0` — ცარიელი ადგილების ნაცვლად გამოიყენებს წამყვან ნულებს (*Leading zeros*);
3. **`15` (ველის სიგანე / Width):** გამოსატანი ველის მინიმალური სიგანე (სულ 15 სიმბოლო);
4. **`.4` (სიზუსტე / Precision):** ზუსტად 4 ციფრი ათწილადის წერტილის შემდეგ;
5. **`L` (სიგრძის მოდიფიკატორი / Length Modifier):** მიუთითებს `long double` ტიპზე;
6. **`f` (სპეციფიკატორი / Specifier):** ათწილადი რიცხვის სტანდარტული ფორმატი.

კოდი 1.7 სრული ფორმატირებული ბეჭდვა C-ში

```c
#include <stdio.h>

int main(void) {
 long double num = 123.456789;
 printf("Shedegi: [%+015.4Lf]\n", num);
 /* ეკრანზე გამოიტანს: [Shedegi: [+000000123.4568]] (სულ 15 სიმბოლო) */
 return 0;
}
```

ცხრილი 1.2 printf და ტიპების სპეციფიკატორები

ტიპი	მაგალითი	სპეციფიკატორი
char	'a', '\n'	%c
char[] (სტრიქონი)	"Hello C"	%s
bool (C99)	true, false	%d, %i
short int	-32767	%hi, %hx, %ho
int	-123, 0xFF	%d, %i, %x, %X, %o
unsigned int	123U	%u, %x, %X, %o
long int	123456L	%ld, %lx, %lo
long long int (C99)	123456789LL	%lld, %llu, %llx
size_t (C99)	sizeof(int)	%zu
ptrdiff_t (C99)	ptr2 - ptr1	%td
float	12.34f, 3.4e-7f	%f, %e, %E, %g, %a (C99)
double	12.345, 3.4e-7	%f, %lf, %e, %g, %a (C99)
long double	12.345L	%Lf, %Le, %Lg, %La (C99)
void* (მითითებელი)	&x, ptr	%p
int* (სიმბოლოების თვლა)	&count	%n

1.11 %c სპეციფიკატორის თავისებურება scanf()-ში

%c გამოიყენება თითო სიმბოლოს წასაკითხად. თუმცა, განსხვავებით %d-სგან ან %f-სგან, **%c არ ტოვებს ხარეებს (Space) და ახალ სტრიქონებს (Enter)**.

თუ მომხმარებელმა წინა შეყვანისას დააჭირა Enter-ს, მეხსიერების ბუფერში რჩება \n სიმბოლო. შემდეგი scanf("%c", &ch) წაიკითხავს სწორედ ამ \n-ს და არ დაელოდება ახალ შეყვანას!

გამოსავალი: scanf()-ში %c-ს წინ ჩავსვათ ცარიელი ადგილი (Space):

კოდი 1.8 სიმბოლოს უსაფრთხო წაკითხვა scanf-ით

```
1  #include <stdio.h>
2
```

```c
int main(void) {
 char ch;
 printf("Enter a character: ");
 /* " %c" აიძულებს scanf-ს გამოტოვოს ყველა წინა Space და Enter */
 scanf(" %c", &ch);
 printf("You entered: %c\n", ch);
 return 0;
}
```

1.12 C99 და თანამედროვე სტანდარტების სპეციფიკატორები

64-ბიტიანი არქიტექტურისა და პორტაბელური კოდის წერის უზრუნველსაყოფად C99 სტანდარტში დაემატა სპეციალური მოდიფიკატორები:

%zu (size_t): გამოიყენება `sizeof` და `strlen()` შედეგების დასაბეჭდად;

%td (ptrdiff_t): გამოიყენება მიმთითებლების გამოკლების შედეგის (სხვაობის) დასაბეჭდად;

%jd / %ju (intmax_t / uintmax_t): სისტემაში არსებული მაქსიმალური ტევადობის მთელი ტიპებისთვის;

<inttypes.h> მაკროსები (PRId32, PRIu64): ფიქსირებული ზომის ტიპების (`int32_t`, `uint64_t`) პორტაბელური ბეჭდვისთვის.

კოდი 1.9 C99 პორტაბელური ტიპების ბეჭდვა

```c
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main(void) {
 size_t sz = sizeof(double);
 uint64_t val64 = 9876543210123456789ULL;

 printf("Size of double: %zu bytes\n", sz);
 printf("uint64_t: %" PRIu64 "\n", val64);
 return 0;
}
```

1.13 მცოცავი ტიპის ცვლადები თექვსმეტობით ფორმატში

C ენაში (C99 სტანდარტიდან) %a (და %A) არის ფორმატირების სპეციფიკატორი მცოცავწერტილიანი (floating-point) რიცხვების თექვსმეტობით (Hexadecimal) ფორმატში გამოსატანად ან წასაკითხად.

თექვსმეტობითი ფლოატები (%a) გამოიყენება მაშინ, როდესაც საჭიროა მცოცავწერტილიანი რიცხვის აბსოლუტური სიზუსტით (ყოველგვარი დამრგვალებისა და დაკარგვის

გარეშე) შენახვა, გადაცემა ან დებაგირება.

1.13.1 printf-ში გამოყენება

printf-ში %a რიცხვს გამოსახავს [-]0xh.hhhhp±d ფორმატით, სადაც:
0x — თექვსმეტობითი სისტემის პრეფიქსი.
h.hhhh — მანტისა თექვსმეტობით ფორმატში.
p±d — ექსპონენტი 2-ის ხარისხში (binary exponent) ათობით ფორმატში.
შენიშვნა: %a იყენებს პატარა ასოებს (0x და p), ხოლო %A — დიდ ასოებს (0X და P).
მაგალითი:

კოდი 1.10 printf, double ტიპი და თექვსმეტობითი ჩანაწერი

```c
#include <stdio.h>

int main() {
 double num = 15.75;

 printf("atobiti (%%f): %f\n", num);
 printf("teqvsmetobiti (%%a): %a\n", num);
 printf("teqvsmetobiti (%%A): %A\n", num);

 return 0;
}
```

ეკრანზე დაიბეჭდება:

```
c:\users\username>   ათობითი (%f): 15.750000
c:\users\username>   თექვსმეტობითი (%a): 0x1.f8p+3
c:\users\username>   თექვსმეტობითი (%A): 0X1.F8P+3
```

როგორ ითვლება `0x1.f8p+3`?

$$1.\mathrm{F8}_{16} = 1 + \frac{15}{16} + \frac{8}{256} = 1.96875$$

$p + 3$ ნიშნავს $2^3 = 8$-ზე გამრავლებას და

$$1.96875 \times 8 = 15.75$$

.

1.13.2 scanf-ში გამოყენება

C99 სტანდარტში scanf-ის დროს %a, %e, %f და %g ფუნქციურად იდენტურია — ოთხივე გამოიყენება მცოცავწერტილიანი რიცხვის წასაკითხად. მას შეუძლია წაიკითხოს რიცხვი

როგორც ათობით, ისე თექვსმეტობით ფორმატში (მაგალითად, `15.75` ან `0x1.f8p+3`).

მაგალითი:

კოდი 1.11 `scanf`, `double` ტიპი და თექვსმეტობითი ჩანაწერი

```c
#include <stdio.h>

int main() {
 double val;

 printf("shemoitanet rickhvi (mag: 0x1.f8p+3 an 15.75): ");
 scanf("%la", &val);
 // double ტიპისთვის გამოიყენება %la (ხოლო float-ისთვის %a)

 printf("cakitkhuli mnishvneloba atobitshi: %f\n", val);

 return 0;
}
```

1.13.3 რატომ არის ათობითი (%f) პრობლემური?

მთავარი მიზეზი, რის გამოც \%a ფორმატი არსებობს, არის ის, რომ კომპიუტერი რიცხვებს ინახავს ორობით (Binary) სისტემაში.

ათობითი წილადების უმრავლესობა ორობით სისტემაში ზუსტად არ გამოისახება. მაგალითად, რიცხვი 0.1 ათობითში მარტივია, მაგრამ ორობითში უსასრულო პერიოდული წილადია (როგორც $1/3 = 0.333...$ ათობითში).როდესაც printf("%f", num)-ს იყენებ, კომპილატორი ორობით რიცხვს გარდაქმნის ათობითში, რისი დროსაც ხდება დამრგვალება. თუ ამ ათობით რიცხვს მოგვიანებით ისევ წაიკითხავ (scanf), შესაძლოა მიიღო ბოლო ბიტში ოდნავ განსხვავებული მნიშვნელობა.

1.13.4 თექვსმეტობითი (%a) ჩანაწერის უპირატესობა.

რადგან $16 = 2^4$, თითოეული თექვსმეტობითი ციფრი ზუსტად შეესაბამება 4 ბიტს. შესაბამისად, %a-ით რიცხვის გამოსახვისას არ ხდება არანაირი დამრგვალება — ის პირდაპირ აჩვენებს მეხსიერებაში არსებულ ბიტებს.

საიდან და როდის გამოიყენება:

1. მონაცემთა ზუსტი შენახვა და გადაცემა (Lossless Serialization)თუ გსურს double ტიპის ცვლადი შეინახო ტექსტურ ფაილში (JSON, XML, კონფიგურაცია) და მოგვიანებით წაიკითხო ისე, რომ 1 ბიტიც კი არ შეიცვალოს, ათობითი ფორმატი არ გამოდგება. %a-ით ჩაწერილი რიცხვი უკან წაკითხვისას გარანტირებულად აღადგენს იდენტურ ბიტებს.
2. სისტემური და მათემატიკური ბიბლიოთეკები (libm). დაბალი დონის ბიბლიოთეკების დეველოპერები (რომლებიც წერენ sin(), cos(), exp(), sqrt() ფუნქციებს) კოდში მათემატიკურ კონსტანტებს ხშირად hex-float ფორმატში წერენ:

```c
// C99 სტანდარტში შესაძლებელია ასეთი კონსტანტის პირდაპირ მითითება:
double pi_exact = 0x1.921fb54442d18p+1; // Pi-ს ზუსტი ბიტური მნიშვნელობა
```

ეს უზრუნველყოფს, რომ სხვადასხვა კომპილატორმა ან პროცესორმა კონსტანტა სხვადასხვანაირად არ დაამრგვალოს.

3. აპარატურული დონე და „დებაგირება“ (Hardware & Drivers):
   ა) პროცესორების და GPU-ების ტესტირება: როდესაც წერ დრაივერს ან ამოწმებ გრაფიკული ბარათის გამოთვლებს, %a გიჩვენებს ზუსტად რა წერია რეგისტრებში.
   ბ) Floating-point ცდომილებების ძებნა: თუ ალგორითმი იძლევა არაზუსტ შედეგს, %a-ით ადვილად დაინახავ, კონკრეტულად მანტისის რომელ ბიტში მოხდა ცდომილება.
4. სპეციალური მნიშვნელობების იდენტიფიცირება %a გვაძლევს საშუალებას მკაფიოდ დავინახოთ არანორმალიზებული რიცხვები (Subnormal numbers), NaN (Not a Number) ან Infinity-ს ბიტური სტრუქტურა.

დავალება 1.2: სინტაქსი, ცვლადები და შეყვანა-გამოყვანა

1. დაწერეთ პროგრამა, რომელიც ეკრანზე დაბეჭდავს შემდეგ ტექსტს ზუსტად ამ ფორმატით:

```
C ენაში მნიშვნელოვანია პატარა და დიდი ასოები.
"main" არის ფუნქცია, რომელიც იწყებს პროგრამის შესრულებას.
ფრჩხილებში '{' და '}' მოთავსებული ბრძანებები ქმნიან ბლოკს.
```

2. იპოვეთ და გაასწორეთ შეცდომები ქვემოთ მოყვანილ კოდში:

```c
#include <stdio.h>
main(Void)
{
 INT sum;
 sum = 25 + 37 - 19
 printf ("The answer is %i\n" sum);
 return 0;
}
```

3. დაწერეთ პროგრამა, რომელიც მომხმარებელს კლავიატურიდან შეაყვანინებს ორ მთელ რიცხვს და დაბეჭდავს მათ ჯამს, სხვაობას, ნამრავლსა და განაყოფს (ათწილადის სახით!).

4. გამოითვალეთ და შეამოწმეთ, რა შედეგს დაბეჭდავს გამოსახულება `printf("%d", 5 + 3 * 2 % 4);`.

თავი 2

გადაწყვეტილების მიღება და ციკლები

2.1 შედარების (პირობითი) ოპერატორები (<, >, <=, >=, ==, !=) . . . 21
2.1.1 ლოგიკური ოპერატორები (&&, ||, !) . . . 22
2.2 'if' და 'if ... else' ბრძანება . . . 22
2.3 ბრძანება switch ... case . . . 27
2.4 ციკლები . . . 32
2.4.1 for ციკლი . . . 32
2.4.2 while ციკლი . . . 35
2.4.3 ბრძანება break და continue . . . 38
2.5 ჩამონათვალი (დანომრილი) ტიპი enum . . . 40

ხშირად სხვადასხვა შემთხვევაში საჭიროა, რომ პროგრამამ მიიღოს სხვადასხვა გადაწყვეტილება ან შეასრულოს სხვადასხვა ბრძანება. მაგალითად, როცა ერთი რიცხვი მეტია მეორეზე, მაშინ სრულდება ერთი ბრძანება, ხოლო თუ არა – მეორე. ან თაგუნას მარჯვენა ღილაკის დაჭერისას ერთი, მარცხენას დაჭერისას სხვა; ან „როცა დაჭერილია Shift კლავიატურის ღილაკი და დაჭერილ მდგომარეობაშია თაგუნას მარცხენა ღილაკი, მაშინ თაგუნას მოძრაობისას კეთდება.....“ და ა.შ.

ასეთმა ბრძანებებმა რომ იმუშაოს, საჭიროა გარკვეული პირობების შემოწმება და შემდეგ, პასუხიდან გამომდინარე, საჭირო ბრძანების შესრულება. ასეთი ბრძანებებია if, if else, switch.

იმისათვის, რომ ამ ბრძანებებმა იმუშაოს, საჭიროა ე.წ. შედარების ოპერატორები.

2.1 შედარების (პირობითი) ოპერატორები (<, >, <=, >=, ==, !=)

`<`, `>`, `<=`, `>=` ამ ოპერატორების აზრი გასაგებია.
`==` ამოწმებს მის მარჯვნივ და მარცხნივ მდგომი სიდიდეების ტოლობას.
`!=` ამოწმებს მის მარჯვნივ და მარცხნივ მდგომი სიდიდეების უტოლობას.
სხვაგვარად რომ ვთქვათ, თუ != მოგვცა მნიშვნელობა „მცდარია“, ეს ნიშნავს, რომ ტოლი ყოფილა. იმის მიხედვით, თუ რა გვსურს ან რომელი მოგვწონს, შეგვიძლია გამოვიყენოთ == ან !=.

კრიტიკულად მნიშვნელოვანია — შედარება vs მინიჭება:
- = არის **მინიჭების ოპერატორი** (მაგ. `x = 5;` ცვლადში წერს 5-ს).
- == არის **შედარების ოპერატორი** (მაგ. `if (x == 5)` ამოწმებს, უდრის თუ არა x 5-ს).

დამწყებთა ყველაზე ხშირი შეცდომაა `if (x = 5)` ჩანაწერი, რომელიც შედარების ნაცვლად ცვლადს მნიშვნელობას ანიჭებს და პირობას ყოველთვის ჭეშმარიტად (True) აღიქვამს!

2.1.1 ლოგიკური ოპერატორები (&&, ||, !)

რთული პირობების შესადგენად (როცა რამდენიმე პირობის ერთდროულად შემოწმება გვსურს) გამოიყენება ლოგიკური ოპერატორები:

&& (ლოგიკური „და“ - AND): ჭეშმარიტია მხოლოდ მაშინ, როცა **ორივე** პირობა ჭეშმარიტია.

|| (ლოგიკური „ან“ - OR): ჭეშმარიტია მაშინ, როცა **ერთ-ერთი მაინც** პირობა ჭეშმარიტია.

! (ლოგიკური უარყოფა - NOT): ცვლის მნიშვნელობას საპირისპიროთი (True ხდება False და პირიქით).

2.2 'if' და 'if ... else' ბრძანება

if შეიძლება გამოყენებულ იქნას სხვადასხვა ფორმით, იმის მიხედვით, თუ რამდენად კომპლექსური პირობებია შესამოწმებელი.

მარტივი if პირობა:

```
if('გამოსახულება') 'ბრძანება1';
```

ამ შემთხვევაში, თუ 'გამოსახულება' ჭეშმარიტია, სრულდება 'ბრძანება1' და შემდეგ გრძელდება პროგრამის შესრულება.

```
if('გამოსახულება')
{
 // ბრძანებები
 'ბრძანება1';
 'ბრძანება2';
 'ბრძანებაN';
}
```

ამ შემთხვევაშიც მხოლოდ მაშინ, როცა 'გამოსახულება' ჭეშმარიტია, სრულდება '{ }' ბლოკში მოთავსებული ბრძანებები და გრძელდება პროგრამის შესრულება.

if ··· else შემთხვევაში გვაქვს:

```
if('გამოსახულება')
{ // სრულდება თუ 'გამოსახულება' ჭეშმარიტია
  'ბრძანება1';
  'ბრძანება2';
   ....
```

```
  'ბრძანებაN';
 }
 else
 { // სრულდება თუ 'გამოსახულება' მცდარია
  'ბრძანება11';
  'ბრძანება12';
  ....
  'ბრძანება1N';
 }
 // გრძელდება პროგრამის შესრულება.
```

განვიხილოთ მაგალითები:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5   int x, y;
6   x = 15;
7   y = 13;
8   if (x > y)
9   {
10   printf("x > y");
11  }
12  return 0;
13 }
```

if ... else პირობების მაგალითი:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5   int x, y;
6   x = 15;
7   y = 18;
8   if (x > y)
9   {
10   printf("x > y");
11  }
12  else
```

```
13   {
14    printf("y > x");
15   }
16   return 0;
17  }
```

შესაძლებელია გვქონდეს ერთმანეთში ჩადგმული რამდენიმე if ... else:

```
if( გამოსახულება )
{
 if( გამოსახულება1 )
 {
  ბრძანებების_ბლოკი1;
 }
 else
 {
  ბრძანებების_ბლოკი2;
 }
}
else
{
 ბრძანებების_ბლოკი3;
}
```

მაგალითი:

კოდი 2.1 სამ რიცხვს შორის უდიდესი

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5   int a, b, c;
6   printf("enter 3 number"); //  ვკითხულობთ a,b,c;
7   scanf("%d%d%d", &a, &b, &c);
8   if (a > b)
9   {
10   if (a > c)
11   {
12    printf("a is greatest");
13   }
```

```
14    else
15    {
16     printf("c is greatest");
17    }
18   }
19   else
20   {
21    if (b > c)
22    {
23     printf("b is greatest");
24    }
25    else
26    {
27     printf("c is greatest");
28    }
29   }
30   return 0;
31  }
```

if .. else if კონსტრუქციის მაგალითი:

```
if(piroba 1)
{
 statement-block1;
}
else if(piroba 2)
{
 statement-block2;
}
else if(piroba 3 )
{
 statement-block3;
}
else
default-statement;
```

ასეთ კონსტრუქციაში სრულდება მხოლოდ ის ბლოკი, სადაც 'piroba n' ჭეშმარიტია. თუ არცერთი არაა ჭეშმარიტი, სრულდება 'else' ანუ 'default-statement'.

if-ში შეიძლება იყოს რამდენიმე პირობა:

```c
#include <stdio.h>

int main(void)
{
 int a;
 printf("shemoitane ricxvi"); // კომენტარი
 scanf("%d", &a);
 if (a % 5 == 0 && a % 8 == 0)
 {
  printf("iyofa 5 ze da 8-ze");
 }
 else if (a % 8 == 0)
 {
  printf("iyofa 8-ze");
 }
 else if (a % 5 == 0)
 {
  printf("iyofa 5-ze");
 }
 else
 {
  printf("ar iyofa arc 5-ze arc 8-ze");
 }
 return 0;
}
```

კოდი 2.2 printf მაგალითი და შედარების ოპერატორები

```c
// Fig. 2.13: fig02_13.c
// Using if statements, relational
// operators, and equality operators
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int num1; // first number to be read from user
   int num2; // second number to be read from user

   printf( "Enter two integers, and I will tell you\n" );
```

```
13    printf( "the relationships they satisfy: " );
14
15    scanf( "%d%d", &num1, &num2 ); // read two integers
16
17    if ( num1 == num2 ) {
18       printf( "%d is equal to %d\n", num1, num2 );
19    } // end if
20
21    if ( num1 != num2 ) {
22       printf( "%d is not equal to %d\n", num1, num2 );
23    } // end if
24
25    if ( num1 < num2 ) {
26       printf( "%d is less than %d\n", num1, num2 );
27    } // end if
28
29    if ( num1 > num2 ) {
30       printf( "%d is greater than %d\n", num1, num2 );
31    } // end if
32
33    if ( num1 <= num2 ) {
34       printf( "%d is less than or equal to %d\n", num1, num2 );
35    } // end if
36
37    if ( num1 >= num2 ) {
38       printf( "%d is greater than or equal to %d\n", num1, num2 );
39    } // end if
40 } // end function main
```

2.3 ბრძანება switch ... case

if ოპერატორებით შესაძლებელია სხვადასხვა ბრძანებების ბლოკის გაშვება სხვადასხვა შემთხვევისათვის, მაგრამ კომპლექსური პირობებისთვის შეიძლება ბევრმა ერთმანეთში ჩადგმულმა პირობამ პროგრამული კოდი გახადოს რთულად გასაგები.

არის ამოცანები, სადაც „თუ/თუ არა“ კონსტრუქციის ნაცვლად რაიმე ცვლადის ან ცვლადების მდგომარეობა შეიძლება აღიწეროს ერთი მთელი ტიპის ცვლადით და იმის მიხედვით, თუ რისი ტოლია ამ ცვლადის მნიშვნელობა, შესრულდეს სხვადასხვა ბრძანება (ან ბრძანებების ბლოკი). ასეთი კონსტრუქციის საშუალებას იძლევა switch ... case.

ბრძანების სინტაქსი:

```
switch (n) {
 case constant1:
 // brdazneba romelic sruldeba roca n=constant1;
 break;
 case constant2:
 // brdazneba romelic sruldeba roca n=constant2;
 break;
 default:
 // brdazneba romelic sruldeba roca n aris ragac sxva;
}
```

n შეიძლება იყოს ან ასო-ნიშანი ან მთელი ტიპის ცვლადი. როცა n ტოლია ერთ-ერთი 'constant'-ის, მაშინ სრულდება შესაბამისი ბრძანება (ან ბრძანებები). ბრძანების შემდეგ წერია break, რაც ნიშნავს, რომ უნდა შეწყდეს switch ბრძანება. თუ რომელიმე ან რამდენიმე case-ში არაა მითითებული break, მაშინ case მუშაობს შემდეგი სახით: სრულდება ის შემთხვევა, როცა n = constant. ამის შემდეგ შესრულდება ყველა დანარჩენი case, სანამ არ შეხვდება ბრძანება break.

განვიხილოთ მაგალითი:

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[])
4  {
5   int n = 3;
6   switch (n)
7   {
8    case 1:
9    printf("n=1\n");
10   break;
11   case 2:
12   printf("n=2\n");
13   case 3:
14   printf("n=3\n");
15   case 4:
16   printf("n=4\n");
17   case 5:
18   printf("n=5\n");
19   break;
20   default:
21   printf("\n n is not defined");
```

```
 }
 return 0;
}
```

კოდი 2.3 ელემენტარული კალკულატორი

```
#include <stdio.h>

int main(void) {
 char o;
 float num1, num2;
 printf("Select an operator either + or - or * or / \n");
 scanf("%c", &o);
 printf("Enter two operands: ");
 scanf("%f%f", &num1, &num2);
 switch(o) {
  case '+':
  printf("%.1f + %.1f = %.1f", num1, num2, num1 + num2);
  break;
  case '-':
  printf("%.1f - %.1f = %.1f", num1, num2, num1 - num2);
  break;
  case '*':
  printf("%.1f * %.1f = %.1f", num1, num2, num1 * num2);
  break;
  case '/':
  printf("%.1f / %.1f = %.1f", num1, num2, num1 / num2);
  break;
  default:
  /* If operator is other than +, -, * or /, */
  /* error message is shown */
  printf("Error! operator is not correct");
  break;
 }
 return 0;
}
```

მნიშვნელოვანია — break-ის როლი: თუ case-ის ბოლოს არ დავწერთ break; ბრძანებას, პროგრამა არ გამოვა switch-იდან! ის გააგრძელებს მომდევნო case-ების კოდის ავტომატურ შესრულებას (ამ

მოვლენას *Fall-through* ეწოდება).

დავალება 2.1: კალკულატორი

შექმენით პროექტი და გაარჩიეთ ზემოთ მოყვანილი კოდი „ელემენტარული კალკულატორი“.

ქვემოთ მოყვანილია მაგალითი `switch` ბრძანების გამოყენებით:

კოდი 2.4 switch case მაგალითი

```
1  // case_test.cpp : Defines the entry point for the console application.
2  //switch case test programm case_test.c
3  #include <stdio.h>
4  int main(int argc, char* argv[])
5  {
6   float T=0;
7   int N=-1;
8   float Tgrad=0;
9   float Tfaren=0;
10  puts("shemoitanet 1 tu temratura aris gradusebshi");
11  puts(" da 0  Temperatura aris farenheitebshi");
12  scanf("%d",&N);
13  printf("N= %d\n",N);
14  puts("shemoitanet Temperatura");
15  scanf("%f",&T);
16  printf("T= %f\n",T);
17  switch (N)
18 {
19 case 0:
20  puts(" N=0 temperatura shemotanil iqna farenheitebshi");
21  puts(" gadamyavs gradusebshi");
22  Tgrad=(T-32)/1.8;
23  printf("Temperatura gradusebshi = %f",Tgrad);
24  break;
25 default:
26  puts(" N=1 temperatura shemotanil iqna gradusebshi");
27  puts(" gadamyavs farenheitebshi");
28  float Tfaren=T*1.8+32;
29  printf("Temperatura farenheitebshi = %f",Tfaren);
30  }
31  return 0;
```

```
32 }
```

დავალება 2.2: ტემპერატურა

1. რა პრობლემები აქვს ზემოთ მოყვანილ კოდს?
2. შეასწორეთ ზემოთ მოყვანილი კოდი.
3. დაუმატეთ კელვინის ერთეულებში შეყვანა.
4. შეამოწმეთ უარყოფით ტემპერატურაზე ცელსიუსის და ტემპერატურის შემთხვევაში (ნაკლები არ უნდა იყოს -273-ზე).
5. დაუმატეთ შემოწმებები ფარენჰეიტის და კელვინის შემთხვევაში.

ქვემოთ მოყვანილია კვადრატული განტოლების ამოხსნა:

კოდი 2.5 კვადრატული განტოლება

```
1  #include <stdio.h>
2  #include <math.h>
3  int main(int argc, char* argv[])
4  {
5   double x1=0;
6   double x2=0;
7   double a=1;
8   double b=2;
9   double c=4;
10  double d=b*b-4*a*c;//დეტერმინანტი
11  if(d<0){
12   printf("D<0 araa amoxsna\n");
13   return 0;
14    }
15  x1=(-b+sqrt(d))/(2*a);
16  x2=(-b-sqrt(d))/(2*a);
17  printf("aris 2 fesvi:\n");
18  printf("x1=%g:\n x2=%g\n",x1,x2);
19  return 0;
20 }
```

დავალება 2.3: კვადრატული განტოლება

1. შეცვალეთ ზემოთ მოყვანილი კოდი ისე, რომ შესაძლებელი იყოს კვადრატული განტოლების კოეფიციენტების შეტანა კონსოლიდან.
2. დაწერეთ პროგრამა, რომელიც კონსოლიდან მიიღებს მთელი ტიპის დადებით რიცხვს. ეს იქნება წამები. გადაიყვანეთ მიღებული რიცხვი წუთებში, საათებში, დღეებში. დააბეჭ-

დინეთ ეკრანზე შემდეგი ფორმატით:
shemotanilia X cami.
X cami = A dge, B saati, C suti, D cami

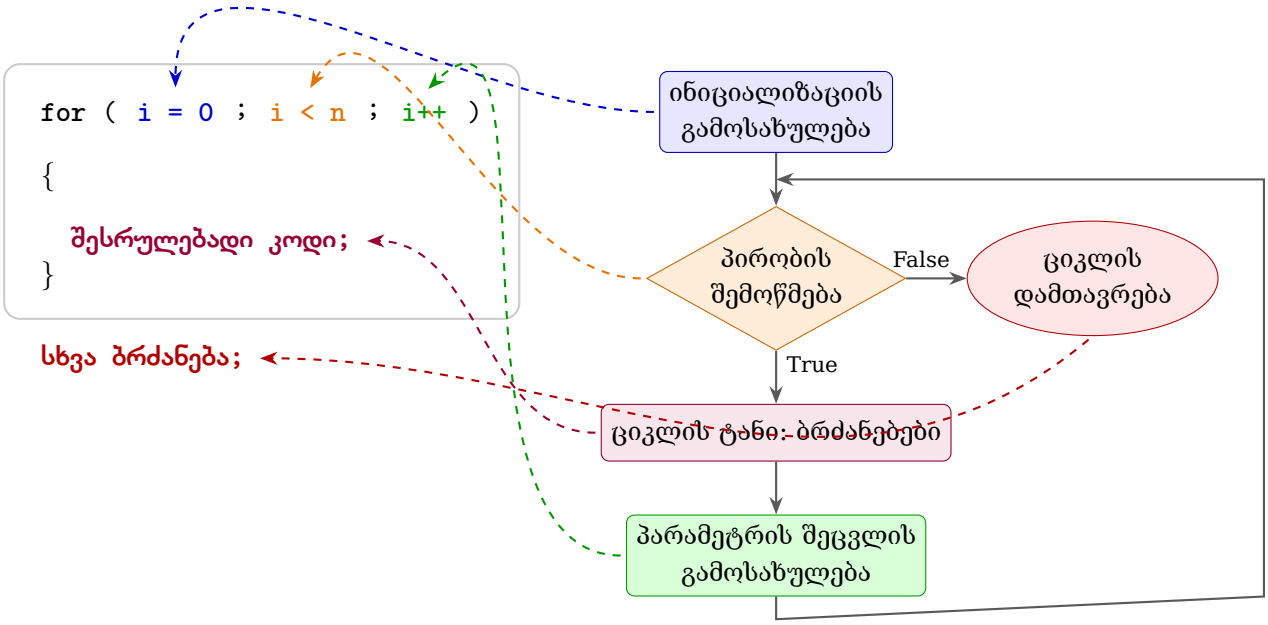
2.4 ციკლები

არის შემთხვევები, როცა ბრძანება ან ბრძანებების სიმრავლე უნდა შესრულდეს რამდენჯერმე — შეიძლება ერთი და იმავე ცვლადზე, ან რამდენიმე ცვლადზე. უფრო მეტიც, შეიძლება გვჭირდებოდეს, რომ რაიმე ოპერაციები (მაგალითად რაღაცის გამოთვლა: ტემპერატურა, კოორდინატი, ელექტრული ველი, დენი, ძაბვა...) სრულდებოდეს მანამ, სანამ არ შესრულდება რაღაც პირობა. ასეთი ამოცანებისთვის გამოიყენება ციკლები.

2.4.1 for ციკლი

for ციკლს აქვს შემდეგი სინტაქსი (იხ. სურ. 2.1):

```
for ( ინიციალიზაცია ; პირობის შემოწმება; პარამეტრის განახლება)
{
 შესრულებადი კოდი;
}
სხვა ბრძანება ;
```



სურ. 2.1 for ციკლის შესრულების სქემა და შესაბამისობა C კოდთან

- ინიციალიზაციის ბრძანება სრულდება მხოლოდ ერთხელ.
- შემდეგ ხდება პირობის შემოწმება. თუ პირობა ჭეშმარიტია, სრულდება 'შესრულებადი კოდი;'.

- შემდეგ ხდება 'პარამეტრის განახლება' ნაწილის შესრულება და კვლავ ხდება პირობის შემოწმება. თუ პირობა არ სრულდება, წყდება ციკლის შესრულება და პროგრამა შეასრულებს სხვა ბრძანება ; და შემდგომ ბრძანებებს.

ქვემოთ მოყვანილი კოდი ითვლის n რიცხვის ჯამს

კოდი 2.6 n რიცხვის ჯამი

```
#include <stdio.h>

int main(void){
 int n, count, sum=0;
 printf("shemoitanet n mnishvneloba.\n");
 scanf("%d", &n);
 for(count=1; count<=n; ++count) // for ციკლი წყდება, როცა count>n
 {
  sum += count; // ეს იგივეა, რაც sum = sum + count
 }
 printf(" %d ricxvis jami=%d", n, sum);
 return 0;
}
```

ქვემოთ მოყვანილია გამრავლების ცხრილი:

კოდი 2.7 გამრავლების ცხრილი

```
// გამრავლების ცხრილი
#include <stdio.h>

int main()
{
 int n, i;
 printf("shemoitanet raime mTeli ricxvi: ");
 scanf("%d", &n);
 for(i=1; i<=10; ++i)
 {
  printf("%d * %d = %d\n", n, i, n*i);
 }
 return 0;
}
```

მაგალითი: რიცხვის გამყოფი

ქვემოთ მოყვანილია პროგრამის კოდი, რომელიც ითვლის რიცხვის გამყოფებს:

კოდი 2.8 რიცხვის გამყოფი

```
#include <stdio.h>

int main()
{
 int n, i;
 printf("shemoitanet mteli dadebiti ricxvi: ");
 scanf("%d", &n);
 printf("%d -s gamyofebia: ", n);
 for(i=1; i<=n; ++i)
 {
  if(n%i == 0)
  printf("%d ", i);
 }
 return 0;
}
```

მაგალითი: ორი რიცხვის გამყოფი

ქვემოთ მოყვანილია პროგრამის კოდი, რომელიც ითვლის ორი რიცხვის საერთო გამყოფს:

კოდი 2.9 საერთო გამყოფი

```
#include <stdio.h>

int main()
{
 int num1, num2, min, i;
 printf("Enter two integers: ");
 scanf("%d %d", &num1, &num2);
 min = (num1 > num2) ? num2 : num1; /* min inaxeba num1 */
 /* da num2 -s shoris minimaluri mnishvneloba */
 for(i=min; i>=1; --i)
 {
  if(num1%i==0 && num2%i==0)
  {
   printf("%d da %d -s saerto gamyofia %d", num1, num2, i);
   break;
```

```
16    }
17   }
18   return 0;
19  }
```

ისევე როგორც if შემთხვევაში, ციკლების შემთხვევაშიც შეიძლება გვქონდეს ე.წ. „ჩადგმული ციკლები“:

კოდი 2.10 ორმაგი ციკლი

```
1  #include <stdio.h>
2
3  int main()
4  {
5   int i, j;
6   for(i=1; i<5; i++)
7   {
8    printf("\n");
9    for(j=i; j>0; j--)
10   {
11    printf(" %d", j);
12   }
13  }
14  return 0;
15 }
```

for ციკლში თუ არ გვაქვს ციკლის შეწყვეტის პირობა ან პირობა, რომელიც არასდროს არ შესრულდება, მაშინ ციკლი იქნება უსასრულო. იმისათვის, რომ ციკლი შეწყდეს, საჭიროა ბრძანება break.

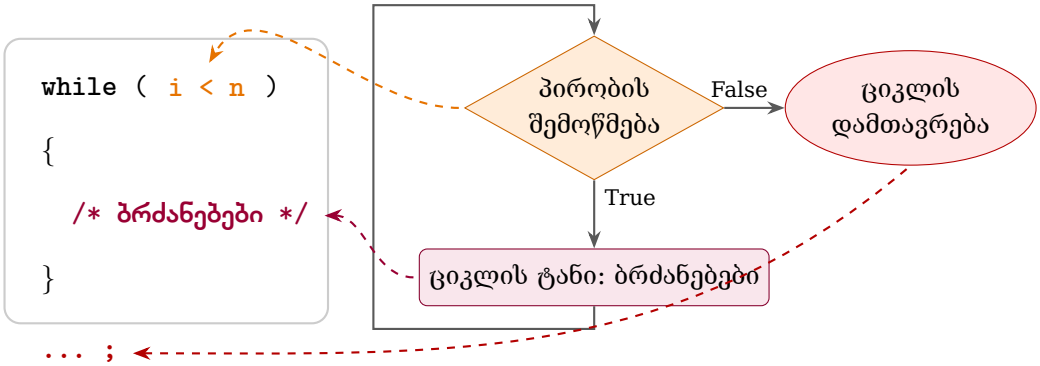
დავალება 2.4: ორმაგი for ციკლი

დაწერეთ პროგრამა, რომელიც კონსოლზე გამოიტანს გამრავლების ცხრილს $(1-9) \otimes (1-9)$

შენიშვნა: გამოიყენეთ 'printf', 'for' *ციკლი და ტაბულაციის ელემენტები* '\t', '\n'

2.4.2 while ციკლი

while ციკლს აქვს შემდეგი სინტაქსი:



სურ. 2.2 while ციკლის შესრულების სქემა და შესაბამისობა C კოდთან

ანუ სანამ 'პირობის შემოწმება' არის ჭეშმარიტი, სრულდება 'შესრულებადი კოდი'. თუ პირობა ყოველთვის ჭეშმარიტია, მაშინ გვაქვს უსასრულო while ციკლი. ციკლი შეიძლება შევწყვიტოთ break ბრძანების გამოყენებით, ისევე როგორც for ციკლის შემთხვევაში. ამ მეთოდის გარდა ასევე გამოიყენება მთვლელი ციკლის კონტროლისათვის.

განვიხილოთ მაგალითები:

```c
// Fig. 4.1: fig04_01.c
// Counter-controlled repetition
#include <stdio.h>

// function main begins program execution
int main( void )
{
 unsigned int counter = 1; // initialization

 while ( counter <= 10 ) { // repetition condition
  printf ( "%u\n", counter ); // display counter
  ++counter; // increment
 } // end while
} // end function main
```

პროგრამა მარტივია: სანამ counter$<=10$, დაბეჭდე counter ცვლადის მნიშვნელობა და შემდეგ გაზარდე ერთი ერთეულით. ამ შემთხვევაში შეგვეძლო გამოგვეყენებინა for ციკლიც, ვინაიდან ვიცით ზედა საზღვარი (ამ შემთხვევაში 10).

მაგრამ არის შემთხვევები, როცა წინასწარ არ ვიცით, როდის შეიძლება შეწყდეს ციკლი და ციკლის შეწყვეტის პირობა გამოითვლება მათემატიკურად (მაგალითად: სანამ რაღაც სიდიდის ცდომილება არის 0.001-ზე მეტი, უნდა დავითვალოთ ეს სიდიდე) ან დამოკიდებულია მომხმარებლის მიერ შეტანილ რაიმე მნიშვნელობაზე.

ქვემოთ მოყვანილ მაგალითში ციკლი წყდება, როცა მომხმარებელი აკრეფს $Ctrl-Z$. ამ მაგალითში მომხმარებლებს შეჰყავს ნიშნები (A-F), პროგრამა ითვლის და ბეჭდავს თი-

თოეული ნიშნის რაოდენობას.

კოდი 2.11 ნიშნები

```
1  // Fig. 4.7: fig04_07.c
2  // Counting letter grades with switch
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8   int grade; // one grade  ნიშანი
9   unsigned int aCount = 0; // number of As
10  unsigned int bCount = 0; // number of Bs
11  unsigned int cCount = 0; // number of Cs
12  unsigned int dCount = 0; // number of Ds
13  unsigned int fCount = 0; // number of Fs
14
15  puts( "Enter the letter grades."  );
16
17  puts( "Enter the EOF character to end input."  );
18
19  // loop until user types end-of-file key sequence
20  while ( ( grade = getchar() ) != EOF ) {
21
22   // determine which grade was input
23   switch ( grade ) { // switch nested in while
24
25    case 'A': // grade was uppercase A
26    case 'a': // or lowercase a
27    ++aCount; // increment aCount
28    break; // necessary to exit switch
29
30    case 'B': // grade was uppercase B
31    case 'b': // or lowercase b
32    ++bCount; // increment bCount
33    break; // exit switch
34
35    case 'C': // grade was uppercase C
36    case 'c': // or lowercase c
37    ++cCount; // increment cCount
```

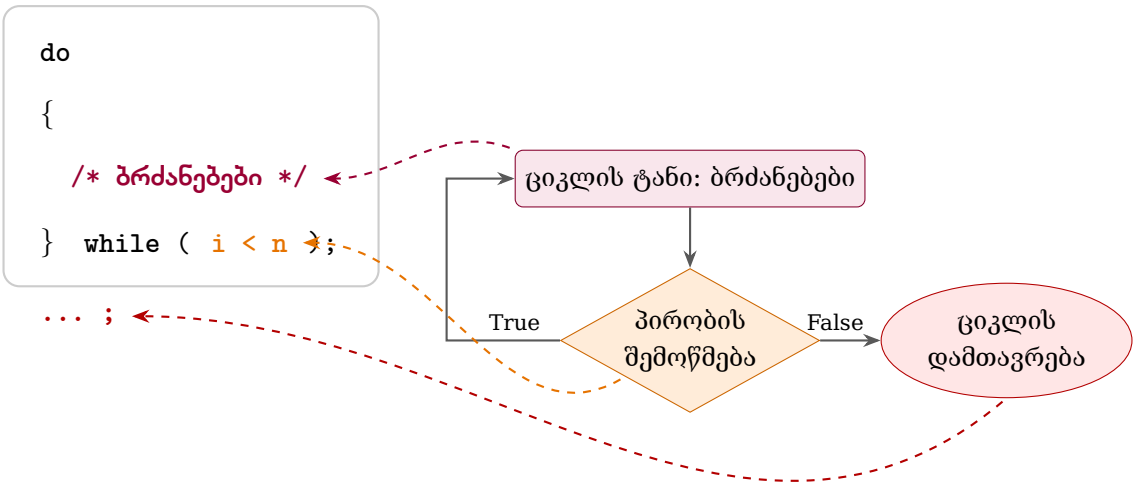
```
38     break; // exit switch
39
40     case 'D': // grade was uppercase D
41     case 'd': // or lowercase d
42     ++dCount; // increment dCount
43     break; // exit switch
44
45     case 'F': // grade was uppercase F
46     case 'f': // or lowercase f
47     ++fCount; // increment fCount
48     break; // exit switch
49
50     case '\n': // ignore newlines,
51     case '\t': // tabs,
52     case ' ': // and spaces in input
53     break; // exit switch
54
55     default: // catch all other characters
56     printf( "%s", "Incorrect letter grade entered." );
57     puts( " Enter a new grade." );
58     break; // optional; will exit switch anyway
59    } // end switch
60   } // end while
61
62   // output summary of results
63   puts( "\nTotals for each letter grade are:" );
64   printf( "A: %u\n", aCount ); // display number of A grades
65   printf( "B: %u\n", bCount ); // display number of B grades
66   printf( "C: %u\n", cCount ); // display number of C grades
67   printf( "D: %u\n", dCount ); // display number of D grades
68   printf( "F: %u\n", fCount ); // display number of F grades
69  } // end function main
```

while ციკლის მსგავსია ციკლი do..while, რომელსაც აქვს შემდეგი სინტაქსი:

როგორც ვხედავთ, აქ ერთხელ მაინც სრულდება რაღაც ოპერაციები და შემდეგ ხდება პირობის შემოწმება.

2.4.3 ბრძანება break და continue

ბრძანება break უკვე შევხვდით switch-ის განხილვისას. ეს ბრძანება ასევე ხშირად გამოიყენება ციკლებში. break ნიშნავს, რომ მოცემული კონსტრუქციის (`switch`, `for`, `while`) შესრულება წყდება და პროგრამა გადადის შემდეგი ბრძანების შესრულებაზე.



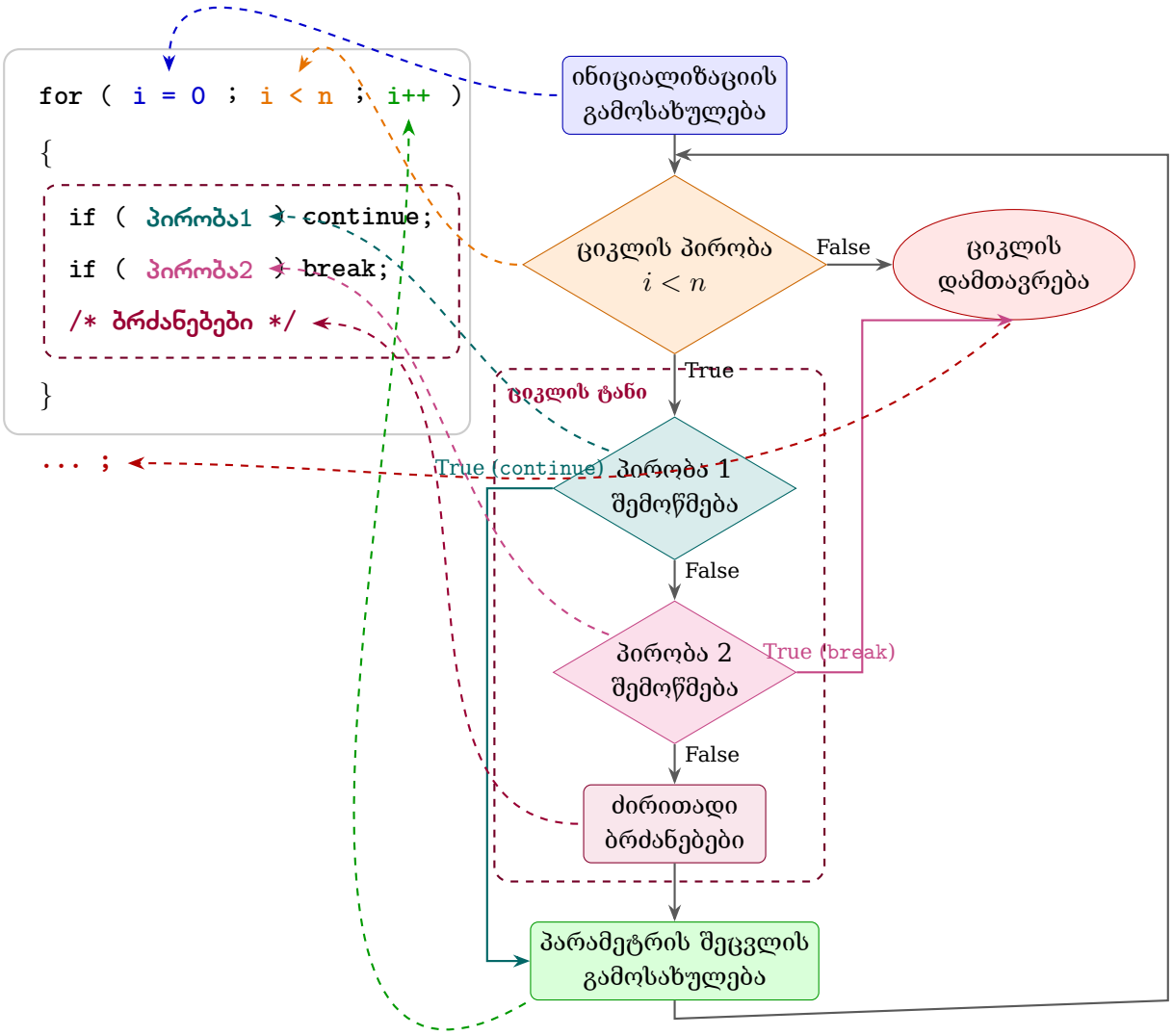
სურ. 2.3 do...while ციკლის შესრულების სქემა.

ციკლებთან მუშაობისას ასევე ხშირად გამოიყენება ბრძანება `continue`. თუ `break` ბრძანება წყვეტდა ციკლს, `continue`-ს შემთხვევაში ხდება ციკლის დასაწყისში გადასვლა.

დავალება 2.5: არაუარყოფითი რიცხვების საშუალო

გაარჩიეთ ქვემოთ მოყვანილი კოდი. კოდს აქვს პრობლემა, როცა i=1. შეასწორეთ კოდი $i = 1$ შემთხვევისათვის.

```c
/* C program to demonstrate the working of break */
/*statement by terminating a loop, if user inputs negative number*/
#include <stdio.h>
int main(){
 float num,average,sum;
 int i,n;
 printf("Maximum no. of inputs\n");
 scanf("%d",&n);
 for(i=1;i<=n;++i){
  printf("Enter n%d: ",i);
  scanf("%f",&num);
  if(num<0.0)
  break;                        //for loop breaks if num<0.0
  sum=sum+num;
 }
 average=sum/(i-1);
 printf("Average=%.2f",average);
 return 0;
}
```



სურ. 2.4 for ციკლი continue და break ოპერატორებით

2.5 ჩამოთვლილი (დანომრილი) ტიპი enum

ვთქვათ ამოცანა არის ასეთი: გვჭირდება ისეთი მონაცემთა ტიპი, რომლის მოცემის შემდეგ მოცემული ტიპის ცვლადმა შეიძლება მიიღოს მხოლოდ რაღაც წინასწარ ჩვენთვის საინტერესო მნიშვნელობები. მაგალითად გვინდა, რომ გვქონდეს ცვლადი myColor, რომელმაც შეიძლება მიიღოს მხოლოდ შემდეგი მნიშვნელობები: red, green, blue. ასეთი რამ შესაძლებელია ჩამოთვლილი ტიპის (ან დანომრილი ტიპის - enumerated type) შემოტანით. ამ ტიპის სპეციალური სიტყვა ენაში არის enum. ზემოთ მოყვანილი მაგალითისთვის შეგვიძლია შემოვიტანოთ შემდეგი ჩამოთვლილი ტიპი:

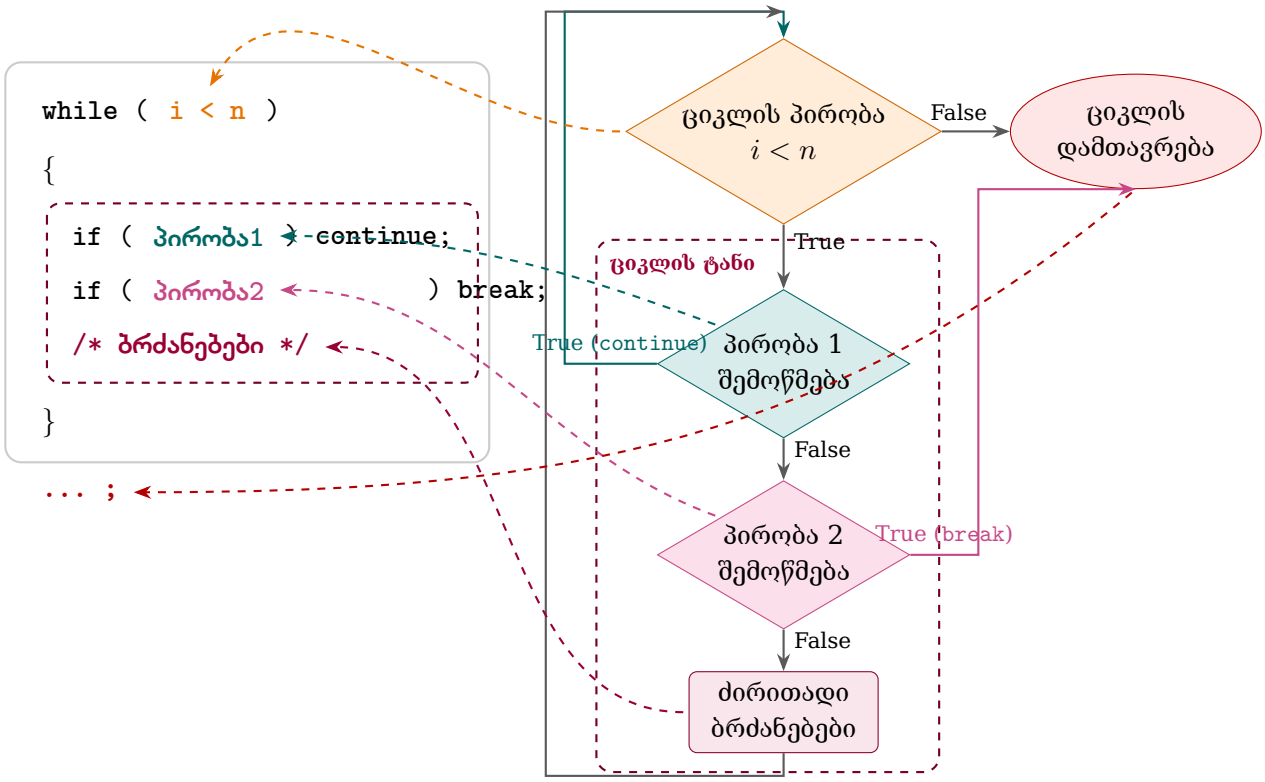
```
enum MainColors {red, green, blue};
```

ეს ბრძანება განმარტავს ტიპს MainColors და ამ ტიპის ცვლადმა შეიძლება მიიღოს მხოლოდ მნიშვნელობები red, green ან blue. როცა გვსურს ცვლადის შემოტანა, ისევ ვიყენებთ სიტყვებს enum MainColors და ცვლადების სახელებს. მაგალითად:

```
enum MainColors color1, color2, myColor;
```

ხოლო პროგრამაში შეგვიძლია გამოვიყენოთ, როგორც:

```
color1=green; color2=red; myColor=color2;
```



სურ. 2.5 while ციკლი continue და break ოპერატორებით

ამ შემთხვევაში ჩვენ შემოვიყვანეთ MainColors ტიპის სამი ცვლადი.

ჩამონათვალი ტიპის გამოყენება ბუნებრივია, როცა ვმუშაობთ თარიღებთან. მაგალითად:

```
enum Month{Jan,Feb,Mar,Apr,May,Jun...};
enum WeekDay {Sun,Mon,Wed,Thr...};[1]
enum Direction {Left,Right,Up,Down};
```

როგორ მუშაობს ასეთი ცვლადი? კომპილატორი ჩამონათვალის ტიპის განმარტებაში არსებულ იდენტიფიკატორებს (Jan,Feb,Mar...) აღიქვამს, როგორც მთელი ტიპის რიცხვებს, რომელთაგან პირველის მნიშვნელობაა 0, მეორისა – 1, მესამისა – 2 და ა.შ. მაგალითად თუ კოდში გვაქვს:

```
enum WeekDay day;
day=Mon;
```

ცვლად day-ს მნიშვნელობა იქნება 1;

თუ გვსურს, რომ მნიშვნელობები იყოს რაღაც სხვა, შეგვიძლია გავაკეთოთ შემდეგი:

```
enum Direction {Left,Right,Up=11,Down};
```

ამ შემთხვევაში: Left=0, Right=1, Up=11, Down=12.

განვიხილოთ ქვემოთ მოყვანილი კოდი:

კოდი 2.12 თვეები

[1] ცხადია '...' მხოლოდ სიმოკლისთვისაა და არა ცვლადის სახელი.

```c
#include <stdio.h>
int main (void)
{
 enum month { january = 1, february, march, april, may, june,
  july, august, september, october, november, december };
 enum month aMonth; int days;
 printf ("Enter month number:  ");
 scanf ("%i", &aMonth);
 switch (aMonth ) {
  case january:
  case march:
  case may:
  case july:
  case august:
  case october:
  case december:
  days = 31;
  break;
  case april:
  case june:
  case september:
  case november:
  days = 30;
  break;
  case february:
  days = 28;
  break;
  default:
  printf ("bad month number\n");
  days = 0;
  break;
 }
 if ( days != 0 ) printf ("Number of days is %i\n", days);
 if ( aMonth == february) printf ("...or 29 if its a leap year\n");
 return 1;
}
```

მე-4 ხაზზე შემოგვყავს ტიპი month, ხოლო მე-6 ხაზზე ცხადდება month ტიპის ცვლადი aMonth. month ტიპის ცვლადის განმარტებისას პირველი ელემენტი january ინიციალიზდება როგორც 1 (january = 1), ამიტომ შემდეგი ელემენტის მნიშვნელობა იქნება 2 და ა.შ.

მე-6 ხაზზე ხდება aMonth ცვლადის წაკითხვა. იმის მიხედვით, თუ რა რიცხვი იქნა შეტანილი მომხმარებლის მიერ, ხდება თვეში არსებული დღეების რაოდენობის ბეჭდვა.

დავალება 2.6: ნაკიანი წელი

შეასწორეთ ზემოთ მოყვანილი კოდი ისე, რომ:

1. შესაძლებელი იყოს წლის შეყვანა. თუ წელი ნაკიანია, დაიბეჭდოს თებერვალში დღეების რაოდენობა 29.
2. შეცვალეთ კოდი ისე, რომ პროგრამა ამთავრებდეს მუშაობას, როცა მომხმარებელი შეიყვანს '-1'.

enum ტიპის ცვლადებს შეიძლება ჰქონდეთ ერთი და იგივე მნიშვნელობაც:

```
enum  SWITCH  { no=0, off=0, yes=1, on=1 };
```

შეიძლება ასევე შემოტანილ იქნას ჩამოწათვალი ტიპის ცვლადი ისე, რომ არ იყოს მითითებული ცვლადის ტიპი:

```
enum { east, west, south, north }  direction;
```

თავი 3

მნიშვნელობა, მისამართი, მითითება, ფუნქციები

3.1 მნიშვნელობა და მისამართი 45
3.2 მიმთითებელი 46
3.2.1 NULL მიმთითებელი 47
3.3 პრეპროცესორის დირექტივები 48
3.3.1 დირექტივა #define 49
3.3.2 არგუმენტიანი #define მაკროები და მათი საფრთხეები 50
3.4 ფუნქციები 51
3.4.1 const მოდიფიკატორი მიმთითებლთან. 54
3.5 მონაცემების გადაცემა მიმთითებლით 55
3.6 მიმთითებელი ფუნქციაზე (Function Pointer). 56
3.6.1 ფუნქციის მიმთითებლის გამოცხადების სინტაქსი 56
3.6.2 ფუნქციაში ფუნქციის მიმთითებლის გადაცემა და გამოძახება. . . . 57
3.7 დეკლარაცია/იმპლემენტაცია სხვადასხვა ფაილებში. 58
3.8 extern "C" 62
3.9 C პროექტის შექმნა 63

3.1 მნიშვნელობა და მისამართი

მისამართის აღების ოპერატორია "&". ეს ჩვენ უკვე შეგვხვდა scanf ფუნქციაში. scanf ფუნქციას გადაეცემა ცვლადის მისამართი, რომელსაც უნდა მიენიჭოს წაკითხული მნიშვნელობა.

განვიხილოთ მაგალითი:

კოდი 3.1 მნიშვნელობა და მისამართი

```
1 #include<stdio.h>
2 int main(int argc, char* argv[])
3 {
4  double x = 1;
5 
6  printf("x mnishvneloba cakitxvamde = %lf\n", x);
7  printf("x misamarti (teqvsmetobitshi) = %p\n", (void*)&x);
8 
```

```
 scanf("%lf", &x);

 printf("x axali mnishvneloba = %lf\n", x);
 printf("x misamarti (teqvsmetobitshi) = %p\n", (void*)&x);

 return 0;
}
```

პროგრამის 3.1 მნიშვნელობა და მისამართი გაშვების შემდეგ ნახავთ, რომ რა ახალი მნიშვნელობაც არ უნდა მივანიჭოთ x-ს, მისი მისამართი იგივე იქნება. ცხადია, ყოველ ახალ გაშვებაზე მისამართი დიდი ალბათობით იქნება სხვა, ვინაიდან ალოკაცია (ცვლადის შექმნა მეხსიერებაში) ხდება პირველსავე შემთხვევით ადგილას თავისუფალ მეხსიერებაში (ავტომატური ცვლადებისთვის, როგორიცაა ზემოთ მოყვანილი ცვლადი — ე.წ. სტეკ მეხსიერებაში) და წინასწარ ვერ ვიტყვით, სად მოხდება ეს. მაგრამ ერთი რამ შეიძლება დარწმუნებით ვთქვათ: მას შემდეგ, რაც შეიქმნება ცვლადი მეხსიერებაში, მისი მისამართი რჩება მუდმივი ამ ცვლადის არსებობის განმავლობაში (ცვლადების დინამიურად შექმნას და „მოკვლას“ შემდეგში განვიხილავთ. ჯერჯერობით საუბარია ე.წ. ავტომატურად შექმნილ ცვლადებზე; სტატიკური ტიპის ცვლადი სხვაა, ამასაც მოგვიანებით განვიხილავთ).

3.2 მიმთითებელი

მიმთითებელი არის ობიექტი, რომელიც უთითებს რაიმე ობიექტზე. მიმთითებლიდან შეიძლება „ავიღოთ“ ცვლადის მნიშვნელობა, რომელზეც ის უთითებს, ასევე მისამართი იმ ცვლადზე, რომელზეც ის უთითებს. მისამართისგან განსხვავება ისაა, რომ ერთი მიმთითებელი სხვადასხვა შემთხვევაში შეიძლება უთითებდეს სხვადასხვა ცვლადზე (რომელთა მისამართი, ცხადია, სხვადასხვაა), ანუ მიმთითებლიდან აღებული მისამართი შეიძლება იყოს სხვადასხვა, მაშინ როცა მოცემული ცვლად-ობიექტის მისამართი უცვლელია.

განვიხილოთ მაგალითი:

კოდი 3.2 მიმთითებელი

```
#include<stdio.h>
int main(int argc, char* argv[])
{
 double x = 1; // cvladebis gamocxadeba da sheqmna inicializacia
 double y = 1;
 double *px = NULL; // mimtiteblis gamocxadeba da inicializacia
 printf("x mnishvneloba = %lf\n", x);
 printf("x misamarti = %p\n", (void*)&x);
 scanf("%lf", &x); // vkitxulobt x cvladis mnisvnelobas
 printf("x axali mnishvneloba = %lf\n", x);
 printf("x misamarti = %p\n", (void*)&x);
```

```
12  px = &x; // mimtitebeli utitebs x cvladze
13  printf("mnishv. romelzec is utitebs px = %lf\n", (*px));
14  printf("misam. romelzec is utitebs px = %p\n", (void*)px);
15  printf("y mnishvneloba = %lf\n", y);
16  printf("y misamarti = %p\n", (void*)&y);
17  px = &y; // igive mimtitebeli exla utitebs y cvldze
18  printf("mnishvneloba romelzec utitebs px = %lf\n", (*px));
19  printf("misam. romelzec utitebs px = %p\n", (void*)px);
20  return 0;
21 }
```

$(*px)$ იძლევა ცვლადის მნიშვნელობას, რომელზეც უთითებს px. $\&(*px)$ (ან უბრალოდ px) იძლევა ცვლადის მისამართს, რომელზეც უთითებს px. ცხადია, გამოცხადებული მიმთითებელი უნდა იყოს იგივე ტიპის, რა ტიპზეც ის უთითებს[1].

ინიციალიზაციის გარეშე გამოცხადებული მიმთითებელი (ე.წ. dangling/wild pointer) არაფერზე განსაზღვრულზე არ უთითებს და ასეთ შემთხვევაში მისამართის ან მნიშვნელობის წაკითხვის/შეცვლის მცდელობა გამოიწვევს არაგამოსახულ (Undefined Behavior) ქცევას ან პროგრამის ავარიულ დასრულებას (Segmentation Fault).

3.2.1 NULL მიმთითებელი

ხშირად მიმთითებლის გამოცხადებისას ინიციალიზაციას უკეთებენ ე.წ. NULL-ზე. ზოგ სისტემაში NULL განმარტებულია როგორც 0, მაგრამ ეს არ ნიშნავს, რომ მიმთითებლის ან ცვლადის (რომელსაც ის უთითებს) მნიშვნელობაა ნული. ესაა სპეციალური ე.წ. ნულ-მიმთითებელი და საჭიროა იმ შემთხვევებში, როცა გვინდა შევამოწმოთ, უთითებს თუ არა მიმთითებელი რაიმეზე:

```
1  #include<stdio.h>
2  int main(void)
3  {
4   int *p = NULL;
5   if (p == NULL) {
6    printf("p is uninitialized!\n");
7   } else {
8    printf("p points to %d\n", *p);
9   }
10  return 0;
11 }
```

აღსანიშნავია, რომ მიმთითებლის შემოტანისას მიმთითებელი ავტომატურად არ ინიცი-

[1] არის ასევე void ტიპის მიმთითებელიც, რომელიც შეიძლება უთითებდეს ნებისმიერი ტიპის ცვლადზე. მას მოგვიანებით განვიხილავთ.

ალიზდება როგორც NULL.

> C23 სტანდარტში NULL-ის ნაცვლად დაემატა ენის ოფიციალური ტიპიზირებული ნულ-მიმთითებელი nullptr.

ცხადია, შესაძლებელია, რომ ერთი და იგივე მიმთითებელი გამოყენებულ იქნას სხვადასხვა ცვლადზე მისათითებლად:

კოდი 3.3 ერთი მიმთითებლის მრავალჯერადი გამოყენება

```
#include<stdio.h>
int main(int argc, char* argv[])
{
 double x = 1; // cvladebis gamocxadeba
 double y = 1; // da sheqmna inicializacia
 double *px = NULL; // mimtiteblis gamocxadeba
 if(px == NULL)
  printf("px araferze ar utitebs\n");

 px = &x; // mimtitebeli utitebs x cvladze
 if(px != NULL)
 printf("px araa NULL e.i. utitebs \n");
 printf("mnishv. romelzec is utitebs px = %lf\n", (*px));
 printf("misam. romelzec is utitebs px = %p\n", (void*)px);
 printf("y mnishvneloba = %lf\n", y);
 printf("y misamarti = %p\n", (void*)&y);

 px = &y; // igive mimtitebeli exla utitebs y cvldze
 if(px != NULL)
 printf("px araa NULL e.i. utitebs \n");
 printf("mnishvneloba romelzec utitebs px = %lf\n", (*px));
 printf("misam. romelzec utitebs px = %p\n", (void*)px);
 return 0;
}
```

3.3 პრეპროცესორის დირექტივები

ჩვენ შეგვხვდა გამოსახულება #include <stdio.h>. ეს არის ბრძანება კომპილატორისათვის, ჩართოს კოდში ფაილი stdio.h და ეს ფაილი უნდა მოძებნოს სისტემურ დირექტორიებში (ამას აღნიშნავს < >).

ხშირად დაგვჭირდება შემდეგი: #include "filename.h". ესეც ნიშნავს, რომ კომპილაციისას ჩართული უნდა იყოს filename.h, მაგრამ ამ შემთხვევაში ამ ფაილის ძებნა წარმოებს მიმდინარე (ანუ სადაც წერია ფაილი, რომელშიც ხდება ჩართვა) დირექტორიაში.

ჩართვის ბრძანებას შეიძლება ჰქონდეს სხვა სახეც: `#include "../filename.h"` ან `#include "dirname/filename.h"`. პირველი ნიშნავს, რომ ფაილი აღებულ უნდა იქნას ერთი დირექტორიით მაღლა მდებარე დირექტორიაში, მეორე კი — მიმდინარე დირექტორიაში არსებული dirname დირექტორიიდან.

3.3.1 დირექტივა #define

დირექტივა #define და სხვა დირექტივები, როგორც წესი, იწერება ფაილის დასაწყისში #include დირექტივების შემდეგ. დირექტივა #define განსაზღვრავს სიმბოლურ მუდმივას. მაგალითად:

```
#define R 100
#define PI 3.14159
```

აქ შემოტანილია ორი რიცხვი (და არა ცვლადი) და კომპილაციის დროს ეს რიცხვები „ჩაჯდება“ იმ სიმბოლოების ადგილას, რომლებიც განმარტებულია ამ რიცხვებით. ზოგადად, # -ით განმარტებული ბრძანებები სრულდება კოდის კომპილაციისას. ზემოთ მოტანილი მაგალითებისათვის PI ცვლადის შექმნა კი არ ხდება, არამედ კომპილაციისას, სადაც წერია PI, გამოყენებულ იქნება მისი მნიშვნელობა.

#define დირექტივა ხშირად გამოიყენება ე.წ. საჭირო ბიბლიოთეკების ჩასართავად, ასევე სხვადასხვა ტიპების განსამარტავად, შესამოწმებლად, თუ რა ოპერაციული სისტემა ან რა კომპილატორია გამოყენებული და ა.შ. განვიხილოთ ქვემოთ მოყვანილი კოდი:

```
1  #include <stdio.h>
2  #define PI 3.141592653589793
3  int main (void)
4  {
5   double radius=0.0;
6   printf("\n %s\n\n%s ",
7   "creciris fartobis gamotvla.",
8   "shemoitanet radiusi:  ");
9   scanf( "%lf", &radius);
10  printf("\n %s\n %s %.2f %s %.2f %s %.2f\n %s %.5f\n\n",
11  "fartobi = PI * radius * radius",
12  "        = ",PI,"*",radius,"*",radius,
13  "        = ",PI*radius*radius);
14  return 0;
15 }
```

ეკრანზე გამოვა შემდეგი:

```
c:\users\username>   creciris fartobis gamotvla.
c:\users\username>
c:\users\username>   shemoitanet radiusi:   2
c:\users\username>
c:\users\username>   fartobi = PI * radius * radius
c:\users\username>   =  3.14 * 2.00 * 2.00
c:\users\username>   =  12.56637
c:\users\username>   Hit ENTER to continue...
```

მაგრამ კალკულატორით თუ გამოვთვლით იგივეს, პასუხი იქნება 12.5663706144 . ეს იმიტომ, რომ პროგრამაში გამოყენებულია დამრგვალება — შეტანისას მძიმის შემდეგ ორი ციფრით %.2f , ხოლო გამოტანისას — მძიმის შემდეგ ხუთი ციფრით %.5f . თუ შევიტანთ რადიუსის სხვა მნიშვნელობას, მაგ. "3.119455" , განსხვავება კიდევ უფრო დიდი იქნება.

3.3.2 არგუმენტიანი #define მაკროები და მათი საფრთხეები

პრეპროცესორის მაკროებს არ გააჩნიათ ტიპების შემოწმება და ტექსტურად ენაცვლებიან კოდს კომპილაციამდე, რამაც შეიძლება მოულოდნელი შეცდომები გამოიწვიოს:

კოდი 3.4 მაკროს გვერდითი ეფექტები

```c
#include <stdio.h>

// araswori makro:
#define SQUARE(x) x * x

// shedarebit usaprtxo, magram mainc problematici:
#define SQUARE_SAFE(x) ((x) * (x))

int main()
{
 int a = 3;

 // SQUARE(a + 1) gaishleba rogorc: a + 1 * a + 1
 // shedegi: 3 + 1*3 + 1 = 7 (da ara 16!)
 printf("SQUARE(a + 1) = %d\n", SQUARE(a + 1));

 // SQUARE_SAFE(a + 1) gaishleba rogorc: ((a + 1) * (a + 1)) = 16
 printf("SQUARE_SAFE(a + 1) = %d\n", SQUARE_SAFE(a + 1));

 // magram inkrementisas: SQUARE_SAFE(a++) gaishleba rogorc ((a++) * (a++))
```

```
21  // es gamoiwvevs gaurkvevel (Undefined) qcevas!
22  return 0;
23 }
```

3.4 ფუნქციები

ჩვენ უკვე შეგვხვდა რამდენიმე ფუნქცია: `main`, `scanf`, `printf`, კვადრატული ფესვის ამოღების ფუნქცია. ესენია ე.წ. სისტემური ფუნქცია (`main`) და დანარჩენი C ბიბლიოთეკების ფუნქციები. ცხადია, მომხმარებელს შეუძლია გამოიყენოს სხვადასხვა ბიბლიოთეკები და ფუნქციები.

ისევე, როგორც სხვა პროგრამულ ენებში, C/C++-ში შეიძლება განმარტებულ იქნას ფუნქციები. ფუნქცია შეიძლება იღებდეს რაღაც ცვლადებს როგორც არგუმენტებს და შეიძლება აბრუნებდეს "რაღაცას". ეს "რაღაც" შეიძლება იყოს არაფერი, შეიძლება იყოს ენაში არსებული ტიპის ცვლადი ან მომხმარებლის მიერ შემოტანილი შედგენილი ტიპის ცვლადის მნიშვნელობა, ან ცვლადის მისამართი/მითითება ცვლადზე.

გავიხსენოთ კვადრატული განტოლების კოდი 2.5 გვ. 31 და შევცვალოთ იგი ფუნქციის გამოყენებით.

შემოვიტანოთ ფუნქცია:

```
int solve_sqeq(double _a, double _b, double _c);
```

რომელიც ხსნის კვადრატულ განტოლებას და პროგრამაში ხდება ამ ფუნქციის გამოძახება. ფუნქცია აბრუნებს მთელ მნიშვნელობას (თუ არაფერს არ აბრუნებს, მაშინ გამოვიყენებდით `void`-ს; რატომ ავიღეთ `int`, ამას მოგვიანებით ვნახავთ) და იღებს სამ ცვლადს (კვადრატული განტოლების კოეფიციენტებს) არგუმენტებად.

მიაქციეთ ყურადღება, რომ ფუნქცია გამოცხადებული და აღწერილია ე.წ. `main` ფუნქციამდე. ჩვენ შეგვეძლო გაგვეკეთებინა სხვაგვარადაც, კერძოდ `main` ფუნქციის წინ გაგვეკეთებინა გამოცხადება შემდეგი სახით[1]:

```
int solve_sqeq(double _a, double _b, double _c);
```

ამის შემდეგ `main` ფუნქცია — როგორც ტექსტშია, და `main` ფუნქციის შემდეგ:

```
int solve_sqeq(double _a, double _b, double _c) [2]
```

ფუნქცია — როგორც ზემოთ ტექსტშია.

კოდი 3.5 კვადრატული განტოლება

```
1  #include <stdio.h>
2  #include <math.h>
3  int solve_sqeq(double _a, double _b, double _c);
4
```

[1] ყურადღება მიაქციეთ ';' სახელის ბოლოს
[2] აქ კი ';' სახელის ბოლოს არაა საჭირო

```
5  int main(int argc, char* argv[])
6  {
7   double a=1;
8   double b=2;
9   double c=4;
10  puts("shemoitanet kv. gant. koeficientebi: a, b, c");
11  scanf("%lf,%lf,%lf",&a,&b,&c);
12  solve_sqeq(a,b,c);
13  return 0;
14 }
15 int solve_sqeq(double _a, double _b, double _c)
16 {
17  double d=_b*_b-4*_a*_c;
18  if(d<0){
19   printf("D<0 araa amoxsna\n");
20   return 0;
21  }
22  double x1=(-_b+sqrt(d))/(2*_a);
23  double x2=(-_b-sqrt(d))/(2*_a);
24  printf("aris 2 fesvi:\n");
25  printf("x1=%g:\n x2=%g\n",x1,x2);
26  return 1;
27 }
```

აქ მთავარი ისაა, რომ `main` ფუნქციამ შესრულებაზე გაშვებისას უნდა იცოდეს ის ფუნქციები (უფრო სწორედ — სახელი, დაბრუნების ტიპი და მისაწოდებელი ცვლადები), რომლებსაც ის იძახებს.

ეს ახალი კოდი აკეთებს ზუსტად იგივეს, რასაც თავდაპირველი პროგრამა. თითქოს ეს კოდი უკეთესია, მაგრამ რჩება პრობლემები: რა ვქნათ, თუ კვადრატული განტოლების ამოხსნა გვჭირდება რამდენიმე ადგილას და ამას გარდა გვჭირდება ამონახსენებიც? როგორ გავიგოთ პროგრამის ფარგლებში, ერთი ამონახსენი აქვს განტოლებას, ორი თუ არცერთი?

მეორე კითხვაზე პასუხი ადვილია: რადგან ფუნქცია დავწერეთ ისე, რომ აბრუნებს მთელ ტიპს, შეგვიძლია დავაბრუნებინოთ ინფორმაცია ამონახსენების შესახებ. თუ აქვს ერთი ამონახსენი, დააბრუნოს 1, თუ ორი — 2, თუ არა აქვს ამონახსენი — 0.

რაც შეეხება ინფორმაციას ამონახსენების მნიშვნელობის შესახებ, ჩვენ გვჭირდება ორი ცვლადი, სადაც შეინახება პასუხები და შემდეგ ეს ცვლადები შეიძლება გამოყენებული იქნას პროგრამაში. თითქოს ქვემოთ მოყვანილი კოდი უნდა აკეთებდეს იმას, რაც გვსურს:

კოდი 3.6 ფუნქცია არგუმენტის გადაცემა მნიშვნელობით

```
#include <stdio.h>
#include <math.h>
int solve_sqeq(double _a, double _b,
double _c, double _x1, double _x2);

int main(int argc, char* argv[])
{
 double a=1;
 double b=2;
 double c=4;
 double x1=0;
 double x2=0;
 int result=-1;
 puts("shemoitanet kv. gant. koeficientebi: a, b, c");
 scanf("%lf%lf%lf",&a,&b,&c);
 result=solve_sqeq(a,b,c, x1,x2);
 return 0;
}
int solve_sqeq(double _a, double _b,
double _c, double _x1, double _x2)
{
 double d=_b*_b-4*_a*_c;
 if(d<0){
  printf("D<0 araa amoxsna\n");
  return 0;
 }
 else if(d==0)
 {
  printf("D=0 aqvs 1 amonaxseni\n");
  _x1=-_b/(2*_a);
  _x2=_x1;
  printf("x1=x2=%g\n",_x1);
  return 1;
 }
 printf("aris 2 fesvi:\n");
 _x1=(-_b-sqrt(d))/(2*_a);
 _x2=(-_b+sqrt(d))/(2*_a);
 printf("x1=%g:\n x2=%g\n",_x1,_x2);
 return 2;
```

```
40 }
```

მაგრამ არ იმუშავებს.

ფუნქცია ამოხსნის კვადრატულ განტოლებას, მაგრამ ამონახსენები არ იქნება ხელმისაწვდომი ფუნქციის გარეთ. ასე იმიტომ მოხდა, რომ ფუნქციას მიეწოდება არა ცვლადები, სადაც უნდა ჩაიწეროს მიღებული მნიშვნელობები, არამედ თავად ცვლადების მნიშვნელობები. ამ შემთხვევაში ამბობენ, რომ ადგილი აქვს არგუმენტის გადაცემას მნიშვნელობით. და ჩვენს შემთხვევაში x1, x2-ის გადაცემას არავითარი აზრი არ აქვს, ვინაიდან ფუნქცია აკეთებს ამ ცვლადების ასლებს და ამ ცვლადების მნიშვნელობებს იღებენ ასლები, თავად x1, x2 ფუნქციის შიგნით მიუწვდომელია. შესაბამისად, მიუწვდომელია მიღებული ამონახსნები ფუნქციის გარეთ.

3.4.1 const მოდიფიკატორი მიმთითებლთან

იმისათვის, რომ შემთხვევით არ მოხდეს ცვლადის მნიშვნელობის არასასურველი შეცვლა (მაგალითად ფუნქციის შიგნით), არგუმენტებში შეიძლება გამოყენებულ იქნას const. ასევე, კომპილატორი ცდილობს ოპტიმიზაცია გაუკეთოს კოდს და const-ის გამოყენებით ხშირ შემთხვევაში კოდი იქნება ოპტიმიზებული (ანუ შესრულდება სწრაფად; const ტიპის ცვლადები, ისევე როგორც ბრძანებები, ინახება 'მხოლოდ წაკითხვად' (read-only) მეხსიერებაში).

როდესაც const საკვანძო სიტყვას ვიყენებთ მიმთითებელთან, მნიშვნელოვანია ვარსკვლავის (*) მდებარეობა:

კოდი 3.7 const მიმთითებლებსა და მითითებებთან

```
1  #include<stdio.h>
2  int main()
3  {
4   int x = 10, y = 20;
5
6   // 1. მიმთითებელი კონსტანტურ მნიშვნელობაზე (Pointer to const)
7   const int *p1 = &x;
8   // *p1 = 15; // shecdoma! x-is mnishvnelobas ver shevcvlit p1-it.
9   p1 = &y;      // sworia! p1-s shegviddlia sxva misamarti mvanichot.
10  // 2. კონსტანტური მიმთითებელი (Const pointer)
11  int *const p2 = &x;
12  *p2 = 15;     // sworia! x-is mnishvneloba sheicvleba (gaxdeba 15).
13  // p2 = &y;  // shecdoma! p2-is misamartis shecvla ar sheidzleba.
14
15  return 0;
16 }
```

3.5 მონაცემების გადაცემა მიმთითებლით

მონაცემები შეიძლება გადაცემულ იქნას მიმთითებლის საშუალებით. ქვემოთ მოყვანილი კოდში, გარდა იმისა, რომ გამოყენებულია მიმთითებლები, წინა კოდისგან განსხვავებით ფუნქცია დეკლარირებულია, შემდეგ არის ე.წ. main ფუნქცია, ხოლო შემდეგ აღწერილია დეკლარირებული ფუნქცია.

კოდი 3.8 არგუმენტის გადაცემა მიმთითებლით (C)

```c
#include <stdio.h>
#include <math.h>
#define EPS 0.000001
int solve_sqeq(const double*_a, const double* _b,
const double* _c, double* _x1, double* _x2);
int main(int argc, char* argv[])
{
 double a=1;
 double b=2;
 double c=4;
 double x1=0;
 double x2=0;
 int result=-1;
 puts("shemoitanet kv. gant. koeficientebi: a, b, c");
 scanf("%lf%lf%lf",&a,&b,&c);
 result=solve_sqeq(&a,&b,&c,&x1,&x2);
 return 0;
}
int solve_sqeq(const double*_a, const double* _b,
const double* _c, double* _x1, double* _x2)
{
 double d=(*_b)*(*_b)-4*(*_a)*(*_c);
 if(d<0){
  printf("D<0 araa amoxsna\n");
  return 0;
 }
 else if(d<EPS)
 {
  printf("D=0 aqvs 1 amonaxseni\n");
  (*_x1)=-(*_b)/(2*(*_a));
  *_x2=*_x1;
  printf("x1=x2=%g\n",(*_x1));
  return 1;
```

```
34   }
35   printf("aris 2 fesvi:\n");
36   (*_x1)=(-(*_b)-sqrt(d))/(2*(*_a));
37   (*_x2)=(-(*_b)+sqrt(d))/(2*(*_a));
38   printf("x1=%g:\n x2=%g\n",(*_x1),(*_x2));
39   return 2;
40  }
```

ზემოთ მოყვანილი კოდი მუშაობს ისე, როგორც საჭიროა:

ა) ითვლის კვადრატული განტოლების ფესვებს x_1, x_2.

ბ) x_1, x_2 გამოყენება შეიძლება ფუნქციის გარეთ.

გ) result ცვლადში გვაქვს ინფორმაცია, თუ რამდენი ფესვი აქვს განტოლებას.

დ) a, b, c ცვლადები დაცულია ფუნქციის შიგნით შემთხვევითი ცვლილებისაგან.

მონაცემების გადაცემა მნიშვნელობით (კოდი 3.6) და მონაცემების გადაცემა მიმთითებლით (კოდი 3.8) მუშაობს C-ში, ხოლო C++-ში ასევე გვაქვს გადაცემა მისამართით (Reference), რაც არ გვაქვს C-ში.

C-ში გვაქვს მონაცემთა გადაცემა მნიშვნელობით. ერთ შემთხვევაში გადაეცემა ცვლადის მნიშვნელობა (მაგალითად, რაიმე რიცხვი), ხოლო მეორე შემთხვევაში გადაეცემა მიმთითებლის მნიშვნელობა (რომელიც არის სწორედ მისამართი და შემდეგ აიღება იმ ცვლადის მნიშვნელობა, რომელზეც უთითებს მიმთითებელი). მონაცემების ასეთი სახით გადაცემას უწოდებენ C სტილით მონაცემების გადაცემას მისამართით.

C++-ში კი მისამართით გადაცემისას ფუნქციის შიგნით მიეწოდება მისამართი (Reference).

3.6 მიმთითებელი ფუნქციაზე (Function Pointer)

ისევე როგორც ცვლადები ინახება მეხსიერებაში და გააჩნიათ თავისი მისამართი, ფუნქციის კოდიც ინახება ოპერატიულ მეხსიერებაში (კოდის სეგმენტში). შესაბამისად, შესაძლებელია შევქმნათ მიმთითებელი, რომელიც უთითებს არა მონაცემზე (ცვლადზე), არამედ უშუალოდ ფუნქციის საწყის მისამართზე.

3.6.1 ფუნქციის მიმთითებლის გამოცხადების სინტაქსი

ფუნქციის მიმთითებლის გამოცხადებისას აუცილებელია მივუთითოთ:

1. რა ტიპის მნიშვნელობას აბრუნებს ფუნქცია,
2. მიმთითებლის სახელი (ფრჩხილებში ვარსკვლავთან ერთად: `(*ptr_name)`),
3. ფუნქციის პარამეტრების ტიპების ჩამონათვალი.

მაგალითად:

```
double (*func_ptr)(double , double);
```

ყურადღება: ფრჩხილები (`*func_ptr`)-ის გარშემო აუცილებელია. თუ ფრჩხილებს არ დავწერთ (მაგალითად: `double *func_ptr(double , double);`), კომპილატორი ჩათვლის, რომ ეს არის ჩვეულებრივი ფუნქციის დეკლარაცია, რომელიც აბრუნებს `double*` ტიპის მიმთითებელს!

3.6.2 ფუნქციაში ფუნქციის მიმთითებლის გადაცემა და გამოძახება

ფუნქციის მიმთითებლის ერთ-ერთი მთავარი უპირატესობა ისაა, რომ შეგვიძლია იგი გადავცეთ სხვა ფუნქციას არგუმენტად. ამ გზით შეგვიძლია შევქმნათ მოქნილი (უნივერსალური) ფუნქციები, რომლებიც შესასრულებელ ოპერაციას „გარედან“ იღებენ.

განვიხილოთ მაგალითი, სადაც შექმნილია გამოთვლის ფუნქცია `compute`, რომელსაც არგუმენტად გადაეცემა ორი რიცხვი და იმ მათემატიკური ოპერაციის (ფუნქციის) მიმთითებელი, რომელიც ამ რიცხვებზე უნდა შესრულდეს:

კოდი 3.9 ფუნქციის მიმთითებელი არგუმენტად

```c
#include <stdio.h>

// 1. მარტივი მათემატიკური ფუნქციები
double add(double a, double b)
{
 return a + b;
}

double multiply(double a, double b)
{
 return a * b;
}

// 2. ფუნქცია, რომელიც არგუმენტად იღებს ფუნქციის მიმთითებელს
double compute(double x, double y, double (*op)(double , double))
{
 // ფუნქციის გამოძახება მიმთითებლის საშუალებით
 return op(x, y);
}

int main(void)
{
 double n1 = 5.0, n2 = 3.0;

 // ა) ფუნქციის გადაცემა პირდაპირ სახელით
 double sum = compute(n1, n2, add);
 double prod = compute(n1, n2, multiply);
```

```
28
29    printf("n1 = %g, n2 = %g\n", n1, n2);
30    printf("Jami (add): %g\n", sum);
31    printf("Namravli (multiply): %g\n", prod);
32
33    // ბ) ცალკე მიმთითებელი ცვლადის შექმნა
34    double (*my_ptr)(double , double) = NULL;
35
36    my_ptr = add; // my_ptr ახლა უთითებს add ფუნქციაზე
37    printf("my_ptr-it gamoZaxeba: %g\n", my_ptr(10, 20));
38
39    return 0;
40  }
```

როგორც მაგალითიდან ჩანს:

- ფუნქციის სახელი (`add` ან `multiply`) ავტომატურად აღიქმება როგორც მისი მისამართი, ამიტომ `&add`-ის დაწერა აუცილებელი არ არის (თუმცა `&add`-იც სწორია).
- მიმთითებლით ფუნქციის გამოძახებისას შეგვიძლია დავწეროთ პირდაპირ `op(x, y)` ან `(*op)(x, y)` — ორივე სინტაქსი აბსოლუტურად ექვივალენტურია.

დავალება 3.1: კალკულატორი

1. შეცვალეთ კოდი 2.3 გვ. 29 ფუნქციებით. დაუმატეთ ხარისხში აყვანის ფუნქცია `pow` რაიმე ღილაკზე.
2. შეტანა შეცვალეთ შემდეგი თანმიმდევრობით: რიცხვი — ოპერაცია — რიცხვი.
3. შეამოწმეთ ნულზე გაყოფა.

3.7 დეკლარაცია/იმპლემენტაცია სხვადასხვა ფაილებში

როგორც ზემოთ ვთქვით, სანამ ფუნქცია გამოყენებულ იქნება სადმე (მაგ. `main` ფუნქციაში), საჭიროა, რომ პროგრამამ იცოდეს ამ ფუნქციის შესახებ. C/C++-ში შესაძლებელია ფუნქციის დეკლარაცია და იმპლემენტაცია (ცხადი სახე) იყოს სხვადასხვა ფაილებში.

როგორც წესი, `.c`, `.cpp`, `.cc`, `.cxx` გაფართოების ფაილები აღნიშნავენ იმპლემენტაციის ფაილებს, ხოლო `.h`, `.hpp`, `.hh` ფაილები — დეკლარაციის ფაილებს[1].

სათაო ფაილის მაგალითებია იგივე `stdio.h` და `math.h`, რაც უკვე შეგვხვდა. სათავო ფაილები მნიშვნელოვანია. შეიძლება არ გვქონდეს კოდის წყარო ფაილები (`.c`, `.cpp`) და გვქონდეს კომპილირებული ბიბლიოთეკა. იმისათვის, რომ გამოვიყენოთ ეს ბიბლიოთეკა, საჭიროა `.h` ფაილი, რომელსაც ჩავრთავთ კოდში და კომპილატორს "ვეტყვით" გამოიყენოს ეს ბიბლიოთეკა[2]. `.h` ფაილი შეიცავს ფუნქციების, ცვლადების, კლასების დეკლარაციებს.

ასეთი შემთხვევა პრაქტიკაში ხშირია. მაგალითად, თუ გვაქვს რაიმე ბიბლიოთეკა და-

[1] `.h` სიტყვისგან "header" — სათავო (თავი) ფაილი

[2] ცხადია, კომპილატორმა უნდა იცოდეს ფაილის მდებარეობაც

წერილი C-ზე (არ აქვს მნიშვნელობა, ამ ბიბლიოთეკის .c ფაილებს ვიყენებთ თუ ბინარულ (უკვე დაკომპილირებულ) .lib ან .a ფაილს[1]) და გვსურს მისი გამოყენება (ანუ ამ ბიბლიოთეკაში არსებული ცვლადების, ობიექტების, ფუნქციების) C++ პროგრამულ კოდში. ამისათვის საჭიროა გვქონდეს ამ ბიბლიოთეკის .h ფაილები.

დავუბრუნდეთ ზემოთ მოყვანილ მაგალითს (კოდი 3.8) და შევცვალოთ კოდები შემდეგი სახით[2]: პროექტში ვამატებთ ორ ფაილს — myheader.c, myheader.h შემდეგი კოდით:

კოდი 3.10 myheader.h დეკლარაციის ფაილი

```
1  //file myheader.h
2  #ifndef MYHEADER___H
3  #define MYHEADER___H
4  #include <math.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10  int solve_sqeq(const double* _a, const double* _b,
11  const double* _c, double* _x1, double* _x2);
12 #ifdef __cplusplus
13 }
14 #endif
15 #endif
```

ზემოთ მოყვანილ ფაილში 7-12 ხაზზე მოყვანილი კონსტრუქცია უზრუნველყოფს C ენაზე დაწერილი კოდის (ამ შემთხვევაში ფუნქციის) გამოყენებას C++ კოდში. extern ზოგადად ნიშნავს, რომ დეკლარირებული სიდიდე (ეს შეიძლება იყოს ცვლადი, ობიექტი, ფუნქცია) აღწერილია სადღაც სხვაგან, მოცემული ფაილის გარეთ და მისი ცხადი სახე ჩართული იქნება კოდში მიკავშირების ("linking") ეტაპზე. ასეთ შემთხვევაში ამ ცვლადს არ ამოწმებს კომპილატორი კომპილაციის ეტაპზე. თუ მიკავშირების დროს ვერ მოიძებნა ეს სიდიდე, მაშინ გვექნება შეცდომის შეტყობინება, რომ "ლინკერმა" ვერ იპოვნა ეს სიმბოლო და გვაქვს განუსაზღვრელი გარე სიმბოლო ("unresolved external symbol")[3].

მე-2 და მე-3 ხაზზე არსებული ბრძანებები უზრუნველყოფს, რომ როცა myheader.c დაკომპილირდება და შეიქმნება დროებითი ობიექტი myheader.obj (ან myheader.o — დამოკიდებულია კომპილატორზე) და ამის შემდეგ საჭირო იქნება ამ ობიექტის მიკავშირება მთავარ გამშვებ პროგრამასთან, ეს ობიექტი პროგრამაში ჩართული იქნება მხოლოდ ერთხელ. გავარჩიოთ ეს შემთხვევა დაწვრილებით:

ვთქვათ, კვადრატული განტოლების ფუნქცია გვჭირდება რამდენიმე სხვადასხვა ფა-

[1] Windows პლატფორმაზე ბიბლიოთეკებს აქვთ .lib და Linux/Unix სისტემებში .a გაფართოება

[2] კოდები და ვიდეო ფაილები იხილეთ კურსის ვებსაიტზე: http://lshoshia.science.tsu.ge.

[3] extern "C" მოგვიანებით განვიხილავთ.

ილში. მაშინ ყოველ ფაილში უნდა გვქონდეს `#include "myheader.h"`. ვთქვათ, მიკავშირების ეტაპზე შეხვდა "ლინკერს" ასეთი პირველი ფაილი. მაშინ ლინკერი ნახავს, რომ `MYHEADER___H` ჯერ არაა განმარტებული. შესრულდება მე-3 ხაზი, ანუ განიმარტება `MYHEADER___H` (გახდება ერთი) და `myheader.obj` ჩაირთვება პროგრამაში. გრძელდება მიკავშირების პროცესი. ვთქვათ, შეხვდა სხვა ფაილი, სადაც ასევე არის `#include "myheader.h"`, მაშინ "ლინკერი" კვლავ ეცდება ჩართოს `myheader.obj` პროგრამაში, მაგრამ ისევ მე-2 ხაზზე შემოწმებისას ნახავს, რომ `MYHEADER___H` უკვე განმარტებულია (ტოლია 1-ის), ამიტომ აღარ ჩართავს ამ ობიექტს პროგრამაში.

2-3 ხაზზე ბრძანებები რომ არ ყოფილიყო, მაშინ "ლინკერი" ეცდებოდა რამდენჯერმე ჩაერთო `myheader.obj` პროგრამაში და ეს იქნებოდა შეცდომა. ასეთ შემთხვევაში არის შეცდომური შეტყობინება მრავალჯერადად განმარტებული სიმბოლოების შესახებ (error: "Multiple defined symbols"). 12-13 ხაზზე პირველი `#endif` ეკუთვნის მე-2 ხაზზე არსებულ `#ifndef` („If not defined“ — „თუ არაა განმარტებული“), ხოლო მეორე — მე-7 ხაზს.

ქვემოთ მოყვანილია ფაილი `myheader.c`:

კოდი 3.11 myheader.c ფუნქციის იმპლემენტაცია

```
1  #include <math.h>
2  #include <stdio.h>
3  #define EPS 0.0000001
4  #include "myheader.h"
5  int solve_sqeq(const double* _a, const double* _b,
6  const double* _c, double* _x1, double* _x2)
7  {
8   // const double & bb = *_c; // შენიშვნა: C++ reference სინტაქსია და C-ში არ
      ↪ იმუშავებს
9   double d=(*_b)*(*_b)-4*(*_a)*(*_c);
10  if(d<0){
11   #ifdef _DEBUG
12   printf("D<0 ar aqvs amoxsna\n");
13   #endif
14   return 0;
15  }
16  else if(d<EPS)
17  {
18   (*_x1)=-(*_b)/(2*(*_a));
19   *_x2=*_x1;
20   #ifdef _DEBUG
21   printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
22   printf("x1=x2=%g\n",*_x1);
23   #endif
```

```
24   return 1;
25  }
26  *_x1=(-(*_b)-sqrt(d))/(2*(*_a));
27  *_x2=(-(*_b)+sqrt(d))/(2*(*_a));
28  #ifdef _DEBUG
29  printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
30  printf("x1=%g:\n x2=%g\n",*_x1,*_x2);
31  #endif
32  return 2;
33 }
```

როგორც ვხედავთ, თავად ფუნქციის კოდში არავითარი ცვლილება არაა. მთავარი პროგრამის ფაილიდან კოპირებულ იქნა ფუნქცია და ფაილის დასაწყისში მითითებული აქვს `#include "myheader.h"`.

შემდეგი ფაილი არის ის ფაილი, სადაც ხდება ფუნქციის გამოძახება. ჩვენს შემთხვევაში ესაა ფაილი, სადაც გვაქვს `main` ფუნქცია:

კოდი 3.12 main.c პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას.

```
1  #include "myheader.h"
2  #define EPS 0.0000001
3  int main(int argc, char* argv[])
4  {
5   double a=1; double b=2; double c=4;
6   double x1=0; double x2=0; int result=-1;
7   puts("shemoitanet kv. gant. koeficientebi: a, b, c");
8   scanf("%lf%lf%lf",&a,&b,&c);
9   result=solve_sqeq(&a,&b,&c,&x1,&x2);
10  switch(result)
11  {
12   case 0:
13   printf("D<0 ar aqvs amoxsna\n");
14   break;
15   case 1:
16   printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
17   printf("x1=%g:\n x2=%g\n", x1, x2);
18   break;
19   case 2:
20   printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
21   printf("x1=%g:\n x2=%g\n", x1, x2);
22   break;
```

```
23    default:
24    break;
25   }
26   return 0;
27  }
```

აქაც, როგორც ვხედავთ, ცვლილება არის მხოლოდ ის, რომ არ გვაქვს ფუნქციის დეკლარაცია (გადატანილ იქნა `myheader.h` ფაილში) და არც ფუნქციის ცხადი სახე (გადატანილ იქნა `myheader.c` ფაილში). ფაილის დასაწყისში მხოლოდ მითითებულია `#include "myheader.h"`. სხვა `#include` ბრძანებებიც აღარ არის აქ საჭირო, ვინაიდან შესაბამისი ფუნქციები ჩაერთვებიან პროგრამის კოდში, რადგან ეს ბრძანებები უკვე გვაქვს `myheader.h` ფაილში. ამ პროექტს თუ დავაკომპილირებთ, ჩვეულებრივ იმუშავებს.

3.8 extern "C"

C++ შემოიტანილ იქნა ე.წ. "name mangling", რაც საჭიროა იმისათვის, რომ ყოველ ფუნქციას, ცვლადს (თუ ის არის ე.წ. namespace-ის ნაწილი, რაც იძლევა საშუალებას განასხვავოს სხვადასხვა namespace-ში არსებული ერთსახელიანი ცვლადები) და ტიპს ჰქონდეს უნიკალური იდენტიფიკატორი ბინარულ დონეზე (მაგ. როცა გვაქვს ე.წ. ფუნქციის გადატვირთვა C++-ში). ეს მექანიზმი გაზიარებულია სხვა კომპილატორებშიც.

კომპილატორი გენერაციას უკეთებს ფუნქციის სახელებს, რომელშიც არის ინფორმაცია ფუნქციაზე გადაცემული არგუმენტების ტიპების შესახებ. C++-ში არ არსებობს ერთიანი სტანდარტი, თუ რა წესით ხდება ეს გენერირება და გენერირებული სახელები დამოკიდებულია კომპილატორზე. C++ კომპილატორი ასევე ცვლის C-ს ცვლადების/ფუნქციების სახელებსაც, რომ იდენტიფიცირება გაუკეთოს namespace-ს, სადაც ეს ცვლადები არსებობს.

გენერირებული სახელები დამოკიდებულია როგორც კომპილატორზე, ასევე ობიექტების მოდელზე. ერთი და იგივე კლასისათვისაც კი, იმის მიხედვით, თუ რა მოდელია გამოყენებული, გენერირებული სახელები იქნება სხვადასხვა. რა მოდელია გამოყენებული, ეს დამოკიდებულია კომპილაციის ოპციებზე.

"Name mangling" არაა სასურველი C მოდულების ან ობიექტების C++ ობიექტებთან მიკავშირებისას. C++ კომპილატორმა რომ არ მოახდინოს ფუნქციების/ცვლადების სახელების შეცვლა, ამისათვის საჭიროა `extern "C"` სპეციფიკატორი.

```
1 #ifdef __cplusplus
2 extern "C" {
3 #endif
4  int f1(int);
5  int f2(int);
6  int f3(int);
7 #ifdef __cplusplus
8 }
```

```
9 #endif
```

ეს ეუბნება C++ კომპილატორს, რომ f1, f2, f3 სახელების გენერირება არ უნდა მოხდეს.

C++ ფუნქციის C-დან გამოსაძახებლად, ფუნქციის დეკლარაცია C++-ის მხარეს უნდა იყოს შემოფარგლული extern "C" კონსტრუქციით, ხოლო C ფაილში ის აღიწერება ჩვეულებრივი C ფუნქციის სახით.

დაწვრილებით: http://en.wikipedia.org/wiki/Name_mangling

3.9 C პროექტის შექმნა

Visual Studio ფაილებს default-ად კომპილირების რეჟიმს გაფართოებით უსაზღვრავს:

- .c ფაილებს ავტომატურად C-ად აკომპილირებს.
- .cpp ფაილებს ავტომატურად C++-ად აკომპილირებს.

ამიტომ აუცილებელია, რომ VisualStudio შემთხვევაში `.cpp` გაფართოების ნაცვლად ფაილებს უნდა ჰქონდეს `.c` გაფართოება და კომპილატორს უნდა ვუთხრათ, რომ პროექტი დააკომპილიროს როგორც C კოდი.

GNU gcc-ის შემთხვევაში, თუ ვაკომპილირებთ C კოდს, ვიყენებთ gcc-ს როგორც კომპილატორს, ხოლო თუ ვაკომპილირებთ C++-ს, მაშინ — g++-ს.

ამას გარდა, როცა ვაკომპილირებთ C პროექტს, მაშინ `myheader.h` ფაილში შეგვიძლია მოვუხსნათ `#ifdef __cplusplus extern "C"` კონსტრუქცია.

მაგრამ სასურველია დავტოვოთ ორი მიზეზის გამო: ა) ეს არაფერს არ უშლის ხელს C კოდის კომპილაციისას და ბ) თუ გვსურს C კოდის/C ბიბლიოთეკის გამოყენება C++ პროგრამაში, ეს კონსტრუქცია საჭიროა. მით უმეტეს ვიზუალსტუდიოში თუ გვაქვს ე.წ. შერეული კოდი (ნაწილი .c და ნაწილი .cpp გაფართოებით.)

C კომპილატორით კომპილირებისას არის კიდევ ერთი პატარა ნიუანსი:

C-ის ოფიციალური სტანდარტის მიხედვით, ტექსტური ფაილის ყოველი სტრიქონი (მათ შორის პრეპროცესორის დირექტივები `#endif`, `#define`, `#include`) **სავალდებულოდ უნდა სრულდებოდეს `\n` (newline) სიმბოლოთი**. თუ ფაილი სრულდება პირდაპირ `#endif`-ით და ახალი ხაზი არ არის, C სტანდარტის მიხედვით ეს დირექტივა ფორმალურად ითვლება დაუსრულებლად და კომპილატორი მოგცემთ შეცდომას ფაილის მოულოდნელი დასასრულის შესახებ („error: unexpected end of file“). ბოლო `#endif`-ის შემდეგ, როგორც იტყვიან, *„Just hit Enter“.*

- **ძველ Visual Studio-ში:** კომპილატორი იძლეოდა მკაცრ შეცდომას:
  `fatal error C1004: unexpected end of file found.`
- **VS 2022-ში:** MSVC-ის ახალი კომპილატორი უმეტეს შემთხვევაში ჩუმად ასწორებს ამას და დაკომპილირდება, თუმცა გარკვეულ სიტუაციებში (განსაკუთრებით მკაცრი C11/C17 სტანდარტის ჩართვისას ან Warning level-ის აწევისას) მაინც ამოაგდებს გაფრთხილებას:
  `warning C4067: unexpected tokens following preprocessor directive - expected a newline.`

ჰედერ ფაილებთან პრობლემა იმიტომაა კრიტიკული, რომ `#include "file.h"`-ის დროს კომპილატორი პირდაპირ აერთებს (Concat) ტექსტებს.

თუ header1.h-ს ბოლოში არ აქვს \n:

```
// header1.h-ის ბოლო ხაზი:
#endif
```

და თქვენს .c ფაილში გიწერიათ:

```
#include "header1.h"
#include "header2.h"
```

კომპილატორი ამას დაინახავს როგორც ერთ ბმულს:

```
#endif#include "header2.h" // <-- შეცდომა!
```

ეს კი გამოიწვევს კომპილაციის ჩაშლას.
ამიტომ „**Hit Enter**“ ჩვევა ყოველი .c და .h ფაილის ბოლოს აუცილებელია.

თავი 4

C ენის სტანდარტები და იმპლემენტაცია

4.1 ახალი საკვანძო სიტყვები (Keywords) C99-დან C23-მდე 69
4.1.1 C99 საკვანძო სიტყვები 69
4.1.2 C11 საკვანძო სიტყვები 69
4.1.3 C23 საკვანძო სიტყვები 69
4.2 `math.h` და `complex.h` სათავო ფაილების გაფართოებები 70
4.2.1 Floating-point კლასიფიკაცია და მაკროები 70
4.2.2 Type-Generic Math (<tgmath.h>) 70
4.3 მაგალითი: კომპლექსური რიცხვი 70
4.4 Compile-Time კონსტანტები C23-ში: constexpr, const და #define 72
4.4.1 C23 სტანდარტი და შემოღების მიზეზები 72
4.4.2 #define მაკროსთან შედარება 72
4.4.3 const საკვანძო სიტყვასთან შედარება 74
4.4.4 `constexpr` C და C++-ში: ყოფაქცევა და განსხვავებები 75
4.4.5 constexpr-ის გამოყენება C23-ში 76
4.5 C11 `_Generic`, ფუნქციების გადატვირთვა (Function Overloading) C-ში და მაკროები 77
4.5.1 `_Generic`-ის და `static inline` ფუნქციების ტანდემი 78
4.5.2 მაკროს ისტორიული პრობლემების გადაჭრა 78
4.5.3 C და C++ Overloading-ის შედარება 79

როცა ამბობენ, რომ C++ არის C-ს გაფართოება, ეს გამართლებული არაა. მით უმეტეს, რომ სხვადასხვა კომპანიასა თუ ორგანიზაციას ჰქონდა/აქვს C++-ის განსხვავებული იმპლემენტაცია და C ცალკე, როგორც C++-ის ნაწილი, შესული იყო სხვადასხვა სტანდარტების სახით (ეს განსაკუთრებით ეხება Microsoft-ის კომპილატორებს. C++ სტანდარტული ბიბლიოთეკა 2003 წლამდე ნაკლული იყო და პროგრამისტები ძირითადად იყენებდნენ http://stlport.sourceforge.net/; რაც შეეხება C-ს, 2013 წლამდე Microsoft-ის C იმპლემენტაცია იყო C89. *Microsoft VS 2013* არის C99 სტანდარტი).

თანამედროვე `GNU` კომპილატორით (რომელსაც იყენებს, მაგალითად, `AVRStudio` Microchip/Atmel მიკროპროცესორების პროგრამირებისათვის) შესაძლებელია დაკომპილირდეს კოდი როგორც C89, ასევე C99 სტანდარტით.

სტანდარტამდე არსებულ C-ს უწოდებენ ''K&R C''-ს („კერნიგანის და რიჩის C“). როცა ამბობდნენ ''C'' 1972-1989 წლებში, იგულისხმებოდა კერნიგანის და რიჩის წიგნში აღწერილი ენა. C ენის გამოქვეყნების შემდეგ სხვადასხვა უნივერსიტეტებსა და კომპანიებში ამ ენის საფუძველზე შეიქმნა სხვადასხვა ვარიაციები/დამატებები.

1983 წელს ამერიკის ეროვნული სტანდარტების ინსტიტუტმა (*ANSI*) შექმნა C სტანდარტიზაციის კომისია. 1989 წელს მოხდა C სტანდარტის რატიფიცირება როგორც ANSI X3.159-1989 "Programming Language C". ამ სტანდარტს შემოკლებით უწოდებენ C89-ს ან ANSI-C-ს.

1990 წელს სტანდარტების საერთაშორისო ორგანიზაციის მიერ ANSI-C მიღებულ იქნა C ენის საერთაშორისო სტანდარტად (ISO 9899:1990, შემოკლებით — C90). ტექნიკური მხრივ, C90-სა და C89-ს შორის არავითარი განსხვავება არაა. 1990-1999 წლებში C ენა ესაა C90.

1999 წელს მოხდა C სტანდარტის გადახედვა (ISO 9899:1999, შემოკლებით C99). 1999-2011 წლებში პროგრამირების ენა C ესაა C99. კომპილატორების უმრავლესობა დღესაც იყენებს ამ სტანდარტს (Microsoft-ის კომპილატორი VS2013 ვერსიიდან).

2011 წელს კვლავ მოხდა სტანდარტის შეცვლა (შეცვლა არ გულისხმობს, ცხადია, რომ „ძველი“ კოდი არ დაკომპილირდება და არ იმუშავებს) — ISO 9899:2011. ამ სტანდარტს უწოდებენ C11-ს. http://en.wikipedia.org/wiki/C11_(C_standard_revision)

იმისათვის, რომ უზრუნველყონ კოდის პორტაბელურობა (კომპილირებადობა კოდში ცვლილებების შეტანის გარეშე სხვადასხვა სისტემებზე და სხვადასხვა კომპილატორით), C ენაზე დაწერილი კოდის დიდი ნაწილი არის C89 და ''K&R C'' სტანდარტითაც კი.

იმისათვის, რომ დაკომპილირებულიყო C99 კოდი *Microsoft VS* გარემოში, საჭირო იყო მისი კომპილაცია როგორც C++ კოდისა. თუ გავითვალისწინებთ, რომ *Microsoft VS*-ის 2013 წლამდე ვერსიები დღესაც ფართოდაა გავრცელებული, მდგომარეობა დიდად არ შეცვლილა. რაც ნიშნავს, რომ შეიძლება C-ზე დაწერილი კოდი (C99 სტანდარტით), რომელიც კომპილირდება `GNU gcc` კომპილატორით, არ დაკომპილირდეს *Microsoft VS* C კომპილატორით, არამედ კომპილატორს უნდა ვუთხრათ, რომ დააკომპილიროს როგორც C++ კოდი. ამიტომ საჭიროა ვიცოდეთ ის მინიმალური განსხვავებები და საერთო ნიშნები მაინც, რაც არის C ენის სხვადასხვა სტანდარტებსა და C++-ს შორის.

C89 და C99

- `//` კომენტარების ნიშანი გვაქვს C99-ში.
- C89 კომპილატორი ანსხვავებდა პირველ 31 ასო-ნიშანს, C99 — 63-ს.
- C99-ში არის ახალი სიტყვები: `inline`, `restrict`, `_Bool`, `_Imaginary`, `_Complex`.
- C89-ში `main` ფუნქცია არ აბრუნებდა არაფერს და ოპერაციული სისტემისათვის გაუგებარი იყო, წარმატებით დამთავრდა თუ არა პროცესი.
- C99-ში `main` ფუნქცია აბრუნებს `int` ტიპს და 0-ის დაბრუნება ნიშნავს პროცესის წარმატებით დასრულებას.
- C99-ში არის შემდეგი ბაზური ტიპები, რაც არ არის C89-ში: `long long int` და `unsigned long long int`, ასევე ამ ტიპების `const` ვარიანტები. C99-ში შესაძლებელია `const float`-ის თექვსმეტობითში ჩაწერა.

სხვა განსხვავებებს აღვნიშნავთ შესაბამის ნაწილებში. აქ მხოლოდ მოვიყვანთ კიდევ რამდენიმე განსხვავებას, რომელიც შეეხება ცვლადების დეკლარირებას და ფუნქციებს:

ცვლადების დეკლარირება: როგორც ვთქვით, პროგრამა შედგება ინსტრუქციებისაგან (ბრძანებები). ყოველ ბრძანება მთავრდება წერტილ-მძიმით ';'. ბრძანებები შეიძლება გაერთიანებული იყვნენ ბლოკში. ბლოკი იწყება ფიგურული ფრჩხილით '{' და მთავრდება ფიგურული ფრჩხილით '}'. C89 სტანდარტის მიხედვით, ცვლადი დეკლარირებული უნდა იყოს ბრძანებათა ბლოკის დასაწყისში, ანუ '{'-ის შემდეგ. C99 (და C++)-ში კი ცვლადი შეიძლება დეკლარირებულ იქნას კოდის ნებისმიერ ადგილას. მაგალითად:

```
{
 int x=1;
 printf("Hello\n");
}
```

დაკომპილირდება როგორც C++, ასევე C კომპილატორით (C89, C90), ხოლო ქვემოთ მოყვანილი კოდი მოგვცემს შეცდომას C89 ოპციით კომპილაციისას:

```
{
 printf("Hello\n");
 int x=1;
}
```

C99-ში (და C++-ში) შესაძლებელია `for` ციკლში პირველი გამოსახულება იყოს დეკლარაცია, მათ შორის ციკლის საკუთარი ცვლადების დეკლარაცია:

```
for(int i=0;i<10;i++)
{
 /*შესრულებადი კოდი*/
}
```

კომპილირდება C99-სა და C++-ში, მაგრამ არა C89-ში. ამ უკანასკნელისთვის გვაქვს:

```
int n=10;
int i;
for( i=0;i<n;i++)
{
 /*შესრულებადი კოდი*/
}
```

C89-ში შეიძლება არ იქნას მითითებული, თუ რა ტიპს აბრუნებს ფუნქცია. თუ არაა მითითებული, იგულისხმება, რომ აბრუნებს `int`-ს. C99 (და C++) მოითხოვს, რომ დაბრუნების

ტიპი ცხადად იყოს მითითებული.

C99 (და C++) მოითხოვს, რომ ფუნქცია დეკლარირებული ან განმარტებული იყოს მის გამოძახებამდე. ასეთი მოთხოვნა არაა C89-ში. თუ ფუნქცია გამოყენებულია მის დეკლარაციამდე ან განმარტებამდე, მაშინ C89 კომპილატორი უშვებს, რომ ფუნქცია აბრუნებს `int` მნიშვნელობას.

აქვე აღვნიშნოთ რამდენიმე განსხვავება K&R C-სა და C89-ს შორის: K&R C-ში არ იყო `enum` ტიპი, `unsigned` შეიძლებოდა ყოფილიყო მხოლოდ `int`, არ იყო `long long`, `const` , `void` და ა.შ. C89-ში დასაშვებია ფუნქციის დეკლარაციისას გადაცემულ პარამეტრებში მითითებული იყოს ცვლადები.

```
1  double square(double x); // სწორია C89
2
3  double square(double);   // სწორია C89 და K&R
```

ფუნქციის იმპლემენტაციისას K&R C მოითხოვს, რომ ცვლადები აღწერილი იყოს ცალკე ჩამონათვალში. ესაა C89:

```
1  double square(double x)
2  {
3   return x*x;
4  }
```

ესაა K&R:

```
1  double square(x)
2  double x;
3  {
4   return x*x;
5  }
```

C სტანდარტიზაციის პროცესის მიმართულებები არის შემდეგი:

ტიპიზაციის გამკაცრება — რაც არ იყო მკაცრი მოთხოვნა K&R C ენაში (მაგ. არა-პირდაპირი `int` ტიპი), მკაცრად მოთხოვნადია ახალ სტანდარტებში;

უსაფრთხოების გაზრდა — პოტენციურად საშიში კონსტრუქციების გაუქმება (მაგ. `gets()` ფუნქციის ამოღება C11-ში) და უსაფრთხო ალტერნატივების შემოღება;

თანამედროვე აპარატურის მხარდაჭერა — მრავალნაკადიანობა (*Multithreading*), ატომური ოპერაციები და მეხსიერების ეფექტური გასწორება (*Alignment*);

ახალი ფუნქციების, ტიპებისა და ბიბლიოთეკების დამატება — C++ სტანდარტთან დაახლოება და ენის სინტაქსური გაუმჯობესება.

ISO C90-ში (ANSI C89) არის 32 საკვანძო სიტყვა:

ტიპები: `int`, `double`, `long`, `char`, `float`, `short`, `signed`, `unsigned`, `typedef`, `sizeof` (10).

კონტროლი: `if`, `else`, `switch`, `case`, `break`, `default`, `for`, `do`, `while`, `continue`, `goto` (11).

ფუნქცია: `return`, `void` (2).

მონაცემთა სტრუქტურა: `struct`, `enum`, `union` (3).

მეხსიერება: `auto`, `register`, `extern`, `const`, `volatile`, `static` (6). სულ 32 სიტყვა.

4.1 ახალი საკვანძო სიტყვები (Keywords) C99-დან C23-მდე

C89-ის შემდეგ ენას დაემატა მრავალი მნიშვნელოვანი საკვანძო სიტყვა:

4.1.1 C99 საკვანძო სიტყვები

ISO C99-ში დამატებულია 5 სიტყვა, სულ 37:

`_Bool` (მაკრო `bool` `<stdbool.h>`-დან) — ლოგიკური ტიპი.

`_Complex` და `_Imaginary` — კომპლექსური რიცხვების მხარდაჭერა.

`inline` — ფუნქციის შიდა გაფართოება (Inlining).

`restrict` — მეხსიერების ალიასინგის ოპტიმიზაცია.

4.1.2 C11 საკვანძო სიტყვები

ISO C11-ში დამატებულია კიდევ 7 სიტყვა, სულ 44:

`_Generic` — ენის დონეზე ფსევდო-გენერიკების (Overloading) მხარდაჭერა ტიპის მიხედვით.

`_Static_assert` — კომპილაციის დროის ასერციები.

`_Alignas` და `_Alignof` — მეხსიერების გასწორების (Alignment) კონტროლი SIMD გამოთვლებისთვის.

`_Atomic` — ატომური ოპერაციები მრავალნაკადიან (Multi-threaded) გარემოში.

`_Thread_local` — ნაკადის ლოკალური ცვლადები.

`_Noreturn` — ფუნქციებისთვის, რომლებიც არ უბრუნდებიან გამოძახების ადგილს.

4.1.3 C23 საკვანძო სიტყვები

ISO C23-ში დაემატა ახალი საკვანძო სიტყვები და „_“-ით დაწყებული პრეფექსები ჩანაცვლდა სტანდარტული ქვედა რეგისტრის სინტაქსით:

`alignas`, `alignof`, `bool`, `constexpr`, `false`, `nullptr`, `static_assert`, `thread_local`, `true`, `typeof`, `typeof_unqual`, `_BitInt` (12).

`bool`, `true`, `false` — პირდაპირ ენის საკვანძო სიტყვები გახდა.

`alignas`, `alignof`, `static_assert`, `thread_local` — სტანდარტული სახელი.

`constexpr` — კომპილაციის დროს მუდმივების გამოთვლა.

`typeof` და `typeof_unqual` — ტიპის ავტომატური განსაზღვრა.

`nullptr` — ტიპიზირებული ნულოვანი პოინტერი (ანაცვლებს `NULL`-ს).

4.2 `math.h` და `complex.h` სათავო ფაილების გაფართოებები

თანამედროვე C-ში მათემატიკურ ბიბლიოთეკებში დაემატა მაღალი სიზუსტის და სპეციალიზებული ფუნქციები:

4.2.1 Floating-point კლასიფიკაცია და მაკროები

`fpclassify(x)` — აბრუნებს ტიპს (`FP_NAN`, `FP_INFINITE`, `FP_ZERO`, `FP_SUBNORMAL`, `FP_NORMAL`).

`isnan(x)`, `isinf(x)`, `isfinite(x)` — სწრაფი შემოწმების მაკროები.

4.2.2 Type-Generic Math (<tgmath.h>)

<tgmath.h> იყენებს _Generic მექანიზმს, რათა ავტომატურად აირჩიოს `sin()`, `sinf()`, `sinl()` ან `csin()` არგუმენტის ტიპის მიხედვით(ფუნქციების გადატვირთვა C-ში, მოგვიანებით დაწვრილებით განვიხილავთ).

კოდი 4.1 c11_generic_example

```c
#include <stdio.h>

// _Generic მაკროს მაგალითი C11-ში
#define print_type(x) _Generic((x), \
int: "mteli (int)", \
double: "namdvili (double)", \
default: "ucnobi tipi" \
)
int main() {
 int a = 10;
 double b = 3.14;
 printf("a is: %s\n", print_type(a));
 printf("b is: %s\n", print_type(b));
 return 0;
}
```

4.3 მაგალითი: კომპლექსური რიცხვი

C99 სტანდარტში და შემდეგ C ენას გააჩნია <complex.h> ბიბლიოთეკა, სადაც კომპლექსური $z = a+bi$ რიცხვების რეალური და წარმოსახვითი ნაწილების ამოსაღებად გამოიყენება `creal()` და `cimag()` ფუნქციები.

C99 სტანდარტში (მაკროს სახით) დაემატა საკვანძო სიტყვა _Complex (პრეფიქსით _, რათა არსებულ კოდში იდენტიფიკატორებთან კონფლიქტი აეცილებინათ). ამავდროულად შემოიღეს <complex.h> სათავო (header) ფაილი, სადაც ეწერა მაკრო: `#define complex _Complex`. შესაბამისად, C99-იდან C17-მდე `complex` სიტყვის გამოყენება შესაძლებელი იყო მხოლოდ ამ ფაილის ჩართვით

(`#include <complex.h>`).
C23-ში `complex` გახდა C ენის ოფიციალური, პირდაპირი საკვანძო სიტყვა (ისევე როგორც `bool`, `alignas`, `static_assert` და ა.შ.). C23-ში მისი გამოყენება უშუალოდ შეიძლება სათავო ფაილის ჩართვის გარეშეც, ხოლო `_Complex` შენარჩუნებულია ძველ კოდთან უკუთავსებადობისთვის (Alternative spelling).

```c
#include <stdio.h>
#include <complex.h>

int main() {
 // 1. კომპლექსური რიცხვების გამოცხადება
 double complex z1 = 3.5 + 4.2 * I;
 double complex z2 = 1.2 - 2.8 * I;

 printf("=== kompleqsuri rickhvis bechdva ===\n\n");

 // 2. მარტივი გამოყვანა (%.2f ზღუდავს ათწილადს 2 ნიშნამდე)
 printf("z1 = %.2f + %.2fi\n", creal(z1), cimag(z1));

 // 3. უარყოფითი წარმოსახვითი ნაწილის დამუშავება
 double real_part = creal(z2);
 double imag_part = cimag(z2);

 if (imag_part >= 0) {
  printf("z2 = %.2f + %.2fi\n", real_part, imag_part);
 } else {
  printf("z2 = %.2f - %.2fi\n", real_part, -imag_part);
 }

 return 0;
}
```

დავალება 4.1: კვადრატული განტოლება

დაწერეთ ფუნქცია, რომელიც მიიღებს კვადრატული ორწევრის კოეფიციენტებს და დააბრუნებს ინფორმაციას ფესვების შესახებ (0, 1, 2 ფესვი — რეალური ან კომპლექსური) და x_1 და x_2 კომპლექსურ ცვლადებში შეინახავს კვადრატული განტოლების ფესვებს.

დასკვნა: *თანამედროვე C ენაზე პროგრამირებისას რეკომენდებულია **C11** ან **C17** სტანდარტის გამოყენება (ხოლო მაქსიმალური პორტაბელურობისთვის ჩაშენებულ/Embedded სისტემებში — მინიმუმ **C99**). C89 სტანდარტი დღეს მოძველებულია და ზღუდავს ენის თა-*

ნამედროვე შესაძლებლობებს. არ არის რეკომენდებული C კოდის კომპილაცია C++ კომპილატორით, რადგან C++ არ წარმოადგენს C ენის სრულ ქვესიმრავლეს და შეიცავს განსხვავებულ ტიპიზაციის წესებს. C99, C11 და C17 სტანდარტების სრული მხარდაჭერა დღეს გაჩნია ყველა წამყვან კომპილატორს (GCC, Clang, MSVC, Intel OneAPI)[1].

4.4 Compile-Time კონსტანტები C23-ში: constexpr, const და #define

C ენის განვითარების ისტორიაში კონსტანტების სწორი განმარტება ყოველთვის დაკავშირებული იყო გარკვეულ არქიტექტურულ კომპრომისებთან. C23 სტანდარტამდე (ISO/IEC 9899:2024) ენაში არ არსებობდა მექანიზმი, რომელიც ერთდროულად უზრუნველყოფდა ტიპების უსაფრთხოებას (Type Safety), წვდომის არეალის შეზღუდვას (Block Scope) და კომპილაციის დროის გარანტირებულ კონსტანტურობას (Compile-Time Evaluation) მეხსიერების დანახარჯის გარეშე.

C23 სტანდარტში შემოღებულ იქნა `constexpr` საკვანძო სიტყვა ცვლადებისთვის, რამაც ფუნდამენტურად შეცვალა კონსტანტებთან მუშაობის პრაქტიკა C-ში.

4.4.1 C23 სტანდარტი და შემოღების მიზეზები

`constexpr` ოფიციალურად დაემატა C ენას C23 სტანდარტით. ამ სიახლის შემოღება განაპირობა რამდენიმე ძირითადმა მიზეზმა:

1. **C-ის ისტორიული შეზღუდვა:** C89/C99/C11 სტანდარტებში `const` საკვანძო სიტყვა ნიშნავს მხოლოდ "*Read-Only ცვლადს*" და არა ჭეშმარიტ კომპილაციის დროის კონსტანტას.
2. **#define-ის ნაკლოვანებები:** ჭეშმარიტი compile-time კონსტანტის მისაღებად დეველოპერები იძულებულნი იყვნენ გამოეყენებინათ პრეპროცესორის მაკროსები (`#define`), რაც არღვევდა ტიპების უსაფრთხოებას და Scope-ის წესებს.
3. **C++-თან თავსებადობა (Interoperability):** თანამედროვე C++-ში `constexpr` ფართოდ გამოიყენება. C23-მა გადადგა ნაბიჯი ორივე ენის სინტაქსური და ლოგიკური ერთგვაროვნებისკენ.

4.4.2 #define მაკროსთან შედარება

მსგავსება #define-თან:

Compile-Time განსაზღვრულობა: ორივეს მნიშვნელობა ცნობილია კომპილაციის ეტაპზე.

ნულოვანი Overhead მეხსიერებაში: ჩვეულებრივ პირობებში `constexpr` ცვლადი არ იკავებს ადგილს RAM-ში. კომპილატორი მის მნიშვნელობას პირდაპირ პროცესორის მანქანურ ინსტრუქციაში ჩასვამს (Immediate Value).

გამოყენება სტატიკურ კონტექსტში: ორივე გამოდგება სტატიკური მასივების ზომად (`int arr[SIZE]`), `switch-case` ბლოკებში და ბიტურ ნიღბებად.

[1] ისტორიულად, `Visual Studio` 2013-მდე MSVC არ უჭერდა მხარს C99-ს და მის ბიბლიოთეკებს C++11-ის მეშვეობით ინტეგრირებდა, თუმცა `Visual Studio` 2019-ის v16.8 ვერსიიდან MSVC-ს აქვს C11/C17-ის ოფიციალური მხარდაჭერა (`/std:c11` და `/std:c17`).

განსხვავება #define-ისგან:

ტიპების უსაფრთხოება (Type Safety):

- #define არის უბრალო ტექსტური ჩანაცვლება. constexpr-ს გააჩნია მკაცრად განსაზღვრული ტიპი (მაგ. constexpr float PI = 3.14159f; მკაცრად 32-ბიტიანი float-ია). #define PI 3.14159265358979323846 პრეპროცესორისთვის PI არ არის float ან double. ეს არის უბრალოდ სიმბოლოების ტექსტური ნაკრები. თუ ამ PI-ს გადასცემთ ფუნქციას, კომპილატორი იძულებული გახდება თავად გამოიცნოს ტიპი (მაგალითად, მიანიჭოს double), რამაც შეიძლება გამოიწვიოს მეხსიერების ზედმეტი დანახარჯი ან ტიპების არასასურველი გარდაქმნა (Implicit Conversion) გაფრთხილების გარეშე.
- **არაპროგნოზირებადი ტიპების გაფართოება (Implicit Promotion):** თუ განმარტავთ ჰექსადეციმალურ ან დიდ რიცხვს: #define MASK 0xFFFFFFFF რადგან ტიპი მითითებული არ არის, სხვადასხვა კომპილატორმა (ან სხვადასხვა პლატფორმაზე, 32-bit vs 64-bit) ეს ტექსტი შეიძლება აღიქვას როგორც int, unsigned int ან long long იმის მიხედვით, თუ კოდის რომელ მონაკვეთში ჩაჯდება ეს ტექსტი. ამან შეიძლება გამოიწვიოს ნიშნის დაკარგვა (Signed/Unsigned Bugs) ან ცვლადის გადავსება (Overflow).

Scope (წვდომის არეალი): #define იგნორირებას უკეთებს ფრჩხილებს {} და გლობალურია. constexpr ემორჩილება C-ს ლოკალურ ბლოკურ Scope-ს.

დებაგირება: #define ქრება კომპილაციამდე (პრეპროცესორის ფაზაში). constexpr რჩება კომპილატორის სიმბოლოების ცხრილში (Symbol Table), რაც დებაგერს საშუალებას აძლევს აჩვენოს ცვლადის სახელი და ტიპი.

გვერდითი ეფექტების პრევენცია: მაგალითი, სადაც #define აფუჭებს კოდს, constexpr კი არა:

```
1  void processData() {
2   {
3    constexpr int size = 64; // size არსებობს მხოლოდ ამ ფრჩხილებში
4   }
5
6   // აქ 'size' უკვე აღარ არსებობს, C23 სწორად ამოაგდებს შეცდომას.
7   int size = 128; // სრულიად უსაფრთხოა!
8  }
9
10 void processDataWithMacro() {
11  {
12   #define SIZE 64
13  }
14
15  // #define-მა იგნორირება გაუკეთა ფრჩხილებს!
```

```
    int SIZE = 128; // შეცდომა! პრეპროცესორი ამას ჩაანაცვლებს როგორც: int 64 = 128;
}
```

ამიტომ C23-ში `constexpr int x = 10;`-ის შემოღების მთავარი მიზანი სწორედ `#define`-ებისგან კოდის გაწმენდა და უსაფრთხოებაა.

- პრობლემა განსაკუთრებით მწვავდება მაკრო-ფუნქციებთან. განვიხილოთ კლასიკური მაგალითი: `#define SQUARE(x) (x * x)`
  ოპერატორების პრიორიტეტის პრობლემა: თუ გამოვიძახებთ: `SQUARE(1 + 2)` პრეპროცესორი ჩაანაცვლებს ტექსტურად: `(1 + 2 * 1 + 2)` შედეგი იქნება 5 და არა 9 (რადგან გამრავლება სრულდება შეკრებაზე ადრე)!
- **ინკრემენტის პრობლემა:** მაკროსის მრავალჯერადი გაშლა(გახსნა) (`SQUARE(a++)`) იწვევს შეცდომებს. თუ გამოვიძახებთ: `SQUARE(a++)` პრეპროცესორი ჩაანაცვლებს: `(a++ * a++)` ცვლადი `a` გაიზარდება ორჯერ, რაც იწვევს გაურკვეველ ქცევას (Undefined Behavior).

C++-ის შემთხვევაში `constexpr` როცა ვიყენებთ ფუნქციასთან ეს რისკები ნულამდე დადის, რადგან არგუმენტები ჯერ ითვლება ტიპიზირებულად და შემდეგ გადაეცემა ფუნქციას. C++11 დან შეგვიძლია დავწეროთ:

```
constexpr int square(int x) { return x*x;}
...
constexpr int a = square(5); // გამოითვლება კომპილაციისას! a = 25 (0 overhead)
```

ხოლო, Runtime-ზე(ანუ პროგრამის შესრულებისას) თუ არგუმენტი ჩვეულებრივი ცვლადია:

```
int x = 5; // Runtime ცვლადი
int b = square(x); // შესრულდება პროგრამის გაშვებისას (როგორც ჩვეულებრივი inline ფუნქცია)
```

შესრულდება როგორც ჩვეულებრივი inline ფუნქცია.

რაც შეეხება C23 `#define SQUARE(x) (x * x)` მსგავსი ანალოგი `constexpr` საშულებით არ არსებობს, ვინაიდან `constexpr` განმარტებულია მხოლოდ ცვლადებზე.

მსგავსი პრობლემები იჭრება C-ში `static inline` ფუნქციების გამოცხადებით `_Generic`-თან კომბინაციაში (ამას ქვემოთ განვიხილავთ)

4.4.3 const საკვანძო სიტყვასთან შედარება

მსგავსება const-თან

Read-Only სტატუსი: ორივეს მნიშვნელობის შეცვლა გამოცხადების შემდეგ აკრძალულია კომპილატორის მიერ.

სინტაქსი და Scope: ორივე ემორჩილება C-ს სტანდარტულ ტიპიზაციასა და წვდომის არეალებს.

განსხვავება const-ისგან

მეხსიერების გამოყოფა: C-ში `const int x = 10;` არის **Runtime ცვლადი**. ის აუცილებლად იკავებს ადგილს მეხსიერებაში (Stack ან `.rodata`). `constexpr int x = 10;` კი მეხსიერებას **არ იკავებს** (0 ბაიტი), თუ პირდაპირ არ ავიღებთ მის მისამართს (`&x`).
Integer Constant Expression: C-ში `const` ცვლადი **არ არის** ჭეშმარიტი კონსტანტა. შესაბამისად, `const` ცვლადით სტატიკური მასივის შექმნა C99/C11-ში ქმნის VLA-ს (Variable Length Array), ხოლო C89-ში იწვევს შეცდომას. `constexpr` კი გარანტირებულად ქმნის ფიქსირებული ზომის სტატიკურ მასივს.

4.4.4 `constexpr` C და C++-ში: ყოფაქცევა და განსხვავებები

`constexpr`-ის ქცევა C23-სა და C++-ს შორის იდენტური არ არის:

ცვლადებთან: C23 და C++ ავლენენ იდენტურ ქცევას. ორივეში `constexpr` ცვლადი არის ტიპიზირებული compile-time კონსტანტა.

ფუნქციებთან: C23-ში `constexpr` ფუნქციებზე არ ვრცელდება. ფუნქციაზე `constexpr`-ის მიწერა C23-ში გამოიწვევს კომპილაციის შეცდომას (`Compilation Error`).

C++-ში `constexpr` ფუნქცია კომპილატორს ავალებს გამოთვალოს ფუნქციის ტანი კომპილაციის პროცესშივე (ჩაშენებული ინტერპრეტატორის მეშვეობით), თუ არგუმენტები compile-time კონსტანტებია. C23 სტანდარტი არ მოიცავს კომპილაციის დროის ასეთ ინტერპრეტატორს ფუნქციებისთვის, ამიტომ C23-მა ფუნქციებზე `constexpr` შეგნებულად არ გაავრცელა.

`#define` გამოიყენეთ მხოლოდ: Header Guard-ებისთვის (`#ifndef HEADER_H`), პლატფორმული კომპილაციისთვის (`#ifdef _WIN32`) და მაკრო-ფუნქციებისთვის.
`const` გამოიყენეთ: ფუნქციის არგუმენტად პოინტერების გადაცემისას (`const char* str`), რათა დაიცვათ მონაცემები ცვლილებისგან, ან Runtime-ში გამოთვლილი Read-Only მონაცემებისთვის.
`constexpr` გამოიყენეთ: ყველა ტიპიზირებული compile-time კონსტანტისთვის (მასივის ზომები, მათემატიკური კონსტანტები, ბუფერის ზღვრები, ფლაგების ბიტები).

ცხრილი 4.1 #define, const და constexpr-ის შედარება C23-ში

მახასიათებელი	#define	const (C)	constexpr (C23)
ტიპების უსაფრთხოება (Type Safety)	არა	დიახ	**დიახ**
ჭვდომის არეალი (Scope)	გლობალური	ბლოკური ({})	**ბლოკური ({})**
კომპილაციის დროის კონსტანტა?	დიახ	არა	**დიახ**
RAM მეხსიერების დანახარჯი	0 ბაიტი	> 0 ბაიტი	**0 ბაიტი** (*თუ & არ ავიღეთ)
გამოყენება მასივის ზომად	დიახ	არა (VLA C99-ში)	**დიახ (სტატიკური)**
ჩანს დებაგერში?	არა	დიახ	**დიახ**
მუშაობს ფუნქციებზე?	დიახ (მაკრო)	დიახ (პოინტერები)	**არა (მხოლოდ C++)**

4.4.5 constexpr-ის გამოყენება C23-ში

ქვემოთ მოყვანილია C23 სტანდარტის შესაბამისი კოდი, სადაც ნაჩვენებია `constexpr`-ის, `const`-ისა და `#define`-ის პრაქტიკული ქცევა:

კოდი 4.2 constexpr-ის გამოყენება C23-ში

```c
#include <stdio.h>

// 1. #define - ძველი გზა (არ აქვს ტიპი, გლობალურია)
#define MAX_BUFFER_SIZE 1024

// 2. constexpr - C23-ის თანამედროვე, ტიპიზირებული კონსტანტა
constexpr float PI = 3.14159265f;
constexpr int ARRAY_LEN = 5;

void process_data(const int* read_only_ptr) {
 // const აქ გამოიყენება პოინტერის დასაცავად (Runtime read-only)
 printf("Value: %d\n", *read_only_ptr);
```

```
}

int main(void) {
 // 3. constexpr ემორჩილება Block Scope-ს
 {
  constexpr int LOCAL_LIMIT = 50;
  // int test_arr[LOCAL_LIMIT]; // ვალიდურია!
 }
 // LOCAL_LIMIT აქ უკვე აღარ არსებობს (კომპილაციის შეცდომა მოხდება)

 // 4. სტატიკური მასივის შექმნა constexpr-ით (0 overhead)
 int data[ARRAY_LEN] = {10, 20, 30, 40, 50};

 // 5. const C-ში: ეს არის RUNTIME ცვლადი!
 const int runtime_size = 10;
 // int bad_array[runtime_size]; // C99-ში შექმნის VLA-ს, C23-ში არ არის
  ↪ constexpr!

 // 6. მეხსიერების დემონსტრაცია
 constexpr int x = 100; // 0 ბაიტი RAM-ში (Immediate Value)

 // თუ ავიღებთ მისამართს (&), კომპილატორი იძულებული გახდება გამოყოს RAM:
 const int* ptr = &x;

 process_data(ptr);

 return 0;
}
```

4.5 C11 _Generic, ფუნქციების გადატვირთვა (Function Overloading) C-ში და მაკროები

C11 სტანდარტამდე C ენას არ გააჩნდა ფუნქციების გადატვირთვის (Function Overloading) მექანიზმი. ამის გამო დეველოპერები იძულებულნი იყვნენ ან სხვადასხვა სახელი დაერქმიათ ფუნქციებისთვის (square_int, square_float), ან გამოიყენებინათ პრეპროცესორის საშიში მაკროსები (#define SQUARE(x) ((x)*(x))).

```
// 1. ცუდი/საშიში გზა C-ში (მაკრო - არ არის Type-Safe, აქვს a++-ის პრობლემა)
  ↪ :
#define SQUARE_MACRO(x) ((x) * (x))

```

```
4  // 2. სწორი, უსაფრთხო გზა C-ში (static inline ფუნქცია):
5  static inline int square_int(int x) {
6   return x * x;
7  }
8
9  static inline float square_float(float x) {
10  return x * x;
11 }
```

C11 სტანდარტში შემოღებულ იქნა _Generic კონსტრუქცია, რომელმაც C-ს მოუტანა კომპილაციის დროის ტიპობრივი არჩევანის (Compile-Time Type Dispatching) საშუალება — ფაქტობრივად C++-ის ფუნქციების გადატვირთვის ანალოგი.

4.5.1 _Generic-ის და static inline ფუნქციების ტანდემი

C-ში სიმბოლოების ცხრილი (Symbol Table) არ უშვებს ფუნქციებისთვის იდენტური სახელების დარქმევას, რადგან მას არ გააჩნია C++-ის კომპილატორის შიდა *Name Mangling* მექანიზმი. ამიტომ, C-ში ფუნქციების უსაფრთხო გადატვირთვისთვის საუკეთესო არქიტექტურული პრაქტიკაა static inline ფუნქციებისა და _Generic მაკროს გაერთიანება:

კოდი 4.3 _Generic-ის გამოყენება ფუნქციების გადატვირთვისთვის C11-ში

```
1  #include <stdio.h>
2
3  // 1. კონკრეტული ტიპების static inline ფუნქციები (Type-Safe)
4  static inline int square_int(int x) { return x * x; }
5  static inline float square_float(float x) { return x * x; }
6  static inline double square_double(double x) { return x * x; }
7
8  // 2. _Generic ფასადი (Function Overloading C-ში)
9  #define square(x) _Generic((x), \
10 float: square_float, \
11 double: square_double, \
12 default: square_int \
13 )(x)
```

4.5.2 მაკროს ისტორიული პრობლემების გადაჭრა

ტრადიციული მაკროსგან (#define) განსხვავებით, _Generic მაკრო თავად კი არ ასრულებს გამოთვლას, არამედ **მხოლოდ ირჩევს** შესაბამის static inline ფუნქციას. ეს სრულად აგვარებს მაკროების ორ მთავარ ისტორიულ პრობლემას:

1. **გვერდითი ეფექტები (x++ ინკრემენტის პრობლემა):** C სტანდარტის მიხედვით, _Generic((x), ...) კონსტრუქციის პირველი არგუმენტი არის *არაშესრულებადი გამოსახულება* (Unevaluated Expression, ისევე როგორც sizeof(x)). კომპილატორი

მას უყურებს მხოლოდ ტიპის დასადგენად და შიდა კოდს არ ასრულებს.
როდესაც ვწერთ `square(a++)`, გაშლილი კოდი იღებს სახეს:

$$\texttt{_Generic((a++), ...)(a++)} \longrightarrow \texttt{square_int(a++)}$$

`_Generic`-ის შიგნით `a++` არ სრულდება, ხოლო არჩეულ `square_int` ფუნქციას `a++` გადაეცემა მნიშვნელობით (Pass-by-value). შესაბამისად, ცვლადი გაიზრდება **ზუსტად ერთხელ**.

2. **ოპერატორების პრიორიტეტი (`1 + 2`):** გამოძახება `square(1 + 2)` გარდაიქმნება ჩვეულებრივ ფუნქციის გამოძახებად: `square_int(1 + 2)`. C ჯერ გამოითვლის ფრჩხილებში არსებულ არგუმენტს ($1 + 2 = 3$) და მხოლოდ შემდეგ გადასცემს ფუნქციას. შედეგად მიიღება ზუსტი პასუხი (9) და არა 5.

4.5.3 C და C++ Overloading-ის შედარება

C++ (Native Overloading): კომპილატორს ენის დონეზე ესმის გადატვირთვა და ფუნქციის იდენტურ სახელებს შიგნით თავად გარდაქმნის (Name Mangling).
C11 (`_Generic` Dispatch): ფუნქციებს ერქმევათ ინდივიდუალური სახელები (`square_int`, `square_float`), ხოლო `_Generic` ქმნის ერთიან, უსაფრთხო და მოხერხებულ ფასადს დეველოპერისთვის (Developer Experience).

შედეგად, `_Generic` + `static inline` მიდგომა C-ში უზრუნველყოფს: **ტიპების სრულ უსაფრთხოებას**, **გვერდითი ეფექტებისგან დაცვას** და **ნულოვან დანახარჯს (Zero Overhead)**, რადგან კომპილატორი `inline` ფუნქციებს პირდაპირ ჩასვამს(ჩააშენებს) მანქანურ კოდში.

თავი 5

ფაილში ჩაწერა და წაკითხვა

5.1 ფაილის ცნება და მოდელი C-ში 81
5.2 ტექსტური და ბინარული რეჟიმი 82
5.2.1 ტექსტურ და ბინარულ ფორმატებს შორის განსხვავება 82
5.2.2 სტანდარტული ნაკადები (Streams) 83
5.3 ფაილის გახსნა და დახურვა: `fopen` და `fclose` 83
5.4 ასო-ასო ჩაწერა და წაკითხვა: `fgetc()` და `fputc()` 84
5.4.1 მაგალითი 1: სიმბოლოების წაკითხვა და ეკრანზე გამოტანა 84
5.4.2 ფაილის დასასრულის შემოწმება 85
5.5 სტრიქონების ჩაწერა და წაკითხვა: `fgets()` და `fputs()` 86
5.5.1 მაგალითი 2: ფაილის წაკითხვა სტრიქონ-სტრიქონ 86
5.6 ფორმატირებული შეტანა/გამოტანა: `fprintf()` და `fscanf()` 87
5.6.1 მაგალითი 3: ბაზური ტიპების ჩაწერა ფაილში (`fprintf`) 87
5.6.2 მაგალითი 4: ბაზური ტიპების წაკითხვა ფაილიდან (`fscanf`) 88
5.7 ბინარული შეტანა/გამოტანა: `fwrite()` და `fread()` 89
5.8 ფაილში პოზიციონირება: `fseek()`, `ftell()` და `rewind()` 89
5.8.1 მაგალითი 6: ფაილის ზომის დადგენა ბაიტებში 90
5.9 შეჯამება: ოპერაციები ბაზურ ტიპებზე 90
5.10 სავარჯიშოები 91

5.1 ფაილის ცნება და მოდელი C-ში

ფაილი არის მონაცემთა სექცია არაფრენად/მუდმივ მეხსიერებაზე (HDD, SSD და ა.შ.), რომელსაც გააჩნია თავისი სახელი.

ოპერატიული სისტემისთვის ფაილს საკმაოდ რთული სტრუქტურა აქვს. ის შეიცავს ინფორმაციას ფაილის ზომის, შექმნის თარიღის, წვდომის უფლებებისა და ფიზიკური განლაგების შესახებ (დისკზე ფაილი შეიძლება ჩაწერილი იყოს სხვადასხვა ფრაგმენტად).

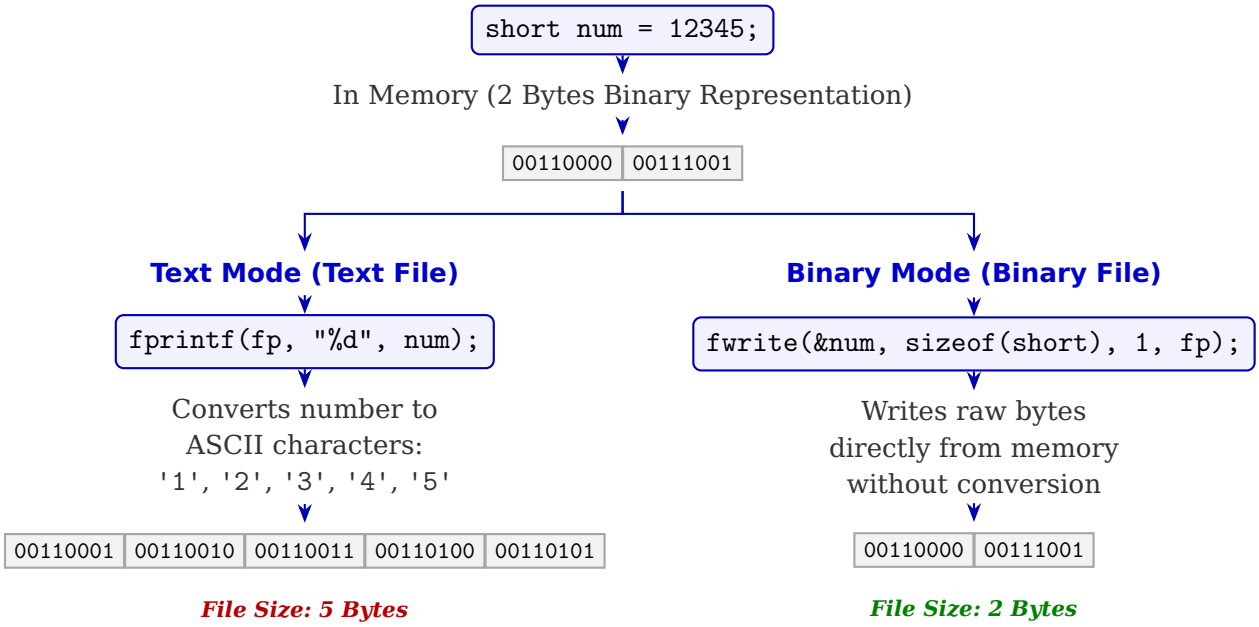
C პროგრამისთვის ფაილი არის *ბაიტების უბრალო მიმდევრობა (ნაკადი — stream)*. თითოეული ბაიტი შეიძლება იქნას წაკითხული ან ჩაწერილი. ეს არის ფაილის მოდელი UNIX სისტემებში, სადაც შეიქმნა C ენა. ვინაიდან სხვა ოპერატიულ სისტემებში შეიძლება ფაილის განსხვავებული მოდელი არსებობდეს, C-ში უზრუნველყოფილია ფაილთან მუშაობის ორი რეჟიმი: *ტექსტური* და *ბინარული*.

5.2 ტექსტური და ბინარული რეჟიმი

ზოგადად, დისკზე ყველა ფაილი ბინარულია, ანუ ჩაწერილია ნულების და ერთიანების მიმდევრობის სახით.

5.2.1 ტექსტურ და ბინარულ ფორმატებს შორის განსხვავება

ტექსტური ფორმატი (ფუნქციები: `fprintf/fscanf`, `fputs/fgets`): წაკითხვადია ადამიანისთვის, პორტაბელურია სხვადასხვა არქიტექტურებს შორის, თუმცა შედარებით ნელია (საჭიროებს რიცხვების ტექსტში კონვერტაციას).
სხვაგვარად, *ტექსტური რეჟიმი* ნიშნავს, რომ ფაილში არსებული ბაიტები წარმოადგენენ სიმბოლოებს (ასოებს, ციფრებს, სასვენ ნიშნებს). ასეთი ფაილის ტექსტურ ედიტორში გახსნისას შინაარსი გამოჩნდება ადამიანისთვის კითხვადი სახით.
ბინარული ფორმატი (ფუნქციები: `fwrite/fread`): ძალიან სწრაფია (RAM მონაცემების პირდაპირი ჩაწერა დისკზე), კომპაქტურია, თუმცა დამოკიდებულია არქიტექტურის Endianness-სა[1] და Struct Alignment-ზე[2].
სხვაგვარად, *ბინარული რეჟიმი* ნიშნავს, რომ ფაილში ინფორმაცია ჩაწერილია ზუსტად ისე, როგორც მას ხედავს ოპერატიული სისტემა ან პროგრამა მეხსიერებაში (მაგალითად, სურათები, შესრულებადი `.exe` ფაილები, აუდიო/ვიდეო).



სურ. 5.1 ბინარული და ტექსტური ფორმატის შედარება

UNIX იყენებს ერთსა და იმავე ფორმატს როგორც ტექსტური, ისე ბინარული ფაილებისთვის. ახალ ხაზზე გადასვლას აღნიშნავს სიმბოლო `'\n'`.

სხვა სისტემებში ფაილებთან მუშაობის განსხვავებული წესები იყო:

[1] **Endianness** — პროცესორის მიერ მეხსიერებაში მრავალბაიტიანი მონაცემების (მაგ. `int`, `float`) ბაიტების შენახვის რიგითობის წესი (*Big-Endian* ან *Little-Endian*). ერთ არქიტექტურაზე (მაგ. x86/ARM) ჩაწერილი ბინარული ფაილის სხვა არქიტექტურაზე წაკითხვისას ბაიტების რიგი შეიძლება აირიოს.

[2] **Struct Alignment (მეხსიერების გასწორება)** — კომპილატორის მიერ სტრუქტურის (`struct`) წევრებს შორის დამატებითი ცარიელი ბაიტების (*padding*) ჩასმა, რათა მონაცემები განთავსდეს პროცესორისთვის ოპტიმალურ მისამართებზე (მაგ. 4 ან 8 ბაიტის ჯერადზე). სხვადასხვა კომპილატორზე ან პლატფორმაზე სტრუქტურის ზომა და ბაიტური განლაგება შეიძლება განსხვავდებოდეს.

- ძველ Macintosh სისტემებში (OS X-მდე) ახალ ხაზზე გადასვლა აღინიშნებოდა `'\r'` სიმბოლოთი.
- MS-DOS და Windows სისტემებში ტექსტურ ფაილებში ახალ ხაზზე გადასვლა აღინიშნება ორი სიმბოლოს კომბინაციით: `"\r\n"`.

იმისათვის, რომ C პროგრამა იყოს პორტაბელური (იმუშაოს ნებისმიერ ოპერატიულ სისტემაზე ერთნაირად), C ბიბლიოთეკა *ტექსტურ რეჟიმში* ავტომატურად ახდენს მოდიფიცირებას:

- Windows-ზე ფაილიდან წაკითხვისას `"\r\n"` ავტომატურად გარდაიქმნება C-სთვის გასაგებ `'\n'` სიმბოლოდ.
- ფაილში ჩაწერისას `'\n'` ავტომატურად ჩაიწერება როგორც `"\r\n"`.

ბინარულ რეჟიმში ასეთი ავტომატური გარდაქმნები არ ხდება — პროგრამა კითხულობს და წერს ბაიტებს ზუსტად ისე, როგორც ისინი რეალურად არიან.

5.2.2 სტანდარტული ნაკადები (Streams)

C პროგრამის გაშვებისას ავტომატურად იხსნება და ხელმისაწვდომია 3 სტანდარტული ნაკადი:

1. `stdin` — სტანდარტული შეტანის ნაკადი (ჩვეულებრივ: კლავიატურა).
2. `stdout` — სტანდარტული გამოტანის ნაკადი (ჩვეულებრივ: ეკრანი/კონსოლი).
3. `stderr` — შეცდომების გამოტანის ნაკადი (ჩვეულებრივ: ეკრანი/კონსოლი).

5.3 ფაილის გახსნა და დახურვა: `fopen` და `fclose`

ფაილთან სამუშაოდ გამოიყენება სპეციალური მიმთითებლის ტიპი `FILE*`. ამ ეტაპზე შეგვიძლია იგი აღვიქვათ როგორც "*ფაილის იდენტიფიკატორი*" ან "*გასაღები*", რომელსაც გვაძლევს `fopen()` ფუნქცია.

ფაილის გახსნის სინტაქსი:

```
1  FILE *fp = fopen("filename.txt", "mode");
```

თუ ფაილის გახსნა ვერ მოხერხდა (მაგალითად, ფაილი არ არსებობს წასაკითხად გახსნისას), `fopen()` აბრუნებს `NULL` მნიშვნელობას. *ყოველთვის აუცილებელია შევამოწმოთ, გახსნა თუ არა ფაილი წარმატებით!*

ცხრილი 5.1 ფაილის გახსნის რეჟიმები (Modes)

რეჟიმი	აღწერა
"r"	ტექსტური ფაილის გახსნა მხოლოდ წასაკითხად. ფაილი უნდა არსებობდეს (თორემ აბრუნებს NULL-ს).

გრძელდება შემდეგ გვერდზე

გრძელდება ცხრილი 5.1

რეჟიმი	აღწერა
"w"	ფაილის გახსნა ჩასაწერად. თუ ფაილი არსებობს, მისი შინაარსი სრულად წაიშლება. თუ არ არსებობს — შეიქმნება ახალი.
"a"	ფაილის გახსნა ბოლოში დამატების რეჟიმში (append). ახალი მონაცემი ჩაიწერება არსებული ტექსტის ბოლოს. თუ ფაილი არ არსებობს — შეიქმნება.
"r+"	გახსნა წასაკითხად და ჩასაწერად (ფაილი უნდა არსებობდეს).
"w+"	გახსნა წასაკითხად და ჩასაწერად (არსებული შინაარსი წაიშლება).
"a+"	გახსნა წასაკითხად და ბოლოში დასამატებლად.
"rb", "wb", "ab"	იგივე რეჟიმები ბინარულ რეჟიმში მუშაობისთვის (ასევე ხელმისაწვდომია "r+b", "w+b", "a+b").

ფაილზე მუშაობის დასრულების შემდეგ აუცილებელია მისი დახურვა `fclose(fp)` ფუნქციით. ეს უზრუნველყოფს მონაცემთა საბოლოო შენახვას დისკზე და ოპერატიული მეხსიერების გათავისუფლებას.

5.4 ასო-ასო ჩაწერა და წაკითხვა: `fgetc()` და `fputc()`

`fputc(c, fp)` — წერს `c` სიმბოლოს `fp` ფაილში.
`fgetc(fp)` — კითხულობს თითო სიმბოლოს `fp` ფაილიდან.

> ***შენიშვნა:*** `fgetc()` აბრუნებს `int` ტიპის მნიშვნელობას და არა `char`-ს! ეს საჭიროა იმისათვის, რომ ფუნქციამ შეძლოს ფაილის დასასრულის კოდის — `EOF` (End Of File, ჩვეულებრივ -1) — დაბრუნება.

5.4.1 მაგალითი 1: სიმბოლოების წაკითხვა და ეკრანზე გამოტანა

განვიხილოთ მარტივი მაგალითი, რომელიც კითხულობს ფაილიდან ტექსტს ასო-ასო და ითვლის სიმბოლოთა რაოდენობას:

კოდი 5.1 ფაილის წაკითხვა და სიმბოლოების დათვლა

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6   int ch; // int ტიპი აუცილებელია EOF-ის სწორად დასაჭერად
7   FILE *fp;
```

```c
 unsigned long count = 0;

 // ფაილის გახსნა მხოლოდ წასაკითხად
 fp = fopen("text.txt", "r");

 // შემოწმება: მოხერხდა თუ არა ფაილის გახსნა
 if (fp == NULL)
 {
  printf("Error: Could not open file text.txt!\n");
  exit(EXIT_FAILURE);
 }

 // ციკლი მუშაობს ფაილის დასასრულამდე (EOF)
 while ((ch = fgetc(fp)) != EOF)
 {
  fputc(ch, stdout); // ბეჭდავს სიმბოლოს ეკრანზე (იგივეა რაც putchar(ch))
  count++;
 }

 // ფაილის დახურვა
 fclose(fp);
 printf("\n\nTotal characters in file: %lu\n", count);

 return 0;
}
```

5.4.2 ფაილის დასასრულის შემოწმება

ფაილის წაკითხვისას გავრცელებული შეცდომაა EOF-ზე შემოწმება წაკითხვის ოპერაციამდე:

```c
// მცდარი მიდგომა: ბოლო სიმბოლო ორჯერ გამოვა ეკრანზე!
while (!feof(fp))
{
 ch = fgetc(fp);
 putchar(ch);
}

// სწორი მიდგომა: ჯერ ვკითხულობთ და იქვე ვადარებთ EOF-ს
while ((ch = fgetc(fp)) != EOF)
{
```

```
    putchar(ch);
}
```

5.5 სტრიქონების ჩაწერა და წაკითხვა: `fgets()` და `fputs()`

მთლიანი სტრიქონების წასაკითხად და ჩასაწერად გამოიყენება `fgets()` და `fputs()` ფუნქციები.

`fputs(str, fp)` — წერს `str` სტრიქონს ფაილში. `puts()`-ისგან განსხვავებით, ავტომატურად **არ ამატებს** ახალ ხაზს (`'\n'`).

`fgets(buffer, nsize, fp)` — კითხულობს მაქსიმუმ `nsize - 1` სიმბოლოს ან ახალი ხაზის (`'\n'`) შეხვედრამდე. ბოლოში ავტომატურად სვამს ნულოვან სიმბოლოს (`'\0'`).

> ***უსაფრთხოების წესი:*** `fgets()` არის C-ში სტრიქონის წაკითხვის ყველაზე უსაფრთხო საშუალება, რადგან იგი ზღუდავს წაკითხული ბაიტების რაოდენობას და იცავს ბუფერს გადავსებისგან (*Buffer Overflow*).

5.5.1 მაგალითი 2: ფაილის წაკითხვა სტრიქონ-სტრიქონ

კოდი 5.2 ფაილის წაკითხვა სტრიქონ-სტრიქონ `fgets`-ის გამოყენებით

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_LINE 256

int main(void)
{
 FILE *fp = fopen("input.txt", "r");
 char buffer[MAX_LINE];
 int line_number = 1;

 if (fp == NULL)
 {
  perror("Error opening file");
  return EXIT_FAILURE;
 }

 // fgets აბრუნებს NULL-ს ფაილის ბოლოს ან შეცდომისას
 while (fgets(buffer, MAX_LINE, fp) != NULL)
 {
  printf("%3d: %s", line_number++, buffer);
 }

```

```c
 fclose(fp);
 return 0;
}
```

5.6 ფორმატირებული შეტანა/გამოტანა: fprintf() და fscanf()

როდესაც გსურთ ფაილში ბაზური მონაცემთა ტიპების (int, float, double, char) ჩაწერა ან წაკითხვა, გამოიყენება fprintf() და fscanf().

ისინი მუშაობენ ზუსტად ისევე, როგორც printf() და scanf(), იმ განსხვავებით, რომ პირველ არგუმენტად გადაეცემათ ფაილის მიმთითებელი (FILE*).

5.6.1 მაგალითი 3: ბაზური ტიპების ჩაწერა ფაილში (fprintf)

შევქმნათ პროგრამა, რომელიც ფაილში ჩაწერს მთელ რიცხვს, ათწილადს და სიმბოლოს:

კოდი 5.3 ბაზური ტიპების ჩაწერა ფაილში

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 FILE *fp;

 // ცვლადები ბაზური ტიპებით
 int id = 101;
 char grade = 'A';
 float score = 94.5f;
 double pi = 3.14159265;

 // ფაილის გახსნა ჩასაწერად
 fp = fopen("results.txt", "w");

 if (fp == NULL)
 {
  printf("Error creating file!\n");
  exit(EXIT_FAILURE);
 }

 // მონაცემთა ფორმატირებული ჩაწერა ფაილში
 fprintf(fp, "ID: %d\n", id);
 fprintf(fp, "Grade: %c\n", grade);
```

```
26  fprintf(fp, "Score: %.2f\n", score);
27  fprintf(fp, "Pi: %.8lf\n", pi);
28
29  // ფაილის დახურვა
30  fclose(fp);
31
32  printf("Data successfully written to results.txt\n");
33
34  return 0;
35 }
```

5.6.2 მაგალითი 4: ბაზური ტიპების წაკითხვა ფაილიდან (fscanf)

ახლა განვიხილოთ შემთხვევა, როდესაც ფაილში ინახება მხოლოდ რიცხვითი და სიმბოლური მონაცემები (მაგალითად: `105 B 88.2`) და გვსურს მათი წაკითხვა შესაბამის ცვლადებში:

კოდი 5.4 ბაზური ტიპების წაკითხვა ფაილიდან

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6   FILE *fp;
7
8   // ცვლადები წაკითხული მნიშვნელობების შესანახად
9   int id;
10  char grade;
11  float score;
12
13  // ფაილის გახსნა წასაკითხად
14  fp = fopen("data.txt", "r");
15
16  if (fp == NULL)
17  {
18   printf("Error: File data.txt not found!\n");
19   exit(EXIT_FAILURE);
20  }
21
22  // fscanf აბრუნებს წარმატებით წაკითხული ელემენტების რაოდენობას
23  // ველოდებით 3 ელემენტის წაკითხვას (int, char, float)
```

```c
 if (fscanf(fp, "%d %c %f", &id, &grade, &score) == 3)
 {
  printf("Read data:\n");
  printf("ID: %d\n", id);
  printf("Grade: %c\n", grade);
  printf("Score: %.1f\n", score);
 }
 else
 {
  printf("Error: Could not read data from file or invalid format.\n")
  ↪ ;
 }

 // ფაილის დახურვა
 fclose(fp);

 return 0;
}
```

5.7 ბინარული შეტანა/გამოტანა: `fwrite()` და `fread()`

ბინარულ რეჟიმში მეხსიერების ბლოკები (სტრუქტურები, მასივები) პირდაპირ გადადის დისკზე და პირიქით.

`fwrite(ptr, size, count, fp)` — იწერება `count` რაოდენობის ელემენტი, თითოეული `size` ბაიტის ზომით `ptr` მისამართიდან.

`fread(ptr, size, count, fp)` — იკითხება `count` რაოდენობის ელემენტი `ptr` ბუფერში.

5.8 ფაილში პოზიციონირება: `fseek()`, `ftell()` და `rewind()`

ფაილის კითხვის/წერის დროს ოპერაციული სისტემა ინახავს მიმდინარე პოზიციის მიმთითებელს (File Position Indicator).

`ftell(fp)` — აბრუნებს მიმდინარე პოზიციის ინდექსს ბაიტებში (ტიპი `long`).

`fseek(fp, offset, whence)` — გადააქვს პოზიცია მითითებულ ადგილას:

- `SEEK_SET` — ფაილის დასაწყისიდან;
- `SEEK_CUR` — მიმდინარე პოზიციიდან;
- `SEEK_END` — ფაილის ბოლოდან.

`rewind(fp)` — აბრუნებს მიმთითებელს ფაილის დასაწყისში (იგივეა, რაც `fseek(fp, 0, SEEK_SET)`).

5.8.1 მაგალითი 6: ფაილის ზომის დადგენა ბაიტებში

```c
#include <stdio.h>

int main(void)
{
 FILE *fp = fopen("input.txt", "rb");
 if (fp == NULL) return 1;

 // გადავდივართ ფაილის ბოლოში
 fseek(fp, 0, SEEK_END);

 // ვკითხულობთ მიმდინარე პოზიციას (ანუ ფაილის ზომას)
 long size = ftell(fp);

 printf("File size: %ld bytes\n", size);

 fclose(fp);
 return 0;
}
```

5.9 შეჯამება: ოპერაციები ბაზურ ტიპებზე

ცხრილი 5.2 ჩაწერა/წაკითხვა

ოპერაცია	კონსოლი (ეკრანი/კლავიატურა)	ფაილი
char ჩაწერა	`putchar(c)` / `putc(c, stdout)`	`fputc(c, fp)` / `putc(c, fp)`
char წაკითხვა	`c = getchar()`	`c = fgetc(fp)` / `c = getc(fp)`
სტრიქონის ჩაწერა	`puts(str)`	`fputs(str, fp)`
სტრიქონის წაკითხვა	`fgets(str, n, stdin)`	`fgets(str, n, fp)`
ფორმ. ჩაწერა	`printf("...", a, b)`	`fprintf(fp, "...", a, b)`
ფორმ. წაკითხვა	`scanf("...", &a, &b)`	`fscanf(fp, "...", &a, &b)`
ბინარული ჩაწერა/წაკითხვა	N/A	`fwrite()` / `fread()`

5.10 სავარჯიშოები

დავალება 5.1: ფაილების კოპირება

დაწერეთ C პროგრამა, რომელიც მიიღებს ფაილის ორ სახელს (წყაროს და სამიზნეს) და დააკოპირებს წყარო ფაილს ბინარულ რეჟიმში. შეამოწმეთ ფაილის გახსნის შეცდომები.

დავალება 5.2: სტრიქონებისა და სიტყვების დათვლა

დაწერეთ პროგრამა, რომელიც წაიკითხავს ტექსტურ ფაილს და დაითვლის მასში:

1. სტრიქონების რაოდენობას;
2. სიტყვების საერთო რაოდენობას (სიტყვები გამოყოფილია ინტერვალებით ან ახალი ხაზით).

დავალება 5.3: სტუდენტების მონაცემთა ბაზა

შექმენით პროგრამა, რომელიც ბინარულ ფაილში ჩაწერს N რაოდენობის სტუდენტის მონაცემებს (ID, სახელი, ქულა), ხოლო შემდეგ საშუალებას მისცემს მომხმარებელს, `fseek()`-ის გამოყენებით წაიკითხოს კერძოდ i-ური სტუდენტის მონაცემი მთლიანი ფაილის თავიდან ბოლომდე წაკითხვის გარეშე.

თავი 6

მასივი

6.1 ერთგანზომილებიანი მასივი 94

6.1.1 ერთგანზომილებიანი მასივის ინიციალიზაცია 94

6.1.2 დანიშნული ინიციალიზატორები (Designated Initializers) და C23-ის სიახლეები 96

6.1.3 მასივის ელემენტებზე მიმართვა 97

6.1.4 მიმთითებელი და მასივი 99

6.1.5 მაგალითი: მთვლელების მასივი 102

6.1.6 მაგალითი: მთელი მონაცემების ჰისტოგრამა 103

6.1.7 მაგალითი: შემთხვევითი რიცხვების გენერატორი 104

6.1.8 მაგალითი: ფიბონაჩის რიცხვები 105

6.1.9 მაგალითი: რიცხვი სხვადასხვა სისტემაში 106

6.2 სიმბოლოების მასივი (სტრიქონები / Strings) 107

6.3 სიმბოლოების მასივი `'0'`-ის გარეშე 110

6.3.1 სინტაქსის დეტალური განმარტება 110

6.4 სტრიქონის წაკითხვა ფაილიდან 112

6.5 მასივის გადაცემა ფუნქციაში 112

6.5.1 მაგალითი: ერთგანზომილებიანი მასივის გადაცემა 113

6.5.2 მაგალითი: მასივის დალაგება 116

6.6 მრავალგანზომილებიანი მასივები 117

6.6.1 მრავალგანზომილებიანი მასივის ინიციალიზაცია 118

6.6.2 მაგალითი: ორგანზომილებიანი მასივების შეკრება 119

6.6.3 2D მასივის გადაცემა ფუნქციაში 120

6.6.4 C99-მდე არსებული ალტერნატიული მეთოდები 2D მასივის გადასაცემად 122

6.7 1D მასივის ჩაწერა/წაკითხვა ფაილიდან 125

6.8 სტატიკური (სტეკ) 3D მასივი 126

6.9 ცვლადი ზომის მასივები (VLA) C99/C11 სტანდარტებში 128

6.9.1 VLA-ის ისტორია და C11 ცვლილებები 128

6.9.2 რატომ გახდა VLA არასავალდებულო? 129

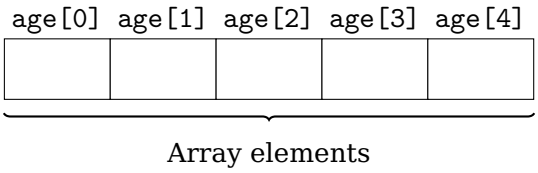
6.9.3 VLA მხარდაჭერის შემოწმება კოდში 129
6.10 ბრძანებათა სტრიქონის არგუმენტები: argc და argv 129
6.10.1 პრაქტიკული მაგალითები: ბრძანებათა სტრიქონის არგუმენტები . 130
6.11 სავარჯიშოები 133
6.12 2D და 3D სტეკ მასივების ჩაწერა და წაკითხვა 134
6.13 მასივის ფორმატირებული ჩაწერა/წაკითხვა 136
6.13.1 კოდის დეტალური განმარტება ხაზების მიხედვით 139
6.13.2 წაკითხვა სიმბოლომდე — %99[^=] 140
6.13.3 \r\n და კროსპლატფორმულობა fscanf()-ში 141
6.14 სავარჯიშოები 142

6.1 ერთგანზომილებიანი მასივი

C ენაში შეიძლება განმარტებულ იქნას ნებისმიერი ტიპის მასივი. სინტაქსი არის შემდეგი:

```
type name[dim];
```

C ენაში მასივი იწყება ნულოვანი პოზიციიდან (ანუ ელემენტების ინდექსირება იწყება ნულიდან). მასივის ელემენტები განლაგებულია მეხსიერებაში მიმდევრობით. მაგალითად, თუ გვაქვს int ელემენტების მასივი და რომელიმე i-ური ელემენტის მისამართია adr_i, i+1 ელემენტის მისამართი იქნება adr_i2 = adr_i + sizeof(int). მაგალითად, თუ გვაქვს int age[5];, მაშინ ეს მასივი მეხსიერებაში იქნება შემდეგი სახით:



სურ. 6.1 მასივის ელემენტების განლაგება მეხსიერებაში

მასივის ზომა იქნება 5 × sizeof(int) ბაიტი. age[0] იქნება პირველი ელემენტი, age[1] — მეორე და ა.შ. ვთქვათ, age[0]-ის მისამართია 2120d და int-ის ზომაა 4 ბაიტი, მაშინ შემდეგი ელემენტის (age[1]) მისამართი იქნება 2124d, age[2] — 2128d და ა.შ.

6.1.1 ერთგანზომილებიანი მასივის ინიციალიზაცია

მასივი შეიძლება ინიციალიზირებულ იქნას დეკლარირებისთანავე:

```
int age[5] = {2, 4, 34, 3, 4};
```

არაა აუცილებელი მასივის ზომის მითითება, თუ ვაკეთებთ ინიციალიზაციას:

```
int myArray[] = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }; // სწორია
```

ელემენტები, რომლებიც არ არის მითითებული, ინიციალიზდება როგორც 0:

```
int myArray[10] = { 1, 2 }; // 1, 2, 0, 0, 0...
```

ყველა ელემენტი ინიციალიზდება როგორც 0:

```
int myArray[10] = { 0 }; // ყველა ელემენტი ნულია
```

მუშაობს C++-ში, მაგრამ არ არის დაშვებული C-ში (C23 სტანდარტამდე):

```
int myArray[10] = {}; // ყველა ელემენტი ნულია (C++)
```

სტატიკური ობიექტები ავტომატურად ინიციალიზდება როგორც 0:

```
static int myArray[10]; // ყველა ელემენტი ნულია
```

```
int age[] = {2, 4, 34, 3, 4};
```

ამ შემთხვევაში კომპილატორი თავად გამოთვლის მასივის ზომას.
char მასივის ინიციალიზაციის მაგალითი:

```
char string[5] = "Hello";
```

ან

```
char letters[5] = { 'H', 'e', 'l', 'l', 'o' };
```

დავალება 6.1: char მასივი

შექმენით C პროექტი და ნახეთ, რა განსხვავებაა ქვემოთ მოყვანილ ცვლადებს შორის (დებაგირებისას 'Watch' ფანჯარაში ნახეთ ცვლადების მნიშვნელობები):

```c
#include <stdio.h>

int main(void)
{
 char str1[5] = "Hello";
 char str2[6] = "Hello";
 char letters[5] = {'H', 'e', 'l', 'l', 'o'};
 return 0;
}
```

მასივის ინიციალიზაციისას ინიციალიზებული ელემენტების რაოდენობა არ უნდა აღემატებოდეს მასივის ზომას. თუ ინიციალიზატორების რაოდენობა მასივის ზომაზე ნაკლებია, დანარჩენი ელემენტები ინიციალიზდება როგორც 0.

6.1.2 დანიშნული ინიციალიზატორები (Designated Initializers) და C23-ის სიახლეები

C99 სტანდარტიდან მოყოლებული C ენას დაემატა **დანიშნული ინიციალიზატორები** (*Designated Initializers*) მასივების, სტრუქტურებისა და გაერთიანებებისათვის (*Unions*).

1. მასივების დანიშნული ინიციალიზაცია (C99)

შესაძლებელია მასივის კონკრეტული ინდექსების მიხედვით მნიშვნელობების მინიჭება:

```c
int a[6] = { [4] = 29, [2] = 15 };
/* იგივეა, რაც: int a[6] = { 0, 0, 15, 0, 29, 0 }; */
```

შენიშვნა დიაპაზონებზე: სინტაქსი `int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2 };` წარმოადგენს **GNU GCC კომპილატორის გაფართოებას** (*GNU C Extension*) და არ არის C99/C11/C23 სტანდარტი, რის გამოც ის არ კომპილირდება სტანდარტულ MSVC ან Clang კომპილატორებში.

2. სტრუქტურების და მასივების სტრუქტურების ინიციალიზაცია (C99)

სტრუქტურის ინიციალიზაციისას მიეთითება **.ველის_სახელი = მნიშვნელობა**. ველების მიმდევრობას მნიშვნელობა არ აქვს:

```c
struct point { int x, y; };

/* C99 დანიშნული ინიციალიზაცია: */
struct point p = { .y = yvalue, .x = xvalue };

```

```
6  /* სტრუქტურების მასივის ინიციალიზაცია: */
7  struct point ptarray[10] = {
8   [2].y = yv2,
9   [2].x = xv2,
10  [0].x = xv0
11 };
```

კომპილატორების თავსებადობა: აღნიშნული სინტაქსი არ მუშაობდა Visual Studio 2010-ში, დაემატა Visual Studio 2013-ში, ხოლო თანამედროვე კომპილატორებში (GCC, Clang, MSVC 2019/2022) სრულად არის მხარდაჭერილი.

C17 და C23 სტანდარტის სიახლეები ინიციალიზაციაში

C17 სტანდარტი: C17 იყო C11-ის შეცდომების შესწორების გამოცემა და მასში ინიციალიზაციის ახალი სინტაქსური ელემენტები არ დამატებულა.
C23 ცარიელი ინიციალიზაცია ({ }): C99/C11/C17 სტანდარტებში სტრუქტურის ან მასივის ნულებით ინიციალიზაციისთვის აუცილებელი იყო მცირე ფორმალობა {0} (რადგან ცარიელი {} სინტაქსური შეცდომა იყო სტანდარტულ C-ში). **C23 სტანდარტში ოფიციალურად დაემატა ცარიელი ინიციალიზატორი {}**, რომელიც ობიექტის ყველა ველს ავტომატურად ანულებს:

```
struct point p = {}; /* C23: ანულებს x-ს და y-ს (C17-მდე იწერებოდა
  ↪ {0}) */
int arr[10] = {};    /* C23: მასივის ყველა 10 ელემენტს ანიჭებს 0-ს */
```

C23 auto ტიპის დედუქცია ინიციალიზაციისას: C23-ში დაემატა auto საკვანძო სიტყვით ტიპის ავტომატური განსაზღვრა ინიციალიზატორის მიხედვით:

```
auto p = (struct point){ .x = 10, .y = 20 }; /* auto ავტომატურად გან
  ↪ საზღვრავს struct point ტიპს */
```

C23 constexpr ინიციალიზაცია: C23-ში შესაძლებელია ცვლადების გამოცხადება constexpr მოდიფიკატორით, რომლებიც კომპილაციის ეტაპზევე ინიციალიზდებიან მუდმივი გამოსახულებებით.

6.1.3 მასივის ელემენტებზე მიმართვა

მასივის ელემენტი განიხილება როგორც ნებისმიერი ჩვეულებრივი ცვლადი. მაგალითად:
ცვლადის წაკითხვა და მინიჭება მასივის მესამე ელემენტზე: `scanf("%d", &age[2]);`
ცვლადის წაკითხვა და მინიჭება მასივის i+1 ელემენტზე (ვინაიდან მასივის პირველი ელემენტი, ნულოვანი ინდექსით, არის age[0]): `scanf("%d", &age[i]);`
პირველი ელემენტის ბეჭდვა: `printf("%d", age[0]);`

i+1 ელემენტის ბეჭდვა: `printf("%d", age[i]);`

განვიხილოთ მაგალითი:

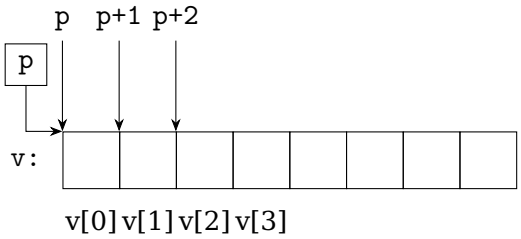
```
#include <stdio.h>

int main(void)
{
 int marks[10], i, n, sum = 0;
 printf("shemoitanet studentebis raodenoba: ");
 scanf("%d", &n); // cakitxva
 for (i = 0; i < n; ++i)
 {
  printf("shemoitanet nishnebi %d: ", i + 1);
  scanf("%d", &marks[i]); // cakitxva da masivis elementze minicheba
  sum += marks[i];        // shekreba
 }
 printf("Jami = %d\n", sum);
 return 0;
}
```

ქვემოთ მოყვანილ კოდში პროგრამა კითხულობს მომხმარებლის მიერ შეყვანილ ხაზს და ასამდე სიმბოლოს ინახავს სიმბოლოების მასივში `char line[100];`:

```
#include <stdio.h>

int main(void)
{
 int n = 0, c;
 char line[100];

 while ((c = getchar()) != '\n' && c != EOF)
 {
  if (n < 100)
  line[n] = c;
  n++;
 }
 printf("length = %d\n", n);
 return 0;
}
```

6.1.4 მიმთითებელი და მასივი

C ენა აღიქვამს მასივის სახელს ისე, როგორც მასივის პირველი ელემენტის მიმთითებელს. ეს მნიშვნელოვანია მასივებში მიმთითებლების არითმეტიკის სწორი გაგებისათვის.



სურ. 6.2 ერთგანზომილებიან მასივსა და მიმთითებელს შორის კავშირი

ანუ თუ v არის მასივი, *v იგივეა, რაც v[0], *(v+1) იგივეა, რაც v[1] და ა.შ. განვიხილოთ მიმთითებლის მასივში გამოყენების მაგალითი:

```c
#include <stdio.h>

#define SIZE 3

int main(int argc, char **argv)
{
 float x[SIZE];
 float *p;
 int i;

 // initialize the array x
 for (i = 0; i < SIZE; i++)
 x[i] = 0.5f * (float)i;

 // print x
 for (i = 0; i < SIZE; i++)
 printf("  %d  %f \n", i, x[i]);

 // make p point to array x
 p = x;

 // print via pointer arithmetic
 for (i = 0; i < SIZE; i++)
 printf("  %d  %f \n", i, *(p + i));

 return 0;
```

```
27  }
```

ვინაიდან `x[i]` ნიშნავს `x` მასივის i-ურ ელემენტს, ხოლო `p = x` უთითებს `x` მასივის დასაწყისზე, მაშინ `*(p+i)` არის იმ ელემენტის მნიშვნელობა, რომელიც i მანძილითაა დაშორებული ელემენტს, რომელზეც უთითებს `p`; ეს ელემენტი კი არის `x[i]`.

მნიშვნელოვანია გვახსოვდეს, რომ მასივის ელემენტებზე მიმართვა უნდა მოხდეს მასივის ზომის ფარგლებში. მაგალითად, თუ გვაქვს 10-ელემენტიანი მასივი `arr[10]`, შეგვიძლია მივმართოთ მასივის ელემენტებს `arr[0]`-დან `arr[9]`-ის ჩათვლით. მაგრამ მიმართვა `arr[10]`, `arr[13]` და ა.შ. იქნება შეცდომა (Out of bounds). კომპილატორმა შეიძლება არ აჩვენოს შეცდომა კოდის კომპილირებისას, მაგრამ პროგრამის გაშვებისას მივიღებთ გაურკვეველ ქცევას (Undefined Behavior).

მასივის ინდექსად შეიძლება მიეწოდოს გამოსახულება, რომელიც აბრუნებს მთელ მნიშვნელობას.

```
1   #include <stdio.h>
2
3   #define SIZE 10
4
5   int main(int argc, char **argv)
6   {
7    double x[SIZE] = {0};
8    int n = 7;
9    int i;
10
11   x[n / 2] = 10.0;
12   x[2] = 4.0;
13   x[7] = 3.5;
14   x[8] = x[2] + x[7];
15
16   for (i = 0; i < SIZE; i++)
17   {
18    printf("x[%d] = %lf\n", i, x[i]);
19   }
20
21   return 0;
22  }
```

ტიპი `size_t` განმარტებულია C ბიბლიოთეკის ფაილში `<stddef.h>` (ეს ფაილი, როგორც წესი, ჩართულია `<stdio.h>`-ში და საჭირო არაა მისი ცალკე მითითება). რადგან მასივის ზომა და ინდექსი არ შეიძლება იყოს 0-ზე ნაკლები, სწორი პრაქტიკაა გამოყენებულ იქნას ტიპი `size_t`, მით უმეტეს, თუ გვსურს კოდმა იდეალურად იმუშაოს 64-ბიტიან პლატფორმაზე.

ასევე, როცა მასივის ზომა მუდმივია და ცნობილია, კოდში რიცხვის პირდაპირ წერის (Hardcoding) ნაცვლად უმჯობესია განმარტებულ იქნას ზომა `#define` ბრძანების საშუალებით(ან, გაცილებით უკეთესია, `constexpr` საშუალებით, რის შესახებაც ადრე გვქონდა საუბარი).

```
1  #include <stdio.h>
2
3  #define SIZE 40
4
5  int main(int argc, char **argv)
6  {
7   double n[SIZE]; // n is an array of 40 doubles
8   size_t i;       // counter
9   double sum = 0.0;
10
11  // initialize elements of array n
12  for (i = 0; i < SIZE; ++i)
13  {
14   n[i] = i * i * 10 / 2.47;
15   sum += n[i];
16  }
17
18  printf("%s%13s\n", "Element", "Value");
19
20  // output contents of array n in tabular format
21  for (i = 0; i < SIZE; ++i)
22  {
23   printf("%7zu%13f\n", i, n[i]);
24  }
25
26  printf("%20s\n", "Array Average");
27  printf("%7d%13f\n", SIZE, sum / SIZE);
28
29  return 0;
30 }
```

`#define SIZE 40` განმარტავს სიმბოლურ მუდმივას `SIZE`, რომლის მნიშვნელობა არის 40.

ზემოთ მოყვანილ კოდში არის კიდევ ერთი სიახლე — ე.წ. ფორმატირებული ბეჭდვა: `%13s` მე-18 ხაზზე ნიშნავს, რომ გამოყოფილია 13 პოზიცია (ითვლება მარჯვნიდან მარცხნივ). შესაბამისად, `%7zu%13f` ნიშნავს, რომ გამოყოფილია 7 პოზიცია უნიშნო `size_t` ცვლადის-

თვის და 13 პოზიცია მცოცავწერტილიანი ცვლადისთვის.

6.1.5 მაგალითი: მთვლელების მასივი

ვთქვათ, ვატარებთ გამოკითხვას მოცემული ათი შოუდან, რომელი უფრო მოსწონს მაყურებელს. ამ ამოცანის მარტივად გადაწყვეტა შეიძლება შემდეგი გზით: შემოგვაქვს ე.წ. მთვლელების 11-ელემენტიანი მასივი. მასივის ინდექსი შეესაბამება შოუს ინდექსს. რომელი შოუც მოსწონს მაყურებელს, შესაბამის მასივის ელემენტს ვზრდით ერთი ერთეულით.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5   int ratingCounters[11], i, response;
6
7   for (i = 1; i <= 10; ++i)
8   ratingCounters[i] = 0;
9
10  printf("shemoitanet shous nomeri\n");
11
12  for (i = 1; i <= 20; ++i)
13  {
14   scanf("%i", &response);
15   if (response < 1 || response > 10)
16   printf("aseti nomeri ar arsebobs: %i\n", response);
17   else
18   ++ratingCounters[response];
19  }
20
21  printf("\n\nRating    gamokitxulta ricxvi\n");
22  printf("------ -------------------\n");
23
24  for (i = 1; i <= 10; ++i)
25  printf("%4i%14i\n", i, ratingCounters[i]);
26
27  return 0;
28 }
```

ყურადღება მიაქციეთ შემდეგს:

1. გამოკითხულთა რაოდენობაა 20.
2. მიუხედავად იმისა, რომ გვაქვს 10 შოუ, შემოტანილ იქნა 11-ელემენტიანი მასივი.
3. ციკლში ხდება მასივის 1–10 ელემენტების ინიციალიზაცია (ნულებით შევსება).

4. შემდეგ ციკლში 20-ჯერ ხდება შოუს ნომრის წაკითხვა კონსოლიდან.
5. პირობით მოწმდება შოუს ნომერი: თუ ის არის საზღვრებში $[1, 10]$, იზრდება შესაბამისი მთვლელის მნიშვნელობა ერთი ერთეულით, წინააღმდეგ შემთხვევაში გამოდის შეტყობინება შეცდომის შესახებ.
6. ბოლო ციკლში ხდება მიღებული შედეგების ბეჭდვა. აქ აღსანიშნავია ის, რომ არ ვიყენებთ მასივის ნულოვან (0) ელემენტს. ეს ელემენტი არც კი ყოფილა ინიციალიზებული.

დავალება 6.2: მასივის ინდექსირება

შეცვალეთ ზემოთ მოყვანილი კოდი ისე, რომ მასივში იყოს ზუსტად 10 ელემენტი და ინდექსირება იწყებოდეს ნულიდან, მაგრამ ისე, რომ ე.წ. შოუს ნომრები კვლავ დარჩეს 1, 2, 3. . . , ანუ მომხმარებელს კვლავ შეჰყავს 1, 2, 3. . . და ა.შ., მაგრამ მასივში ჩალაგება მოხდეს 0-იდან დაწყებული ინდექსებით.

6.1.6 მაგალითი: მთელი მონაცემების ჰისტოგრამა

კონსოლურ პროგრამებში ხშირად ნახავთ `'*'` ან სხვა ფსევდოგრაფიკულ სიმბოლოებს.

```
1  // Fig. 6.8: fig06_08.c
2  // Displaying a histogram.
3  #include <stdio.h>
4
5  #define SIZE 10
6
7  // function main begins program execution
8  int main(int argc, char **argv)
9  {
10  // use initializer list to initialize array n
11  int n[SIZE] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
12  size_t i; // outer for counter for array elements
13  int j;    // inner for counter counts *s in each histogram bar
14
15  printf("%s%13s%17s\n", "Element", "Value", "Histogram");
16
17  // for each element of array n, output a bar of the histogram
18  for (i = 0; i < SIZE; ++i) {
19   printf("%7zu%13d          ", i, n[i]);
20   for (j = 1; j <= n[i]; ++j) { // print one bar
21    printf("%c", '*');
22   } // end inner for
23   puts(""); // end a histogram bar
24  } // end outer for
```

```
25
26   return 0;
27  } // end main
```

ზემოთ მოყვანილ კოდში გვაქვს ორმაგი ციკლი. პირველ ციკლში გავდივართ მასივის ელემენტებს, ხოლო მეორე ციკლში ყოველი ელემენტისთვის ვბეჭდავთ იმდენ '*' სიმბოლოს, რა მნიშვნელობაც აქვს მასივის მოცემულ ელემენტს.

დავალება 6.3: რეიტინგების შოუ და ჰისტოგრამა

„შოუს რეიტინგების“ კოდში ჩაამატეთ რეიტინგების ჰისტოგრამის ბეჭდვა.

6.1.7 მაგალითი: შემთხვევითი რიცხვების გენერატორი

ქვემოთ მოყვანილი კოდით ვამოწმებთ, რამდენად სწორად მუშაობს შემთხვევითი რიცხვების გენერატორი. ამოცანა არის ასეთი: წარმოვიდგინოთ, რომ ვაგორებთ კამათელს. კამათელს აქვს 6 წახნაგი. კამათელს ვაგორებთ 6 000 000-ჯერ. ყველა წახნაგის მოსვლის ალბათობა ერთი და იგივეა, ამიტომ შემთხვევითი რიცხვების გენერატორმა თითოეული წახნაგისთვის უნდა მოგვცეს დაახლოებით ტოლი რიცხვები.

```
1  // Fig. 6.9: fig06_09.c
2  // Roll a six-sided die 6,000,000 times
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  #define SIZE 7
8
9  // function main begins program execution
10 int main(void)
11 {
12  size_t face;          // random die value 1 - 6
13  unsigned int roll;    // roll counter 1 - 6,000,000
14  unsigned int frequency[SIZE] = { 0 }; // clear counts
15
16  srand((unsigned int)time(NULL)); // seed random number generator
17
18  // gagoreba xdeba 6,000,000-jer
19  for (roll = 1; roll <= 6000000; ++roll)
20  {
21   face = 1 + rand() % 6;
22   ++frequency[face]; // vzrdit mocemuli caxnagis mosvlis raodenobas
```

```
 }

 printf("%s%17s\n", "Face", "Frequency"); // bechdva

 // output frequency elements 1-6 in tabular format
 for (face = 1; face < SIZE; ++face)
 {
  printf("%4zu%17u\n", face, frequency[face]);
 }

 return 0;
} // end main
```

6.1.8 მაგალითი: ფიბოწაჩის რიცხვები

ვიღებთ საწყის ორ რიცხვს და ყოველი შემდეგი რიცხვი მიიღება წინა ორის შეკრებით: $F_i = F_{i-1} + F_{i-2}$. ქვემოთ მოყვანილი კოდი შეესაბამება ამოცანას საწყისი რიცხვებით 0 და 1.

```
#include <stdio.h>

#define SIZE 15

int main(int argc, char **argv)
{
 int Fibonacci[SIZE], i;

 Fibonacci[0] = 0; // by definition
 Fibonacci[1] = 1; // ditto

 for (i = 2; i < SIZE; ++i)
 Fibonacci[i] = Fibonacci[i - 2] + Fibonacci[i - 1];

 for (i = 0; i < SIZE; ++i)
 printf("%i\n", Fibonacci[i]);

 return 0;
}
```

ზემოთ მოყვანილი კოდი შეესაბამება შემდეგ ამოცანას: ვთქვათ, გვაქვს კურდღლების ერთი წყვილი. ყოველი წყვილი თვის ბოლოს წარმოშობს ახალ წყვილს. კურდღლები არ

იხოცებიან.

დავალება 6.4: ფიბონაჩის რიცხვები

რამდენი წყვილი კურდღელი იქნება მეათე თვის ბოლოს? მეთორმეტე თვის ბოლოს?

6.1.9 მაგალითი: რიცხვი სხვადასხვა სისტემაში

ქვემოთ მოყვანილ პროგრამას ათობითში შეყვანილი რიცხვი გადაჰყავს სხვა თვლის სისტემაში (ორობითი, რვაობითი, თექვსმეტობითი).

```
#include <stdio.h>

int main(void)
{
 const char baseDigits[16] = {
  '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
 };
 int convertedNumber[64];
 long int numberToConvert;
 int nextDigit, base, index = 0;

 // get the number and the base
 printf("Number to be converted? ");
 scanf("%ld", &numberToConvert);
 printf("Base? ");
 scanf("%i", &base);

 // convert to the indicated base
 do {
  convertedNumber[index] = numberToConvert % base;
  ++index;
  numberToConvert = numberToConvert / base;
 } while (numberToConvert != 0);

 // display the results in reverse order
 printf("Converted number = ");
 for (--index; index >= 0; --index) {
  nextDigit = convertedNumber[index];
  printf("%c", baseDigits[nextDigit]);
 }
```

```
32   printf("\n");
33
34   return 0;
35  }
```

აქ ისევ შეგვხვდა საკვანძო სიტყვა `const`, რაც ნიშნავს, რომ გვაქვს მუდმივების მასივი. ანუ ამ მასივის ელემენტების მნიშვნელობის შეცვლა პროგრამის მიერ აკრძალულია[1].

დავალება 6.5: თვლის სისტემები

გაარჩიეთ ზემოთ მოყვანილი კოდი. აღწერეთ პროგრამის ყოველი ნაბიჯი (რა კეთდება თითოეულ ხაზზე) და უპასუხეთ შემდეგ კითხვებს:

1. რა ინახება მასივში `convertedNumber` და როგორ? რას ნიშნავს 64 ჩანაწერი `convertedNumber[64]`?
2. რატომ იქნა შემოტანილი ტიპი `long int`?
3. რა ხდება ციკლის შემდგომ `--index` ბრძანებაში?
4. რას ნიშნავს ცვლადი `index` და რატომ გახდა მისი შემცირება საჭირო ბეჭდვამდე?
5. შეიძლება თუ არა პროგრამამ იმუშაოს `nextDigit` ცვლადის გარეშეც? რატომ და როგორ?

შეცვალეთ პროგრამა ისე, რომ შემოწმდეს, თუ მომხმარებელმა შეიტანა ბაზად რიცხვი > 16, მისცეს მას ახალი ბაზის შეყვანის შესაძლებლობა.

6.2 სიმბოლოების მასივი (სტრიქონები / Strings)

სიმბოლოებს ვიყენებდით `printf` ფუნქციაში. ზემოთ რამდენჯერმე შეგვხვდა სიმბოლოების მასივი. სიმბოლოების მიმდევრობისათვის C++ სტანდარტულ ბიბლიოთეკაში არის კლასი `std::string`. C ენაში ცალკე ასეთი მონაცემთა ტიპი არ არსებობს. „სტრიქონი“ (*string*) C ენაში არის სიმბოლოების სპეციალური მასივი.

აღსანიშნავია, რომ სიმბოლოების მასივი ჯერ კიდევ არ ნიშნავს, რომ გვაქვს C-სტრიქონი. მაგალითად:

```
char string1[] = "first";
```

ეს ბრძანება ქმნის მასივს, რომლის ელემენტებია სიმბოლოები + ე.წ. ნულ-სიმბოლო (*Null terminator*) `'\0'`:

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

ასე რომ, მასივი:

[1] თუმცა `const` ჯერ კიდევ არ ნიშნავს, რომ ასეთი ტიპის მნიშვნელობის შეცვლა აბსოლუტურად შეუძლებელია hardware დონეზე — ის შეიძლება შეიცვალოს „გარედან“ რაიმე მოწყობილობის მიერ, მაგალითად, მოწყობილობა, რომელიც ზომავს ძაბვას ან ტემპერატურას და პროგრამას „უგზავნის“ გაზომილ მნიშვნელობებს

```
char string2[] = { 'f', 'i', 'r', 's', 't' };
```

არ არის იგივე, რაც `string1`. C-სტრიქონი არის სწორედ `string1` და არა `string2`. `'\0'` სიმბოლო საჭიროა იმისათვის, რომ C-ს ფუნქციებმა, რომლებიც ასრულებენ ოპერაციებს სტრიქონებზე, იცოდნენ, სად მთავრდება ტექსტი. C-ში სტრიქონებზე მანიპულაცია შეიძლება მხოლოდ როგორც მასივის ელემენტებზე — ან მიმთითებლების საშუალებით, ან სპეციალური ფუნქციების საშუალებით სტანდარტული ბიბლიოთეკიდან `<string.h>`.

შეგვიძლია სტრიქონი წავიკითხოთ `scanf` ფუნქციის საშუალებით:

```
char string2[20];
scanf("%19s", string2);
```

ზემოთ მოყვანილ კოდში არის ერთი სიახლე, რაც არ გვიხსენებია `scanf` ფუნქციის განხილვისას — ესაა `%19s`, რაც ნიშნავს, რომ წაკითხულ უნდა იქნას მაქსიმუმ 19 სიმბოლო. ეს საჭიროა მასივის უსაფრთხოდ წასაკითხად. ყოველ მასივს აქვს ფიქსირებული ზომა და იმისათვის, რომ არ მოხდეს მასივის გადავსება (ბუფერის გადავსება / Buffer Overflow), უნდა მიეთითოს, თუ რამდენი სიმბოლო უნდა იქნას წაკითხული. რადგან გამოცხადებულია 20-სიმბოლიანი მასივი, წაკითხულ უნდა იქნას მაქსიმუმ 19 სიმბოლო, ვინაიდან მე-20 უნდა იყოს `'\0'` სიმბოლო.

ასევე, განსხვავებით სხვა ტიპის ცვლადების წაკითხვისგან, მასივის წაკითხვისას არაა საჭირო `&` სიმბოლოს მითითება, ვინაიდან მასივის სახელი C-ში თავად წარმოადგენს მასივის პირველი ელემენტის მისამართს. `scanf` წაიკითხავს და `string2`-ში შეიტანს სიმბოლოებს მანამ, სანამ არ შეხვდება ხარე (Space), ტაბულაცია, ახალი ხაზის ნიშანი ან ფაილის დასასრულის სიმბოლო (EOF). `scanf` ავტომატურად არ ამოწმებს მასივის ზომას, ამიტომ პროგრამისტმა თავად უნდა უზრუნველყოს, რომ წაკითხვისას არ გამოვიდეთ მასივის ფარგლებს გარეთ.

სტრიქონი შეიძლება გამოტანილ იქნას ეკრანზე:

```
printf("%s\n", string2);
```

ქვემოთ მოყვანილ პროგრამაში გამოცხადებულია ორი სტრიქონი (`char` მასივი). მეორე ცვლადის ინიციალიზაცია ხდება სტრიქონული ლიტერალით. პირველ მასივში იკითხება მაქსიმუმ 19 სიმბოლო. შემდეგ იბეჭდება `string1` მასივი სიმბოლოებს შორის ცარიელი ადგილებით (ჰარებით): `"%c "`. დაბეჭდვის პირობა არის ორი: ან სანამ არ მიაღწევს მასივის ბოლოს, ან სანამ არ შეხვდება `'\0'` სიმბოლო.

```
1  // Fig. 6.10: fig06_10.c Treating character arrays as strings.
2  #include <stdio.h>
3
4  #define SIZE 20
```

```
5
6  // function main begins program execution
7  int main(int argc, char **argv)
8  {
9   char string1[SIZE];                 // reserves 20 characters
10  char string2[] = "string literal"; // reserves 15 characters
11  size_t i;                           // counter
12
13  // read string from user into array string1
14  printf("%s", "Enter a string (no longer than 19 characters): ");
15  scanf("%19s", string1); // input no more than 19 characters
16
17  // output strings
18  printf("string1 is: %s\nstring2 is: %s\n"
19  "string1 with spaces between characters is:\n",
20  string1, string2);
21
22  // output characters until null character is reached
23  for (i = 0; i < SIZE && string1[i] != '\0'; ++i) {
24   printf("%c ", string1[i]);
25  }
26  puts("");
27
28  return 0;
29 } // end main
```

მიმთითებლების გამოყენება მარტივია, ვინაიდან, როგორც ყველა მასივისათვის, მასივის სახელი მიუთითებს მასივის პირველ ელემენტზე (ნულოვან ინდექსზე). განვიხილოთ შემდეგი კოდი:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5   char text_1[100], text_2[100], text_3[100];
6   char *ta, *tb;
7   int i;
8   char text_x[] = "test";
9
10  // set message to be an array of characters
```

```
11  char message[] = "Hello, I am a string; what are you?";
12
13  printf("Original message: %s\n", message);
14
15  // copy the message to text_1 the hard way
16  i = 0;
17  while ((text_1[i] = message[i]) != '\0')
18  i++;
19  printf("Text_1: %s\n", text_1);
20
21  // use explicit pointer arithmetic
22  ta = message;
23  tb = text_2;
24  while ((*tb++ = *ta++) != '\0')
25  ;
26  printf("Text_2: %s\n", text_2);
27
28  ta = text_x;
29  tb = text_3;
30  while ((*tb++ = *ta++) != '\0')
31  ;
32  printf("Text_3: %s\n", text_3);
33
34  return 0;
35 }
```

6.3 სიმბოლოების მასივი '0'-ის გარეშე

არის შემთხვევები, როცა გვაქვს სიმბოლოების მასივი, რომელიც არ მთავრდება '\0'-ით. ასეთ შემთხვევაში შეგვიძლია გამოვიყენოთ ბრძანება printf("%.*s", length, ptr), რაც არის C/C++ ენაში C-სტილის სტრიქონების ბეჭდვის ძლიერი და უსაფრთხო მექანიზმი.

მოკლედ რომ ვთქვათ, ის ეუბნება printf-ს: „დაბეჭდე ზუსტად length რაოდენობის სიმბოლო ptr მისამართიდან და შეჩერდი (ნუ ეძებ ნულოვან სიმბოლოს \0)“.

6.3.1 სინტაქსის დეტალური განმარტება

თითოეული სიმბოლო %.*s ჩანაწერში ნიშნავს:

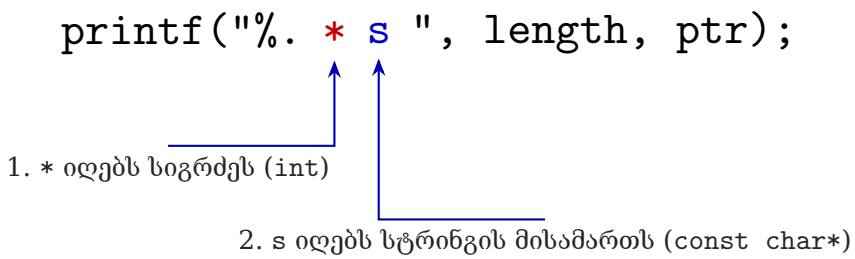
% — აცხადებს ფორმატირების დასაწყისს.

s — მიუთითებს, რომ გადაეცემა სტრინგის მაჩვენებელი (const char*).

. **(წერტილი)** — სტრინგების შემთხვევაში განსაზღვრავს **მაქსიმალურ სიგრძეს** (precision), თუ რამდენი სიმბოლო უნდა დაიბეჭდოს.

* **(ფიფქი)** — ეუბნება printf-ს, რომ ეს სიგრძე წინასწარ კი არ არის გაწერილი, არა-

მედ **დინამიურად გადაეცემა არგუმენტად**.
ამიტომ, ამ ფორმატს სჭირდება **ორი არგუმენტი**:




```c
// %.*s ეუბნება printf-ს: წაიკითხე ზუსტად n სიმბოლო sec-დან და გაჩერდი!
printf("%.*s\n", n, sec);
```

ცხრილი 6.1 printf და ტექსტი

ფორმატი	კოდის მაგალითი	რას აკეთებს printf?	შედეგი
%s	printf("%s", "hello")	ბეჭდავს '\0'-მდე.	hello
%.3s	printf("%.3s", "hello")	ბეჭდავს 3 („ჩაშენებულია“) სიმბოლოს.	hel
%.*s	printf("%.*s", 3, "hello")	ბეჭდავს 3 (დინამიურად გადაეცემა) სიმბოლოს.	hel

განვიხილოთ C-style მაგალითი C-style მასივებისთვის, როცა გვაქვს char მასივი, რომელსაც ბოლოში '\0' არ აქვს:

კოდი 6.1 სიმბოლოების მასივი '\0'-ის გარეშე

```c
#include <stdio.h>

int main() {
 // მასივი NULL-ის ('\0') გარეშე!
 char buffer[5] = {'H', 'e', 'l', 'l', 'o'};
 // %s-ის გამოყენება გამოიწვევდა Undefined Behavior-ს
 // %.*s უსაფრთხოდ ბეჭდავს მხოლოდ 5 სიმბოლოს:
 printf("%.*s\n", 5, buffer);
 return 0;
}
```

6.4 სტრიქონის წაკითხვა ფაილიდან

განვიხილოთ მაგალითი:

კოდი 6.2 სტრიქონ-სტრიქონ წაკითხვა ფაილიდან

```c
#include <stdio.h>
#include <string.h>

#define MAX_LEN 256

int main(void)
{
 FILE* fp;
 fp = fopen("file.txt", "r");
 if (fp == NULL) {
  perror("Failed: ");
  return 1;
 }
 char buffer[MAX_LEN];
 while (fgets(buffer, MAX_LEN, fp))
 {
  // Remove trailing newline
  buffer[strcspn(buffer, "\r\n")] = 0;
  printf("%s\n", buffer);
 }
 fclose(fp);
 return 0;
}
```

6.5 მასივის გადაცემა ფუნქციაში

როგორც ითქვა, მასივის სახელი იგივეა, რაც მასივის მისამართი და იგივეა, რაც მასივის პირველი ელემენტის (ნულოვანი ინდექსით) მისამართი. ვნახოთ ეს მაგალითზე:

```c
// Fig. 6.12: fig06_12.c
// Array name is the same as the address of the array's first element.
#include <stdio.h>

// function main begins program execution
int main(int argc, char **argv)
{
 char array[5]; // define an array of size 5
```

```
 printf("    array = %p\n&array[0] = %p\n"
 "   &array = %p\n", array, &array[0], &array);
 return 0;
} // end main
```

ფუნქციას შეიძლება მიეწოდოს მასივი როგორც არგუმენტი, ან მასივის ელემენტი. მასივის ელემენტის მიწოდების მაგალითი:

```
#include <stdio.h>

void display(int a)
{
 printf("%d", a);
}

int main(){
 int c[] = {2, 3, 4};
 display(c[2]); // gadaecema mxolod elementi c[2].
 return 0;
}
```

ანუ, მასივის ელემენტი მიეწოდება ისევე, როგორც სხვა ჩვეულებრივი ცვლადი.

ფუნქციას შეიძლება გადაეცეს მასივიც, უფრო სწორედ, მასივის მისამართი. რადგან მასივის სახელი ინახავს მასივის მისამართს (ანუ ამავე დროს არის მიმთითებელიც, სადაც შენახულია მასივის მისამართი), ფუნქციას მიეწოდება მასივის სახელი.

6.5.1 მაგალითი: ერთგანზომილებიანი მასივის გადაცემა

ფუნქციას გადაეცემა ერთგანზომილებიანი მასივი და ითვლის ამ მასივის ელემენტების საშუალო მნიშვნელობას:

```
#include <stdio.h>

float average(float a[]);

int main(int argc, char **argv)
{
 float avg, c[] = {23.4, 55, 22.6, 3, 40.5, 18};
 avg = average(c); // Only name of array is passed as argument.
 printf("Average age=%.2f", avg);
 return 0;
}
```

```
12
13 float average(float a[]){
14  int i;
15  float avg, sum = 0.0;
16  for(i = 0; i < 6; ++i){
17   sum += a[i];
18  }
19  avg = (sum / 6);
20  return avg;
21 }
```

როგორც ვხედავთ, მასივის გადაცემისას არაა აუცილებელი გადაეცეს მასივის ზომაც. ოღონდ, ცხადია, უნდა ვიცოდეთ, რა ზომის მასივს გადავცემთ. ზემოთ მოყვანილ მაგალითში ესაა 6. თუ მასივის ზომა კომპილაციის მომენტში არ ვიცით და საჭიროა, რომ დათვლილ იქნას სხვადასხვა ზომის მასივების საშუალო, მაშინ ფუნქციას უნდა გადავცეთ მასივის ზომაც: `float average(float a[], int N);` და ციკლი გაივლის არა 6-ს, არამედ N ელემენტს.

მაგალითად, ზემოთ როცა ვარჩევდით კვადრატული განტოლების ამოხსნას, ვიცით, რომ ფუნქციას მიეწოდება სამი ცვლადი, როგორც კვადრატული განტოლების კოეფიციენტები და ორი ცვლადი, როგორც კვადრატული განტოლების ამონახსნები. რაც ნიშნავს, რომ შეგვიძლია ფუნქციას მივაწოდოთ ერთ, ორ და სამგანზომილებიანი მასივები და აუცილებელი არ იქნება მათი ზომების მიწოდება. მაგრამ თუ ფუნქციამ არ იცის მასივის ზომა, მაშინ იმისათვის, რომ ოპერაციები შეასრულოს მასივის ელემენტებზე, საჭიროა მასივის ზომის მიწოდებაც.

```
1  // Fig. 6.13: fig06_13.c
2  // Passing arrays and individual array elements to functions.
3  #include <stdio.h>
4
5  #define SIZE 5
6
7  // function prototypes
8  void modifyArray( int b[], size_t size );
9  void modifyElement( int e );
10
11 int main(int argc, char **argv)
12 {
13  int a[SIZE] = { 0, 1, 2, 3, 4 }; // initialize array a
14  size_t i; // counter
15  puts( "Effects of passing entire array by reference:\n\nThe "
16  "values of the original array are:" );
```

```
17  // output original array
18  for ( i = 0; i < SIZE; ++i ) {
19   printf( "%3d", a[i] );
20  } // end for
21  puts( "" );
22
23  // pass array a to modifyArray by reference
24  modifyArray( a, SIZE );
25
26  puts( "The values of the modified array are:" );
27  // output modified array
28  for ( i = 0; i < SIZE; ++i ) {
29   printf( "%3d", a[i] );
30  } // end for
31
32  // output value of a[3]
33  printf( "\n\n\nEffects of passing array element "
34  "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
35  modifyElement( a[3] ); // pass array element a[ 3 ] by value
36
37  // output value of a[3]
38  printf( "The value of a[3] is %d\n", a[3] );
39 } // end main
40
41 // in function modifyArray, "b" points to the original array "a"
42 // in memory
43 void modifyArray( int b[], size_t size )
44 {
45  size_t j; // counter
46
47  // multiply each array element by 2
48  for ( j = 0; j < size; ++j ) {
49   b[j] *= 2; // actually modifies original array
50  } // end for
51 } // end function modifyArray
52
53 // in function modifyElement, "e" is a local copy of array element
54 // a[3] passed from main
55 void modifyElement( int e )
56 {
```

```
57  // multiply parameter by 2
58  printf( "Value in modifyElement is %d\n", e *= 2 );
59 } // end function modifyElement
```

მასივის გადაცემისას `void arraytest(int a[])` და `void arraytest(int *a)` არის ერთი და იგივე. C-ში მასივი გადაეცემა არა მნიშვნელობით, არამედ მისამართით, ანუ მასივის მნიშვნელობები ლოკალურად არ კოპირდება. მასივების შემთხვევაშიც შეიძლება გამოყენებულ იქნას `const` კვალიფიკატორი, რაც ნიშნავს იმას, რომ გვაქვს მუდმიველემენტებიანი მასივი და მისი მნიშვნელობების შეცვლა აკრძალულია პროგრამის მიერ.

დავალება 6.6: კვადრატული განტოლება მასივით

შეცვალეთ ზემოთ გარჩეული კვადრატული განტოლების ამოხსნის პროექტის უკანასკნელი ვერსია ისე, რომ კოეფიციენტების და ფესვების ცვლადების ნაცვლად გამოყენებულ იქნას მასივები. გაითვალისწინეთ, რომ მასივის სახელი იგივეა, რაც მიმთითებელი.

6.5.2 მაგალითი: მასივის დალაგება

ქვემოთ მოყვანილია მასივის დალაგების ე.წ. „Bubble Sort“ ალგორითმი:

```
1  // Fig. 6.15: fig06_15.c
2  // Sorting an array's values into ascending order.
3  #include <stdio.h>
4  #define SIZE 10
5  // function main begins program execution
6  int main(int argc, char **argv)
7  {
8   // initialize a
9   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
10  int pass; // passes counter
11  size_t i; // comparisons counter
12  int hold; // temporary location used to swap array elements
13  puts( "Data items in original order" );
14  // output original array
15  for ( i = 0; i < SIZE; ++i ) {
16   printf( "%4d", a[ i ] );
17  } // end for
18  // bubble sort
19  // loop to control number of passes
20  for ( pass = 1; pass < SIZE; ++pass ) {
21   // loop to control number of comparisons per pass
22   for ( i = 0; i < SIZE - 1; ++i ) {
```

```
23     // compare adjacent elements and swap them if first
24     // element is greater than second element
25     if ( a[ i ] > a[ i + 1 ] ) {
26      hold = a[ i ];
27      a[ i ] = a[ i + 1 ];
28      a[ i + 1 ] = hold;
29     } // end if
30    } // end inner for
31   } // end outer for
32   puts( "\nData items in ascending order" );
33   // output sorted array
34   for ( i = 0; i < SIZE; ++i ) {
35    printf( "%4d", a[ i ] );
36   } // end for
37   puts( "" );
38  } // end main
```

დავალება 6.7: მასივის წაკითხვა/ბეჭდვა

1. შეცვალეთ მასივის დალაგების პროგრამა მასივის დალაგების ფუნქციად, რომელსაც გადაეცემა მასივი და მასივის ზომა.
2. შემოიტანეთ ფუნქცია, რომელიც ბეჭდავს მასივს.
3. დაწერეთ ფუნქცია, რომელიც წაიკითხავს მასივს.
4. დაწერეთ პროგრამა და გამოიყენეთ ზემოთ დაწერილი ფუნქციები: წააკითხეთ 10-ელემენტიანი მასივი, გამოიტანეთ ეკრანზე, დაალაგეთ ზრდადობით, შედეგი გამოიტანეთ ეკრანზე.

დავალება 6.8: მასივის დალაგება

იგივე, რაც ზემოთ, ოღონდ შემოიტანეთ დალაგების ფუნქცია, რომელიც მასივს დაალაგებს კლებადობით.

6.6 მრავალგანზომილებიანი მასივები

C-ში შესაძლებელია განმარტებულ იქნას მრავალგანზომილებიანი მასივები, ანუ მასივის მასივი. მაგალითად: `float a[2][6];` — a არის ორგანზომილებიანი მასივი. ეს მასივი გრაფიკულად შეგვიძლია წარმოვიდგინოთ ასეთი სახით:

	სვ. 1	სვ. 2	სვ. 3	სვ. 4	სვ. 5	სვ. 6
სტრ. 1	$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$	$a[0][4]$	$a[0][5]$
სტრ. 2	$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$	$a[1][4]$	$a[1][5]$

მანქანის მეხსიერებაში მასივი სწორედ ასეა განლაგებული, ანუ სტრიქონების მიხედვით

(იხ. სურ. 6.3), განსხვავებით Fortran-ის მასივებისგან. Fortran-ის მასივები მეხსიერებაში განლაგდებიან სვეტების მიხედვით. მიაქციეთ ყურადღება, რომ შიდა ინდექსი გარბის სვეტებს და გარე — სტრიქონებს.

6.6.1 მრავალგანზომილებიანი მასივის ინიციალიზაცია

C ენაში მრავალგანზომილებიანი მასივების ინიციალიზაცია შეიძლება სხვადასხვა გზით:

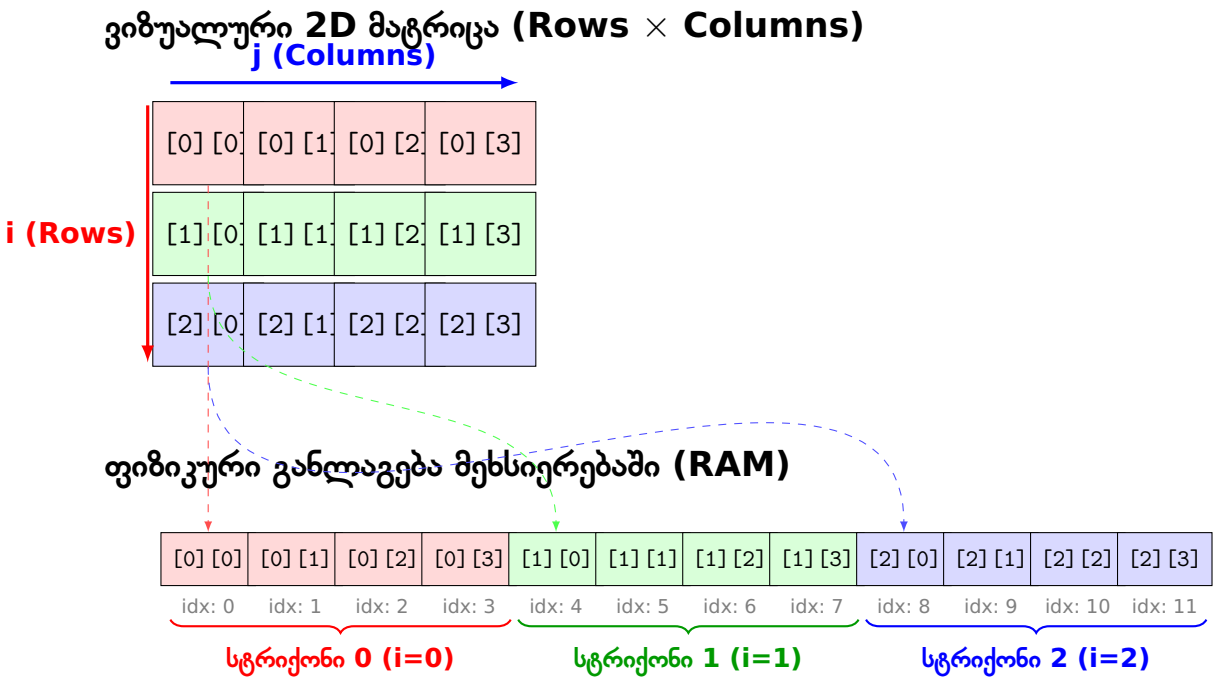
`int c[2][3]={{1,3,0}, {-1,5,9}};` — ცხადად მიეთითება ორივე განზომილება $(2, 3)$ და თითოეული ერთგანზომილებიანი მასივი (3-ელემენტიანი) ინიციალიზდება ცალ-ცალკე.

`int c[][3]={{1,3,0}, {-1,5,9}};` — შეიძლება მიეთითოს მხოლოდ ერთ განზომილებიანი მასივის განზომილება (ეს აუცილებელია, ანუ მასივების რაოდენობა შეიძლება არ მიეთითოს).

`int c[2][3]={1,3,0,-1,5,9};` — შეიძლება ინიციალიზდეს, როგორც ერთგანზომილებიანი მასივი.

`int myPoints[][] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9} };` — შეცდომაა.

`int myPoints[9][] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9}};` — შეცდომაა.



სურ. 6.3 2D მასივი C ენაში და მეხსიერება

მეხსიერების ინდექსის (idx) გამოთვლის ფორმულა:

ზოგადი ფორმულა: **idx = i * COLS + j** (სადაც `COLS = 4`)

მაგალითად `[1][2]` ელემენტისთვის:

`idx = 1 * 4 + 2 =` **6**

(იხილეთ მეხსიერებაში `idx: 6` ბლოკი)

სამგანზომილებიანი მასივის ინიციალიზაციის მაგალითი:

```
double cprogram[3][2][4] = {
 {{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

ეს სამგანზომილებიანი მასივი შეიძლება გავიგოთ ასე: გვაქვს ორსვეტიანი და სამსტრიქონიანი მასივი, რომლის ელემენტებია ოთხელემენტიანი მასივი.

თუ გვაქვს, მაგალითად, `arr[i][j][k][m]` მასივი, მაშინ ამ მასივის ელემენტების რაოდენობაა `i * j * k * m`.

6.6.2 მაგალითი: ორგანზომილებიანი მასივების შეკრება

ამოცანა: ქვემოთ მოყვანილ კოდში მომხმარებელს შეჰყავს კონსოლიდან ორი ორგანზომილებიანი მასივი, რომელთა ელემენტები იკრიბება და მიენიჭება მესამე ორგანზომილებიანი მასივის ელემენტებს.

```
#include <stdio.h>

int main(){
 float a[2][2], b[2][2], c[2][2];
 int i, j;

 printf("shemoitanet pirveli matricis elementebi\n");
 // vkiTxulobT elementebs.
 for(i = 0; i < 2; ++i)
 for(j = 0; j < 2; ++j){
  printf("shemoitane a%d%d: ", i + 1, j + 1);
  scanf("%f", &a[i][j]);
 }

 printf("shemoitanet meore matricis elementebi\n");
 for(i = 0; i < 2; ++i)
 for(j = 0; j < 2; ++j){
  printf("shemoitane b%d%d: ", i + 1, j + 1);
  scanf("%f", &b[i][j]);
 }

 for(i = 0; i < 2; ++i)
 for(j = 0; j < 2; ++j){
  // vkribavt ori matricis elementebs da vwert mesameshi.
```

```
25    c[i][j] = a[i][j] + b[i][j];
26   }
27
28   printf("\nmatricebis jamia:\n");
29   for(i = 0; i < 2; ++i)
30   for(j = 0; j < 2; ++j){
31    printf("%.1f\t", c[i][j]);
32    if(j == 1) // gadavdivarT axal xazze.
33    printf("\n");
34   }
35
36   return 0;
37  }
```

6.6.3 2D მასივის გადაცემა ფუნქციაში

ორგანზომილებიანი მასივის გადაცემისას მიეწოდება ამ მასივის დასაწყისის მისამართი, მსგავსად ერთგანზომილებიანი მასივის შემთხვევისა:

```
1   #include <stdio.h>
2
3   void Function(int c[2][2]);
4
5   int main(){
6    int c[2][2], i, j;
7    printf("shemoitane 4 ricxvi:\n");
8    for(i = 0; i < 2; ++i)
9    for(j = 0; j < 2; ++j){
10    scanf("%d", &c[i][j]);
11   }
12   Function(c); // 2D მასივის გადაცემა ფუნქციაში
13   return 0;
14  }
15
16  void Function(int c[2][2]){
17   // შეიძლება ასევე გადაცემულ იქნას, როგორც
18   // void Function(int c[][2]){.....}
19   int i, j;
20   printf("Displaying:\n");
21   for(i = 0; i < 2; ++i)
22   for(j = 0; j < 2; ++j)
```

```
 printf("%d\n", c[i][j]);
}
```

ასევე შესაძლებელია და სწორია შემდეგი გადაცემაც: `void Function(int c[][2]) {.....}`, ანუ არაა აუცილებელი გარე განზომილების (პირველი განზომილების, ანუ სტრიქონთა რაოდენობის) აღწერა/გადაცემა. მსგავსი შემთხვევა გვქონდა ერთგანზომილებიან შემთხვევაშიც. მაგრამ აუცილებელია დანარჩენი განზომილებების აღწერა/გადაცემა.

ზემოთ მოყვანილ შემთხვევაში ეს ნიშნავს, რომ ფუნქციას გადაეცემა ორელემენტიანი მასივები, მაგრამ რამდენი ორელემენტიანი მასივი გადაეცემა, ამის მითითება არაა აუცილებელი. ცხადია, ამ მასივებთან სამუშაოდ უნდა ვიცოდეთ, რამდენი მასივი გადაეცემა. ეს კი ან უნდა იყოს თავიდანვე ჩადებული პროგრამაში კომპილაციის დროს, ან ფუნქციას უნდა გადავცეთ მასივების რაოდენობაც, ისე, როგორც ეს გვქონდა ერთგანზომილებიან შემთხვევაში.

მეორე და დანარჩენი განზომილებების გადაცემა კი აუცილებელია.

1) ვთქვათ, მეორე განზომილება ხელმისაწვდომია გლობალურად (ანუ მოცემულია ან მაკროსის სახით, ან ცვლადის სახით, ან ცხადი სახით მითითებულია კოდში კომპილაციის მომენტში):

```
#include <stdio.h>

const int n = 3;

void print(int arr[][n], int m)
{
 int i, j;
 for (i = 0; i < m; i++)
 for (j = 0; j < n; j++)
 printf("%d ", arr[i][j]);
}

int main()
{
 int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
 print(arr, 3);
 return 0;
}
```

ზემოთ მოყვანილი მეთოდი მუშაობს, თუ მეორე განზომილება ფიქსირებულია (ანუ ცნობილია კოდის კომპილაციისას) და არაა მომხმარებელზე დამოკიდებული.

2) თუ კოდი არის C99 თავსებადი C99 ვერსიიდან C ენაში შესაძლებელია ცვლადი ზომის მასივების გადაცემაც ზომების გადაცემის საშუალებით (VLA — Variable Length Array). ქვემოთ მოყვანილი მაგალითი მუშაობს მხოლოდ C99 თავსებადი კომპილატორისათვის:

```c
#include <stdio.h>

// n must be passed before the 2D array
void print(int m, int n, int arr[][n])
{
 int i, j;
 for (i = 0; i < m; i++)
 for (j = 0; j < n; j++)
 printf("%d ", arr[i][j]);
}

int main()
{
 int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
 int m = 3, n = 3;
 print(m, n, arr);
 return 0;
}
```

6.6.4 C99-მდე არსებული ალტერნატიული მეთოდები 2D მასივის გადასაცემად

თუ კომპილატორი არ არის C99 თავსებადი, გამოიყენება ქვემოთ მოყვანილი მეთოდებიდან ერთ-ერთი.

ა) მიმთითებელზე ტიპის დაყვანა (Type Casting)

ამ მეთოდის გამოყენებისას მრავალგანზომილებიან მასივს ვიხილავთ როგორც ერთგანზომილებიანს, სადაც „ჩალაგებულია“ მრავალგანზომილებიანი მასივის ყველა ელემენტი. საჭიროა ე.წ. „ტიპის კასტირება“ (Type Casting). მაგალითად, ქვემოთ მოყვანილ კოდში `print` ფუნქციას ცხადი სახით ვეუბნებით, რომ `arr` გადაეცემა როგორც `int*` ტიპის მიმთითებელი:

```c
#include <stdio.h>

void print(int *arr, int m, int n)
{
 int i, j;
 for (i = 0; i < m; i++) {
```

```
  for (j = 0; j < n; j++) {
   printf("%d ", *((arr + i * n) + j));
  }
  printf("\n");
 }
}

int main(void)
{
 int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
 int m = 3, n = 3;
 print((int *)arr, m, n);
 return 0;
}
```

ბ) მიმთითებლების მასივის (Array of Pointers) გამოყენება

შემდეგი კოდი **არ იმუშავებს** და გამოიწვევს შეცდომას ან არასწორ ქცევას:

```
// არასწორი მიდგომა: int static 2D array არ გადადის int**-ში
#include <stdio.h>

void print(int **arr, int m, int n)
{
 int i, j;
 for (i = 0; i < m; i++)
 for (j = 0; j < n; j++)
 printf("%d ", *((arr + i * n) + j));
}

int main(void)
{
 int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
 int m = 3, n = 3;
 print((int **)arr, m, n); // შეცდომა!
 return 0;
}
```

მთავარი პრობლემა იმაშია, რომ `int arr[3][4]` (ორგანზომილებიანი სტატიკური მასივი) არ არის იგივე, რაც `int**` (ორმაგი მიმთითებელი / pointer to pointer):

- `int arr[3][4]` არის მეხსიერების ერთიანი, უწყვეტი 12-ელემენტიანი ბლოკი.

- int** კი ელოდება მეხსიერების მისამართს, სადაც წერია სხვა მისამართების (მიმთითებლების) მასივი, რომლებიც თავის მხრივ მიუთითებენ სტრიქონებზე.

იმისათვის, რომ კოდმა იმუშაოს ორმაგი მიმთითებლის (int**) გამოყენებით დინამიური ალოკაციის (malloc) გარეშე, შეგვიძლია შევქმნათ მიმთითებლების დამხმარე მასივი (array of pointers) სტეკზე და გადავცეთ ის ფუნქციას:

```c
#include <stdio.h>

// ფუნქცია იღებს ორმაგ მიმთითებელს (int**)
void print_test(int **arr, int m, int n)
{
 int i, j;
 for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
   // arr[i][j] იგივეა, რაც *(*(arr + i) + j)
   printf("%d ", arr[i][j]);
  }
  printf("\n");
 }
}

int main(void)
{
 // 1. სტატიკური 2D მასივი
 int arr[3][4] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}
 };

 int m = 3, n = 4;

 // 2. მიმთითებლების სტატიკური მასივი (Array of pointers)
 // თითოეულ ელემენტს ვუსვამთ arr-ის შესაბამისი სტრიქონის საწყის მისამართს
 int *row_pointers[3] = { arr[0], arr[1], arr[2] };

 // 3. ფუნქციას გადავცემთ ამ მიმთითებლების მასივს.
 // row_pointers ავტომატურად გარდაიქმნება int** ტიპად!
 print_test(row_pointers, m, n);

```

```
35  return 0;
36 }
```

row_pointers არის 3-ელემენტიანი მასივი, სადაც თითოეული ელემენტი არის int* ტიპის მიმთითებელი. row_pointers[0] ინახავს arr მასივის პირველი სტრიქონის მისამართს (&arr[0][0]), row_pointers[1] — მეორე სტრიქონის მისამართს (&arr[1][0]) და ა.შ.

ფუნქციაში arr[i][j] ოპერაცია ჯერ მიმართავს row_pointers[i]-ს (რომელიც სწორ მისამართზე მიუთითებს) და შემდეგ იქიდან იღებს j-ურ ელემენტს. კოდი მუშაობს სწრაფად და უსაფრთხოდ, ყოველგვარი malloc-ისა და free-ს გარეშე.

6.7 1D მასივის ჩაწერა/წაკითხვა ფაილიდან

ქვემოთ მოყვანილი კოდი ფაილში ამატებს ახალ სიტყვებს ("a+" რეჟიმი).

კოდი 6.3 ფაილში ჩაწერა და წაკითხვა

```c
/* count.c -- using standard I/O */
#include <stdio.h>
#include <stdlib.h> // exit() prototype
#include <string.h>

#define MAX 41

int main(void)
{
 FILE *fp;
 char words[MAX];

 if ((fp = fopen("wordy", "a+")) == NULL)
 {
  fprintf(stderr, "Can't open \"wordy\" file.\n");
  exit(EXIT_FAILURE);
 }

 puts("Enter words to add to the file; press the #");
 puts("key at the beginning of a line to terminate.");

 while ((fscanf(stdin, "%40s", words) == 1) && (words[0] != '#')) {
  fprintf(fp, "%s\n", words);
 }

 puts("\nFile contents:");
```

```
27  rewind(fp); /* go back to beginning of file */
28
29  while (fscanf(fp, "%40s", words) == 1) {
30   puts(words);
31  }
32
33  puts("Done!");
34  if (fclose(fp) != 0) {
35   fprintf(stderr, "Error closing file\n");
36  }
37
38  return 0;
39 }
```

კოდის დეტალური განმარტება:

მე-6 ხაზი: მაკროსი `MAX` განსაზღვრავს მასივის ზომას (41). ეს ნიშნავს, რომ სიტყვის მაქსიმალური სიგრძე შეიძლება იყოს 40 სიმბოლო, ხოლო ბოლო 41-ე პოზიციას იკავებს ნულ-სიმბოლო `'\0'`.

მე-14 ხაზი: `"a+"` რეჟიმი ფაილს ხსნის წაკითხვისა და დამატების მიზნით (Append/Update). თუ ფაილი არ არსებობს, იქმნება ახალი. ჩაწერილი მონაცემები ყოველთვის ფაილის ბოლოში დაემატება.

მე-22 ხაზი: `fscanf(stdin, "%40s", words)` იღებს სიტყვას სტანდარტული შემოსავლიდან (ეკრანიდან). ფორმატი `%40s` ზღუდავს წაკითხვას 40 სიმბოლომდე, რაც იცავს ბუფერს გადავსებისგან (Buffer Overflow). პირობა `words[0] != '#'` უზრუნველყოფს, რომ `#` სიმბოლოთი დაწყებული სიტყვა ჩაითვალოს შეწყვეტის სიგნალად.

მე-23 ხაზი: `fprintf(fp, "%s\n", words)` ჩაწერს წაკითხულ სიტყვას ფაილში და გადადის ახალ სტრიქონზე.

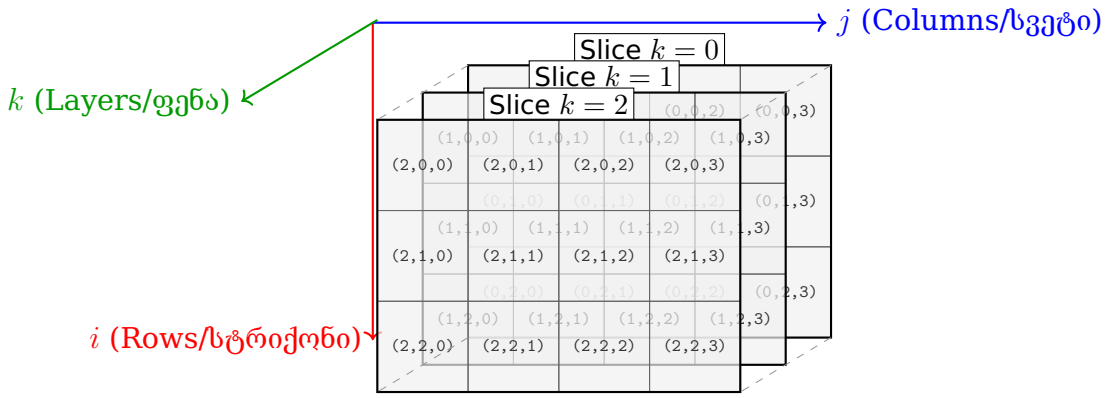
27-ე ხაზი: `rewind(fp)` ფუნქცია ფაილის მიმთითებელს (File Position Indicator) აბრუნებს ფაილის დასაწყისში, რათა შესაძლებელი გახდეს ახლახან ჩაწერილი მონაცემების ხელახლა წაკითხვა.

29–31 ხაზები: `fscanf(fp, "%40s", words)` კითხულობს ფაილიდან სიტყვა-სიტყვამდე, ვიდრე არ მიაღწევს ფაილის ბოლოს (EOF).

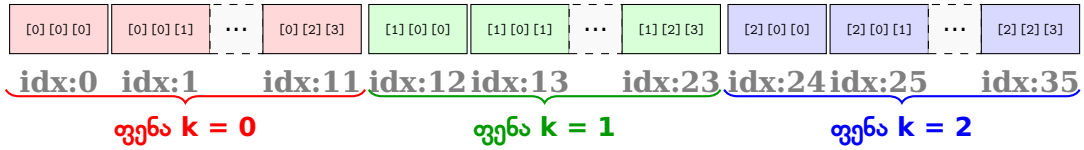
34–35 ხაზები: `fclose(fp)` ხურავს ფაილის ნაკადს.

6.8 სტატიკური (სტეკ) 3D მასივი

განვიხილოთ 3D სტატიკური მასივი, რომელიც განთავსებულია სტეკ მეხსიერებაში.



სურ. 6.4 3D მასივის `arr[3][3][4]` ლოგიკური სტრუქტურა



სურ. 6.5 3D მასივის `arr[3][3][4]` ფიზიკური განლაგება მეხსიერებაში

როგორ მუშაობს 3D მეხსიერების სტრუქტურა:

- მეხსიერება არის სრულიად **უწყვეტი (Contiguous)**. ჯერ ინახება ფენა 0, შემდეგ ფენა 1 და ბოლოს ფენა 2.
- თითოეული ფენის შიგნით მონაცემები ლაგდება **სტრიქონების მიხედვით (Row-Major Order)**: `[k][0][0]` → `[k][0][1]` → ... → `[k][2][3]`

მეხსიერების 1D ინდექსის (idx) გამოთვლის ფორმულა:

ზოგადი ფორმულა:

$$\text{idx} = k \cdot (\text{ROWS} \cdot \text{COLS}) + i \cdot \text{COLS} + j$$

ჩვენს შემთხვევაში ($\text{ROWS} = 3, \text{COLS} = 4$):

$$\text{idx} = k \cdot 12 + i \cdot 4 + j$$

მაგალითად `[1][2][3]` ელემენტისთვის:

$$\text{idx} = 1 \cdot 12 + 2 \cdot 4 + 3 = 12 + 8 + 3 = 23$$

3D მასივის ინდექსირების ვარიანტები

3D მასივის ერთგანზომილებიან მეხსიერებაში ასახვისას (Flat Indexing) ინდექსის გამოთვლის ფორმულა დამოკიდებულია იერარქიაზე — თუ რომელი განზომილება წარმოადგენს გარე ფენას და რომელი იცვლება ყველაზე სწრაფად.

1. **ფენები Z (k) ღერძზე, ყველაზე სწრაფად ცვლადი Y (j)**
   - **იერარქია:** k (გარე) → i (შუა) → j (შიდა)
   - **ზომები:** N_z (ფენები / *LAYERS*), N_x (სტრიქონები / *ROWS*), N_y (სვეტები / *COLS*)

- **ინდექსის გამოთვლის ფორმულა:**

$$\text{idx} = k \cdot (N_x \cdot N_y) + i \cdot N_y + j$$

- **განმარტება:** მეხსიერებაში გვერდიგვერდ ინახება Y (j) ინდექსის ელემენტები. ჯერ ივსება j, შემდეგ იზრდება i, ხოლო XY სიბრტყის სრულად შევსების შემდეგ გადავდივართ Z-ის (k) შემდეგ ფენაზე.

2. **ფენები X (i) ღერძზე, ყველაზე სწრაფად ცვლადი Z (k)**
   - **იერარქია:** i (გარე) $\rightarrow$ j (შუა) $\rightarrow$ k (შიდა)
   - **ზომები:** N_x (ფენები), N_y (სტრიქონები), N_z (სვეტები/სიღრმე)
   - **ინდექსის გამოთვლის ფორმულა:**

$$\text{idx} = i \cdot (N_y \cdot N_z) + j \cdot N_z + k$$

   - **განმარტება:** მასივი ფენებად დაყოფილია X (i) ღერძის გასწვრივ. მეხსიერებაში ყველაზე სწრაფად იცვლება Z (k) ინდექსი. k-ს შევსების შემდეგ იზრდება Y (j), ხოლო მთლიანი YZ სიბრტყის შევსების შემდეგ გადავდივართ X-ის (i) მომდევნო ფენაზე.
3. **ფენები Z (k) ღერძზე, ყველაზე სწრაფად ცვლადი X (i)**
   - **იერარქია:** k (გარე) $\rightarrow$ j (შუა) $\rightarrow$ i (შიდა)
   - **ზომები:** N_z (ფენები), N_y (სტრიქონები), N_x (სვეტები)
   - **ინდექსის გამოთვლის ფორმულა:**

$$\text{idx} = k \cdot (N_y \cdot N_x) + j \cdot N_x + i$$

   - **განმარტება:** ფენები კვლავ Z (k) ღერძზეა, თუმცა თითოეულ ფენაში მეხსიერებაში მიმდევრობით ლაგდება X (i) მიმართულების ელემენტები. i-ს ამოწურვის შემდეგ იზრდება Y (j), ხოლო XY სიბრტყის შევსების შემდეგ გადავდივართ Z-ის (k) მომდევნო ფენაზე.

ზოგადი წესი: გარე ინდექსი მრავლდება მის შიგნით არსებული ყველა განზომილების ზომების ნამრავლზე, შუა ინდექსი მრავლდება ბოლო (ყველაზე შიდა) განზომილების ზომაზე, ხოლო ყველაზე შიდა (სწრაფი) ინდექსი პირდაპირ ემატება.

6.9 ცვლადი ზომის მასივები (VLA) C99/C11 სტანდარტებში

6.9.1 VLA-ის ისტორია და C11 ცვლილებები

C99 სტანდარტი: VLA (Variable Length Array) პირველად დაინერგა C99 სტანდარტში, სადაც ის **სავალდებულო** (Mandatory) ფუნქციონალი იყო. VLA საშუალებას იძლევა მასივის ზომა განისაზღვროს გაშვებისას (Runtime Variable-ით) სტეკ მეხსიერებაში (მაგ.: `int arr[n];`).

C11 სტანდარტი: C11-ში VLA გახდა **არასავალდებულო** (Optional) ფუნქცია. კომპილატორებს უფლება მიეცათ არ დაუჭიროთ მხარი VLA-ს.

`__STDC_NO_VLA__` **მაკროსი:** კომპილატორი აცხადებს მაკროს `__STDC_NO_VLA__ = 1`, თუ მას **არ აქვს** VLA-ს მხარდაჭერა.

6.9.2 რატომ გახდა VLA არასავალდებულო?

1. **სტეკის გადავსების რისკი (Stack Overflow):** VLA მეხსიერებას გამოყოფს სტეკზე. თუ ცვლადი n ძალიან დიდია (მაგ. $n = 1000000$), სტეკი გადაივსება და პროგრამა ავარიულად დაასრულებს მუშაობას (Segmentation Fault).
2. **შეცდომების მართვის არარსებობა:** VLA-ის გამოყოფისას კომპილატორი ვერ ამოწმებს წარმატებით გამოიყო თუ არა მეხსიერება (განსხვავებით `malloc`-ისგან, რომელიც აბრუნებს `NULL`-ს).
3. **უსაფრთხოების ხვრელები (Security Vulnerabilities):** არასანდო მომხმარებლის შეყვანილმა ზომამ შეიძლება გამოიწვიოს Stack Smashing შეტევები.

6.9.3 VLA მხარდაჭერის შემოწმება კოდში

```c
#include <stdio.h>

int main(void)
{
 #ifdef __STDC_NO_VLA__
 printf("VLA is NOT supported by this compiler (C11 standard optional
  ↪ feature).\n");
 #else
 printf("VLA is supported!\n");

 int n = 5;
 int vla_array[n]; // VLA გამოცხადება

 for (int i = 0; i < n; i++) {
  vla_array[i] = (i + 1) * 10;
  printf("vla_array[%d] = %d\n", i, vla_array[i]);
 }
 #endif
 return 0;
}
```

VLA-ის უსაფრთხო ალტერნატივაა დინამიური მასივი ჰიპ (Heap) მეხსიერების გამოყენება (`malloc`) რასაც მოგვიანებით განვიხილავთ.

6.10 ბრძანებათა სტრიქონის არგუმენტები: argc და argv

C და C++ ენებში `main` ფუნქციას შეუძლია მიიღოს პარამეტრები ოპერაციული სისტემიდან (Terminal / CMD). ამისათვის გამოიყენება შემდეგი სინტაქსი:

```
1  int main(int argc, char *argv[])
```

ძირითადი ცვლადების განმარტება:

argc **(Argument Count):** `int` ტიპის მთელი რიცხვია, რომელიც ინახავს გადაცემული არგუმენტების საერთო რაოდენობას. ყოველთვის $argc \geq 1$, რადგან პირველ არგუმენტად ავტომატურად ითვლება თვითონ პროგრამის გაშვების ბრძანება/გზა.

argv **(Argument Vector):** სტრიქონების (`char*`) მასივია:

- `argv[0]` — შეიცავს გაშვებული პროგრამის სახელს ან სრულ მისამართს.
- `argv[1]` ... `argv[argc - 1]` — შეიცავს მომხმარებლის მიერ გადაცემულ არგუმენტებს.
- `argv[argc]` — ყოველთვის არის `NULL`.

6.10.1 პრაქტიკული მაგალითები: ბრძანებათა სტრიქონის არგუმენტები

მაგალითი 1: ყველა არგუმენტის ეკრანზე გამოტანა

კოდი 6.5 არგუმენტების გადაცემა

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[])
4  {
5
6   printf("gadacemuli argumentebis raodenoba (argc) = %d\n", argc);
7
8   for (int i = 0; i < argc; ++i)
9   {
10   printf("argv[%d] = %s\n", i, argv[i]);
11  }
12
13  return 0;
14 }
```

ტერმინალიდან გაშვების და შედეგის მაგალითი:

```
c:\users\username>    ./program.exe hello world 123
```

შედეგად მივიღებთ ასეთ სურათს:

```
c:\users\username>   gadacemuli argumentebis raodenoba (argc) = 4
c:\users\username>   argv[0] = ./program.exe
c:\users\username>   argv[1] = hello
c:\users\username>   argv[2] = world
c:\users\username>   argv[3] = 123
```

მაგალითი 2: ორი რიცხვის შეკრება ბრძანებათა სტრიქონიდან

რადგან argv მასივის ელემენტები სტრიქონებია (char*), მათი რიცხვად გადასაყვანად გამოიყენება stdlib.h ბიბლიოთეკის ფუნქცია atoi() (მთელი რიცხვებისთვის) ან atof() (წილადი რიცხვებისთვის).

კოდი 6.6 ორი რიცხვის შეკრება

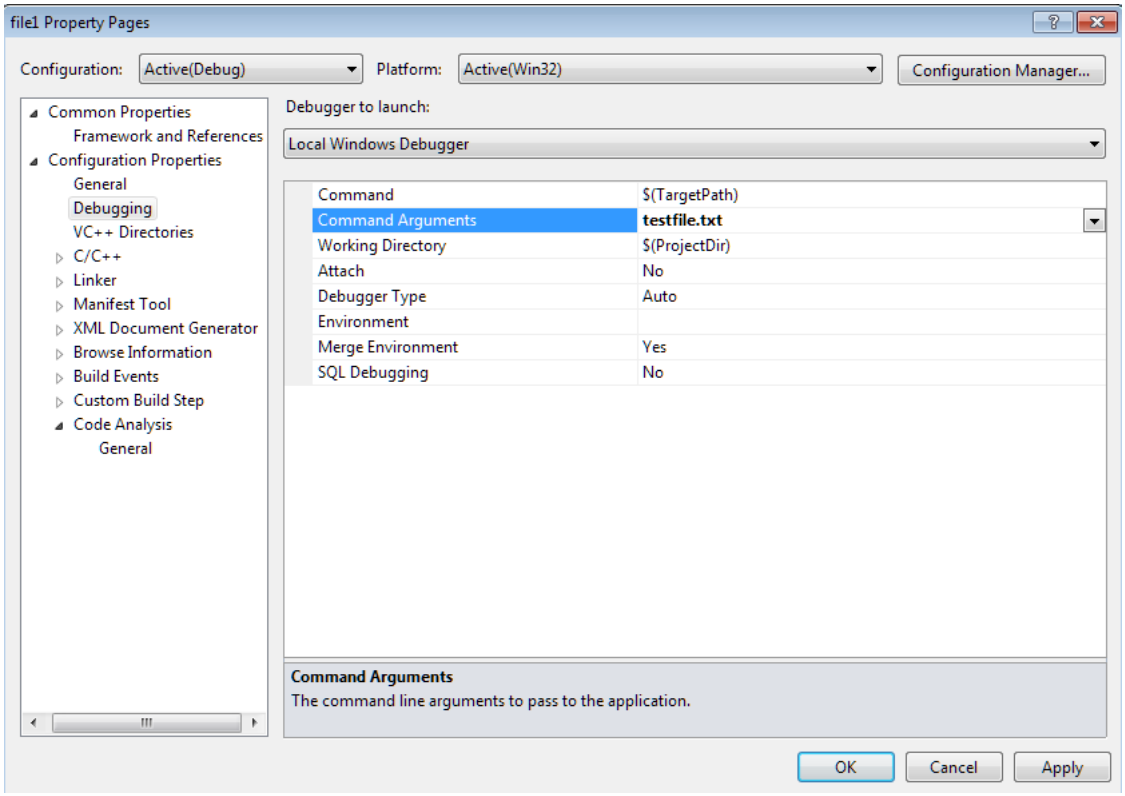
```c
#include <stdio.h>
#include <stdlib.h> // atoi() ფუნქციისთვის

int main(int argc, char* argv[]) {
 // ვამოწმებთ, გადასცა თუ არა მომხმარებელმა ზუსტად 2 რიცხვი
 if (argc != 3) {
  printf("shecdoma: gtxovt sheikvanot zustad 2 argumenti!\n");
  printf("gamokenebis cesi: %s <num1> <num2>\n", argv[0]);
  return 1;
 }

 int a = atoi(argv[1]); // პირველი არგუმენტი მთელ რიცხვად
 int b = atoi(argv[2]); // მეორე არგუმენტი მთელ რიცხვად

 printf("ჯამი: %d + %d = %d\n", a, b, a + b);

 return 0;
}
```

რჩევა უსაფრთხოებაზე: atoi() vs strtol()

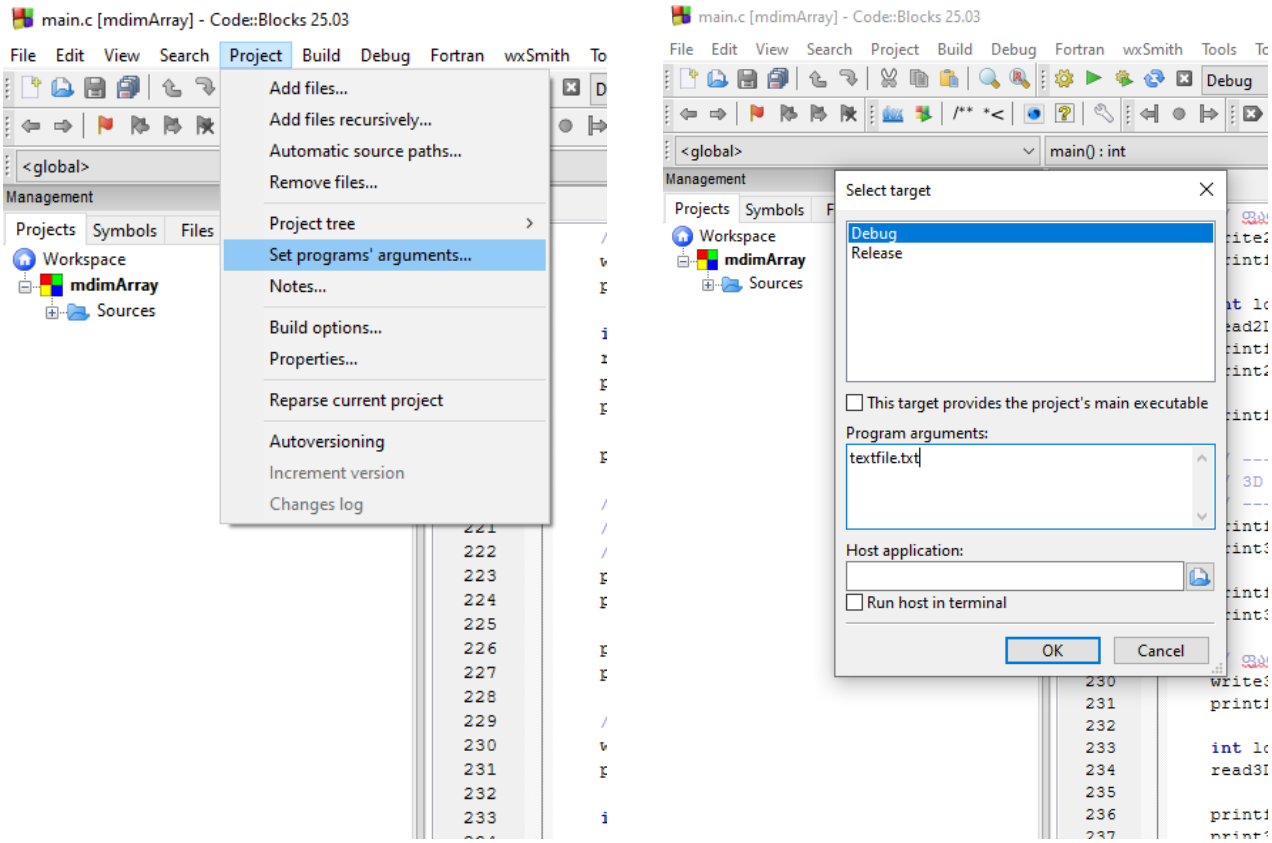
atoi() ფუნქცია მარტივია, თუმცა არასანდო შეყვანისას (მაგ. როცა რიცხვის ნაცვლად ტექსტს გადავცემთ) შეცდომას ვერ აფიქსირებს და უბრალოდ 0-ს აბრუნებს. რეალურ/პროფესიონალურ პროექტებში რეკომენდებულია strtol() (String to Long) ფუნქციის გამოყენება, რომელიც საშუალებას იძლევა შემოწმდეს, იყო თუ არა არგუმენტი ვალიდური რიცხვი.

დებაგირებისათვის პროგრამას შეიძლება არგუმენტები მიეწოდოს პროგრამული გარემოს (IDE) ფარგლებშიც. Visual Studio-ში არგუმენტების მიწოდება ნაჩვენებია სურ. 6.6,

ხოლო Code::Blocks გარემოში — სურ. 6.7.



სურ. 6.6 არგუმენტის მიწოდება პროექტიდან [Visual Studio]



ა)მენიუ ბ)პარამეტრების მიცემა

სურ. 6.7 პროგრამის არგუმენტები [CodeBlocks]

6.11 სავარჯიშოები

დავალება 6.9: 3D მასივის ინდექსაცია

დაწერეთ C პროგრამა, რომელიც ინიციალიზაციას გაუკეთებს $3 \times 3 \times 4$ ზომის სტატიკურ 3D მასივს.

1. შექმენით ფუნქცია, რომელიც იღებს ერთგანზომილებიან მიმთითებელს (`int *ptr`) და ინდექსებს (k, i, j) და გამოითვლის შესაბამის 1D ინდექსს ფორმულით:

$$\text{idx} = k \cdot 12 + i \cdot 4 + j$$

2. დაბეჭდეთ ელემენტის მნიშვნელობა და მისი მისამართი მეხსიერებაში.

დავალება 6.10: ბრძანებათა სტრიქონით ფაილების ასლი (File Copy via `argc`/`argv`)

დაწერეთ პროგრამა, რომელიც მიიღებს 2 არგუმენტს ბრძანებათა სტრიქონიდან: `./mycopy source.txt destination.txt`

1. შეამოწმეთ, რომ $argc == 3$. თუ არგუმენტები აკლია, გამოიტანეთ შეტყობინება ინსტრუქციით.
2. გახსენით `source.txt` წასაკითხად და `destination.txt` ჩასაწერად.
3. დააკოპირეთ ფაილის შიგთავსი სიმბოლო-სიმბოლო (`fgetc` / `fputc`).

დავალება 6.11: 2D მასივის ტრანსპონირება მიმთითებლებით

დაწერეთ ფუნქცია, რომელიც იღებს $N \times M$ 2D მასივს (გადაცემულს ტიპის დაყვანით `int *arr`) და ბეჭდავს მის ტრანსპონირებულ მატრიცას ($M \times N$).

6.12 2D და 3D სტეკ მასივების ჩაწერა და წაკითხვა

სტატიკური (სტეკზე განთავსებული) მრავალგანზომილებიანი მასივების ფაილში ჩაწერისა და წაკითხვისას მნიშვნელოვანია გვახსოვდეს, რომ მეხსიერებაში მონაცემები ინახება უწყვეტად (Row-Major Order). შესაბამისად, 2D ან 3D მასივის ჩაწერილი ფაილიდან აღდგენისას საჭიროა ციკლების იმავე თანმიმდევრობით გაშვება.

ქვემოთ მოყვანილია 2D და 3D სტატიკური მასივების ფაილურ სისტემასთან მუშაობის სრული მაგალითი:

კოდი 6.7 2D და 3D სტეკ მასივების ჩაწერა და წაკითხვა

```c
#include <stdio.h>
#include <stdlib.h>

#define DEPTH 2
#define ROWS  3
#define COLS  4

int main(void) {
 FILE *fp;
 int arr3D[DEPTH][ROWS][COLS];
 int restored3D[DEPTH][ROWS][COLS];
 int k, i, j;

 // 1. 3D მასივის ინიციალიზაცია
 int count = 1;
 for (k = 0; k < DEPTH; k++) {
  for (i = 0; i < ROWS; i++) {
   for (j = 0; j < COLS; j++) {
    arr3D[k][i][j] = count++;
   }
  }
 }

 // 2. 3D მასივის ჩაწერა ტექსტურ ფაილში
 fp = fopen("array3d_data.txt", "w");
 if (fp == NULL) {
```

```
27      perror("Error opening file for writing");
28      return EXIT_FAILURE;
29    }
30
31    for (k = 0; k < DEPTH; k++) {
32      fprintf(fp, "# Layer %d\n", k);
33      for (i = 0; i < ROWS; i++) {
34        for (j = 0; j < COLS; j++) {
35          fprintf(fp, "%d ", arr3D[k][i][j]);
36        }
37        fprintf(fp, "\n");
38      }
39    }
40    fclose(fp);
41    printf("3D Array written to file successfully.\n");
42
43    // 3. 3D მასივის წაკითხვა ფაილიდან
44    fp = fopen("array3d_data.txt", "r");
45    if (fp == NULL) {
46      perror("Error opening file for reading");
47      return EXIT_FAILURE;
48    }
49
50    char line_buf[100];
51    for (k = 0; k < DEPTH; k++) {
52      // გამოვტოვოთ თავსართის სტრიქონი (# Layer K)
53      fscanf(fp, "%99[^\n]\n", line_buf);
54      for (i = 0; i < ROWS; i++) {
55        for (j = 0; j < COLS; j++) {
56          fscanf(fp, "%d", &restored3D[k][i][j]);
57        }
58      }
59    }
60    fclose(fp);
61
62    // 4. აღდგენილი მასივის ბეჭდვა
63    printf("\n--- Restored 3D Array ---\n");
64    for (k = 0; k < DEPTH; k++) {
65      printf("Layer %d:\n", k);
66      for (i = 0; i < ROWS; i++) {
```

```
    for (j = 0; j < COLS; j++) {
     printf("%3d ", restored3D[k][i][j]);
    }
    printf("\n");
   }
   printf("\n");
  }

  return 0;
}
```

დინამიურად ალოკაციისას გვაქვს ორი მეთოდი: მიმთითებლების მასივი (Array of Pointers) და ორმაგი მიმთითებელი (Pointer to Pointer). ეს მეთოდები გამოდგება ასევე C99 თავსებადი კომპილატორის შემთხვევაშიც, როცა საჭიროა 2D მასივის დინამიური ალოკაცია `malloc()` ფუნქციის საშუალებით.

6.13 მასივის ფორმატირებული ჩაწერა/წაკითხვა

პრაქტიკულ ამოცანებში ხშირად გვხვდება ფაილები, სადაც მონაცემებამდე ჩაწერილია დამხმარე ინფორმაცია — ე.წ. **თავსართი (*Header*)**. თავსართი შეიცავს მეტამონაცემებს (მაგალითად: ფაილის აღწერას, მასივის განზომილებებს `xsize`, `ysize` და ა.შ.), ხოლო მის შემდეგ მოდის უშუალოდ რიცხვითი მონაცემები.

ქვემოთ მოყვანილ მაგალითში ორგანზომილებიანი მასივი იწერება ფაილში სპეციალური თავსართით, ხოლო შემდეგ ხდება ამ ფაილის წაკითხვა, პარამეტრების დინამიკური ანალიზი (*Parsing*) და მასივის აღდგენა.

კოდი 6.8 ორგანზომილებიანი მასივის ფაილში ჩაწერა და შერეული ფორმატით წაკითხვა

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

#define XSIZE 5
#define YSIZE 4

int main(void) {
 FILE *fp;
 const char *filename = "matrix_data.txt";
 double grid[XSIZE][YSIZE];
 char buffer[100];
 int nx = 0, ny = 0;
 int i, j, read_count;

```

```
15  // 1. მასივის ინიციალიზაცია საწყისი მნიშვნელობებით
16  for (i = 0; i < XSIZE; i++) {
17   for (j = 0; j < YSIZE; j++) {
18    grid[i][j] = (i + 1) * 10.0 + (j + 1) * 0.5;
19   }
20  }
21
22  // 2. ფაილის გახსნა ჩასაწერად ("w" მოდა)
23  fp = fopen(filename, "w");
24  if (fp == NULL) {
25   fprintf(stderr, "Error opening file %s for writing!\n", filename);
26   exit(EXIT_FAILURE);
27  }
28
29  // თავსართის (Header) ჩაწერა ფაილში
30  fprintf(fp, "# Header: Data Grid File\n");
31  fprintf(fp, "xsize=%d\n", XSIZE);
32  fprintf(fp, "ysize=%d\n", YSIZE);
33
34  // 2D მასივის ჩაწერა: ინდექსები და მნიშვნელობა (i,j,value)
35  for (i = 0; i < XSIZE; i++) {
36   for (j = 0; j < YSIZE; j++) {
37    fprintf(fp, "%d,%d,%.2lf\n", i, j, grid[i][j]);
38   }
39  }
40  fclose(fp);
41  printf("Data successfully written to %s\n\n", filename);
42
43  // 3. ოპერატიულ მეხსიერებაში მასივის განულება
44  for (i = 0; i < XSIZE; i++) {
45   for (j = 0; j < YSIZE; j++) {
46    grid[i][j] = 0.0;
47   }
48  }
49
50  // 4. ფაილის გახსნა წასაკითხად ("r" მოდა)
51  fp = fopen(filename, "r");
52  if (fp == NULL) {
53   fprintf(stderr, "Error opening file %s for reading!\n", filename);
54   exit(EXIT_FAILURE);
```

```
55  }
56
57  // პირველი კომენტარის ხაზის (#) გამოტოვება
58  fscanf(fp, "%99[^\r\n]\n", buffer);
59
60  // xsize=XX პარამეტრის წაკითხვა და ანალიზი
61  read_count = fscanf(fp, "%99[^=]=%d\n", buffer, &nx);
62  if (read_count != 2) {
63   fprintf(stderr, "Header error: Failed to parse xsize!\n");
64   fclose(fp);
65   exit(EXIT_FAILURE);
66  }
67
68  // ysize=YY პარამეტრის წაკითხვა და ანალიზი
69  read_count = fscanf(fp, "%99[^=]=%d\n", buffer, &ny);
70  if (read_count != 2) {
71   fprintf(stderr, "Header error: Failed to parse ysize!\n");
72   fclose(fp);
73   exit(EXIT_FAILURE);
74  }
75
76  printf("Header parsed successfully: grid dimensions = %dx%d\n", nx,
     ↪ ny);
77
78  // 5. მონაცემების წაკითხვა ციკლში ფაილის ბოლომდე
79  int r_i, r_j;
80  double val;
81  while ((read_count = fscanf(fp, "%d,%d,%lf\n", &r_i, &r_j, &val)) ==
     ↪  3) {
82   if (r_i >= 0 && r_i < XSIZE && r_j >= 0 && r_j < YSIZE) {
83    grid[r_i][r_j] = val;
84   }
85  }
86  fclose(fp);
87
88  // 6. წაკითხული და აღდგენილი მასივის ეკრანზე გამოტანა
89  printf("\nRestored Array Content from File:\n");
90  for (i = 0; i < nx; i++) {
91   for (j = 0; j < ny; j++) {
92    printf("%6.2lf ", grid[i][j]);
```

```
93    }
94    printf("\n");
95   }
96
97   return 0;
98  }
```

6.13.1 კოდის დეტალური განმარტება ხაზების მიხედვით

ხაზები 4–5: `#define XSIZE 5` და `#define YSIZE 4` განსაზღვრავს 2D მასივის სტატიკურ ზომებს კომპილაციის ეტაპზე.

ხაზები 15–20: ორმაგი `for` ციკლით ხდება `grid` მასივის შევსება სატესტო რიცხვითი მონაცემებით.

ხაზები 23–27: `fopen(filename, "w")` ხსნის ფაილს ჩასაწერად. თუ ფაილი არ არსებობს — იქმნება, ხოლო თუ არსებობს — მისი შინაარსი სრულად იშლება. შეცდომის შემთხვევაში `fp` აბრუნებს `NULL`-ს და პროგრამა სრულდება შეცდომის კოდით.

ხაზები 30–32: `fprintf` ფუნქციით ფაილში იწერება თავსართის 3 ხაზი:

- პირველი ხაზი: კომენტარი `# Header: Data Grid File`;
- მეორე ხაზი: ცვლადის სახელი და ზომა `xsize=5`;
- მესამე ხაზი: ცვლადის სახელი და ზომა `ysize=4`.

ხაზები 35–39: მასივის ელემენტები იწერება ფაილში მძიმით გამოყოფილ ფორმატში: `i,j,value` (მაგალითად: `0,0,10.50`).

ხაზები 44–48: `grid` მასივი მეხსიერებაში ივსება ნულებით. ეს კეთდება იმის დასამტკიცებლად, რომ შემდგომ ეტაპზე მონაცემები მართლაც ფაილიდან იკითხება და არ არის დარჩენილი ძველი მნიშვნელობები.

ხაზი 58: `fscanf(fp, "%99[^\r\n]\n", buffer);` — ბრძანება `%99[^\r\n]` ნიშნავს: *„წაიკითხე მაქსიმუმ 99 სიმბოლო ახალი სტრიქონის ან \r-\n შეხვედრამდე“*. ეს გამოიყენება პირველი კომენტარის ხაზის გამოტოვებისათვის.

ხაზები 61 და 68 — თავსართის სპეციალური ფორმატით წაკითხვა:

ბრძანება `fscanf(fp, "%99[^=]=%d\n", buffer, &nx)` იშლება სამ ნაწილად:

`%99[^=]` — კითხულობს ყველა სიმბოლოს ტოლობის (`=`) ნიშნამდე და ინახავს `buffer` მასივში (აქ ჩაიწერება ტექსტი `"xsize"`);

`=` — ითხოვს, რომ ფაილში ზუსტად ეწეროს ტოლობის ნიშანი;

`%d` — ტოლობის შემდეგ მდგომ მთელ რიცხვს კითხულობს და ინახავს `&nx` ცვლადში.

ხაზები 62–65 და 69–72 — წაკითხული არგუმენტების შემოწმება: `fscanf` აბრუნებს წარმატებით წაკითხული/შესაბამისობაში მოსული ველების რაოდენობას. რადგან ჩვენ 2 ველს ვკითხულობთ (`buffer` და `&nx`), `read_count`-ის მნიშვნელობა აუცილებლად უნდა იყოს 2. თუ ის არ უდრის 2-ს, ეს ნიშნავს, რომ ფაილის თავსართში ფორმატი დარღვეულია.

ხაზები 78–82 — მონაცემების წაკითხვა `while` ციკლში:

```
while ((read_count = fscanf(fp, "%d,%d,%lf\n", &r_i, &r_j, &val)) ==
    ↪ 3)
```

ეს არის ფაილის წაკითხვის ყველაზე საიმედო მეთოდი C ენაში:

- ყოველ ბიჯზე `fscanf` ცდილობს წაიკითხოს 3 ველი: ორი მთელი რიცხვი (`r_i`, `r_j`) და ერთი ათწილადი (`val`);
- სანამ ფაილში მონაცემები სწორი ფორმატითაა, `fscanf` დააბრუნებს 3-ს და ციკლი გაგრძელდება;
- როდესაც ფაილი დამთავრდება (EOF) ან შეხვდება არასწორი სიმბოლო, `fscanf` დააბრუნებს 3-ზე ნაკლებს და ციკლი ავტომატურად შეწყდება.

ხაზები 79–81: მოწმდება, რომ ფაილიდან წაკითხული ინდექსები `r_i` და `r_j` არ სცილდებოდეს გამოყოფილი მასივის საზღვრებს.

აქ საჭიროა ერთი დაზუსტება: აქ ჩვენ ვგულისხმობთ, რომ ფაილის ფორმატი მკაცრადაა დაცული, ანუ ხაზის დასაწყისიდან `'='` ნიშნამდე შეიძლება იყოს მაქსიმუმ 99 სიმბოლო.

6.13.2 წაკითხვა სიმბოლომდე — %99[^=]

`%99[^=]` ნიშნავს, რომ წაიკითხოს მხოლოდ 99 სიმბოლო და ჩაწეროს ბუფერში `buffer`. ეს ნიშნავს, რომ ბუფერს ვიცავთ გადავსებისგან. ეს მეთოდი იდეალურად მუშაობს, როცა მონაცემები შეგვყავს კონსოლიდან. 99 სიმბოლო წაიკითხება, დანარჩენი იგნორირდება. ბუფერი არის მინიმუმ 100-ელემენტიანი და ბოლო ელემენტი იქნება `'\0'` სიმბოლო.

ასე არ მოხდება, როცა მონაცემებს ვკითხულობთ ფაილიდან. ვთქვათ ფაილში წერია 200 სიმბოლო `'='` ნიშნამდე. მაშინ ზემოთ დაწერილი კოდის მიხედვით ბუფერში ჩაიწერება 99 სიმბოლო, ხოლო დანარჩენი სიმბოლოები (მე-100-დან 200-მდე) **ფაილიდან არ იშლება და არც ავტომატურად იგნორირდება**. ისინი რჩებიან ფაილის წაკითხვის რიგში (ნაკადში / stream-ში).

რა მოხდება პრაქტიკაში, თუ ტოლობამდე 200 ასო წერია?

თუ ფორმატად გვიწერია `"%99[^=]=%d"` და ფაილში წერია 200 ასო, რასაც მოსდევს `=10`:

1. `%99[^=]` წაიკითხავს პირველ 99 ასოს და ჩასვამს `buffer`-ში;
2. შემდეგ `fscanf` ელოდება, რომ ფაილში **99 ასოს შემდეგ შეხვდება `=` ნიშანი**;
3. მაგრამ ფაილის მიმთითებელი ამ დროს დგას **მე-100 ასოზე** (და არა `=` ნიშანზე);
4. `fscanf` ხედავს ასოს `=`-ის ნაცვლად, წყვეტს მუშაობას და `read_count` დააბრუნებს `1`-ს (ანუ მხოლოდ `buffer` წაიკითხა, ხოლო `&nx` ვერა).

თუ ასოების რაოდენობა < 99, კოდი მუშაობს უშეცდომოდ:

1. `%99[^=]` — წაიკითხავს ტექსტს პირველივე `=` ნიშნამდე (მაგალითად "width"), ჩაწერს მას `buffer`-ში და ბოლოში ავტომატურად დაუმატებს `\0`-ს. რადგან ტექსტი 99 სიმბოლოზე ნაკლებია, ის `=` ნიშნის დანახვისთანავე შეჩერდება (შეჩერდება ტოლობის ნიშნამდე).
2. `=` — `fscanf` ნაკადში წაიკითხავს და გამოტოვებს („გადაყლაპავს“) უშუალოდ ტოლობის ნიშანს.
3. `%d` — მომდევნო ნაბიჯზე წაიკითხავს ტოლობის შემდეგ მდგომ მთელ რიცხვს და ჩა-

წერს &nx ცვლადში (წინმსწრებ სარვეზებსაც ავტომატურად გამოტოვებს).

4. \n — რიცხვის შემდეგ გამოტოვებს დარჩენილ ახალ ხაზს (\n, \r) ან ინტერვალებს.

შეჯამება და ალტერნატივა: Assignment Suppression

- 99 იცავს მხოლოდ ოპერატიულ მეხსიერებას (მასივს) გადავსებისგან.
- ის **არ გადააბჭებს** მე-100-დან 200-მდე სიმბოლოს ფაილში.

თუ გვსურს, რომ ზედმეტი სიმბოლოები =-მდე საერთოდ გამოიტოვოს და ფაილშიც იგნორირება გაუკეთოს, ამისათვის გამოიყენება ვარსკვლავი * (*Assignment Suppression*):

```
// წაიკითხავს და იგნორირებას გაუკეთებს ყველა სიმბოლოს '='-მდე (თუნდაც 1000 იყ
    ↪ ოს)
 fscanf(fp, "%*[^=]=%d", &nx);
```

6.13.3 \r\n და კროსპლატფორმულობა fscanf()-ში

კოდში გვიწერია: fscanf(fp, "%99[^\r\n]\n", buffer);
რატომ გამოიყენება \r\n?

Windows-ისა და Linux/macOS-ის ოპერაციულ სისტემებში ახალი სტრიქონის აღმნიშვნელი სიმბოლოები განსხვავდება: Windows იყენებს CRLF (\r\n), ხოლო Unix-სისტემები — LF (\n). fscanf()-ით ფაილების წაკითხვისას ამ განსხვავებამ შესაძლოა ფარული შეცდომები გამოიწვიოს.

ტექსტური სტრიქონის წაკითხვა: %99[^\n] vs %99[^\r\n]

თუ ფაილიდან მთლიანი ტექსტური სტრიქონის წაკითხვა გსურთ, კროსპლატფორმულობისთვის აუცილებელია \r-ის გათვალისწინება:

```
1  // არა კროსპლატფორმული ჩანაწერი:
2  fscanf(fp, "%99[^\n]\n", buffer);
3
4  // სწორი, კროსპლატფორმული ჩანაწერი:
5  fscanf(fp, "%99[^\r\n]\n", buffer);
```

%99[^\n]-ის პრობლემა: ამბობს „წაიკითხე \n-მდე“. თუ Windows-ზე შექმნილ ფაილს წავიკითხავთ Linux-ზე, \r სიმბოლო ჩაიწერება buffer მასივის ბოლოს ("text\r"), რაც დაამახინჯებს მონაცემებს და შედარებებს შეაფერხებს.
%99[^\r\n]-ის უპირატესობა: ამბობს „წაიკითხე \r-მდე ან \n-მდე“. ამით buffer-ში არც \r მოხვდება და არც \n, მიუხედავად იმისა, თუ რომელ სისტემაზე შეიქმნა ფაილი.

ფორმატირებული მონაცემების წაკითხვა: %99[^=]=%d

როდესაც ვკითხულობთ კონკრეტულ ველებს, მაგალითად:

```
read_count = fscanf(fp, "%99[^=]=%d\n", buffer, &nx);
```

აქ კვადრატულ ფრჩხილებში \r-ის ჩამატება **საჭირო არ არის**, რადგან:

1. **დელიმიტერი (გამყოფი) (=)**: წაკითხვა წყდება = ნიშანთან, რომელიც სტრიქონის ბოლოს არსებულ \r\n-მდე ბევრად ადრე გვხვდება.
2. **Whitespace-ის შთანთქმა:** `fscanf`-ის ფორმატ სტრიქონში ნებისმიერი ხარვეზი ან \n მოქმედებს როგორც *Whitespace consumer* — ის ავტომატურად გამოტოვებს როგორც \n-ს, ისე \r-ს, \t-ს ან ინტერვალებს.
3. **%d-ს ყოფაქცევა:** %d სპეციფიკატორი რიცხვის წაკითხვამდე ავტომატურად იგნორირებას უკეთებს ნებისმიერ წინმსწრებ ხარვეზს.

წინა ხაზის ნარჩენებისგან დაცვა

საუკეთესო პრაქტიკაა ფორმატ სტრიქონის დასაწყისში ერთი ინტერვალის (Space) დასმა:

```
read_count = fscanf(fp, " %99[^=]=%d", buffer, &nx);
```

დასაწყისში არსებული ხარვეზი ავტომატურად შთანთქავს წინა სტრიქონის ბოლოდან დარჩენილ ნებისმიერ \n-ს, \r-ს თუ გამოტოვებულ ადგილებს.

6.14 სავარჯიშოები

დავალება 6.12: ბრძანებათა სტრიქონის კალკულატორი

დაწერეთ C პროგრამა, რომელიც ბრძანებათა სტრიქონიდან იღებს 3 არგუმენტს: `./calc <number1> <operator> <number2>` (მაგალითად: `./calc 12.5 + 3.5`).

1. შეამოწმეთ `argc` არგუმენტების რაოდენობაზე.
2. დაამუშავეთ ოპერაციები: `+`, `-`, `x` (ან `*`) და `/`.
3. გაითვალისწინეთ ნულზე გაყოფის შეცდომის შემოწმება.

დავალება 6.13: ფაილის თავსართის ანალიზი და მატრიცის ტრანსპონირება

შექმენით პროგრამა, რომელიც წაიკითხავს ტექსტურ ფაილს შემდეგი თავსართით:

```
# Matrix Output File
rows=3
cols=3
1 2 3
4 5 6
7 8 9
```

1. გამოიყენეთ `fscanf` თავსართიდან `rows` და `cols` მნიშვნელობების წასაკითხად.

2. წაიკითხეთ მატრიცა 2D მასივში, გამოითვალეთ ტრანსპონირებული მატრიცა და ჩაწერეთ ახალ ფაილში `"transposed.txt"` იგივე ფორმატის თავსართით.

დავალება 6.14: 3D მასივის ბინარული ჩაწერა და წაკითხვა

დაწერეთ პროგრამა, რომელიც $2 \times 3 \times 4$ ზომის 3D მასივს ჩაწერს ბინარულ ფაილში (`"array3d.bin"`) `fwrite()` ფუნქციით და შემდეგ წაიკითხავს `fread()`-ის გამოყენებით. შეადარეთ ბინარული ფაილის ზომა ტექსტური ფაილის ზომას.

თავი 7

შედგენილი ტიპები: struct და union

7.1 სტრუქტურები – struct 145
7.2 სტრუქტურის ცვლადების ინიციალიზაცია 148
7.3 მაგალითი: სტრუქტურების მასივის დალაგება 149
7.4 ჩადგმული სტრუქტურები (Nested Structs) 151
7.5 სტრუქტურები და ფუნქციები. სტრუქტურის მიმთითებელი 152
7.6 typedef და struct 153
7.6.1 სტრუქტურის ფორმატირებული ბეჭდვა 153
7.7 ბიტური ველები (Bit-fields) სტრუქტურაში 154
7.8 სტრუქტურები და დინამიკური მეხსიერება 155
7.9 გაერთიანება – union 155
7.10 სპეციალური მაგალითები: Vector3D და RGB/Color Union 157
7.10.1 მაგალითი : Vector3D (სახელით და მასივით წვდომა) 157
7.10.2 მაგალითი : RGBA Color (32-ბიტიანი Hex რიცხვი vs 8-ბიტიანი არხები) 158
7.11 სტრუქტურების და იუნიონების ფაილში ჩაწერა და წაკითხვა 159
7.11.1 ტექსტურ ფორმატში ჩაწერა და წაკითხვა (fprintf / fscanf) 159
7.11.2 ბინარულ ფორმატში ჩაწერა და წაკითხვა (fwrite / fread) 160

C ენა საშუალებას იძლევა შემოტანილ იქნას ახალი, მომხმარებლის მიერ განსაზღვრული შედგენილი ტიპები: სტრუქტურები (`struct`) და გაერთიანებები (`union`).

7.1 სტრუქტურები – struct

საბაზო ტიპები (მაგალითად, `int`, `float`, `char`) საშუალებას გვაძლევს შევქმნათ ცალკეული ობიექტები. როცა მოცემული გვაქვს ერთი და იმავე ტიპის ბევრი ობიექტი, C ენა გვთავაზობს მათ გაერთიანებას მასივში. თუმცა, რეალურ ამოცანებში ხშირად გვხვდება რთული ობიექტები, რომლებიც ხასიათდებიან სხვადასხვა ტიპის თვისებებით (მაგალითად: სახელი, ასაკი, ხელფასი).

თუ ასეთი ობიექტის თითოეული თვისებისთვის ცალ-ცალკე მასივს შემოვიღებთ (პარალელური მასივები), ობიექტის წაშლა, დამატება ან დალაგება მოითხოვს ყველა მასივში ცვლილების ერთდროულად განხორციელებას, რაც კოდს რთულად შესანახსა და შეცდომებისადმი მოწყვლადს ხდის.

ამ პრობლემის გადასაჭრელად C ენაში გამოიყენება სტრუქტურა (struct). სტრუქტურა საშუალებას იძლევა სხვადასხვა ტიპის ცვლადები გაერთიანდეს ერთ ლოგიკურ ობიექტში.

```
struct Employee
{
 char m_Name[50];
 int m_BirthDateDay;
 int m_BirthDateMonth;
 int m_BirthDateYear;
 int m_Salary;
};
```

ზემოთ დეკლარირებულია სტრუქტურა Employee (თანამშრომელი), რომელიც შედგება 50-ელემენტიანი char მასივისგან (m_Name) და ოთხი int ტიპის ველისგან (დაბადების დღე, თვე, წელი და ხელფასი).

სიტყვა struct არის ინსტრუქცია, რომელიც აცხადებს, რომ ფრჩხილებში ({ }) მოთავსებული ცვლადები წარმოადგენენ ახალი Employee ტიპის წევრებს (ველებს).

სტრუქტურა არის კომპლექსური მონაცემთა ტიპი. ის შეიძლება შედგებოდეს როგორც ბაზური ტიპის ცვლადებისგან, ასევე მასივებისგან, მიმთითებლებისგან და სხვა სტრუქტურებისგან.

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

struct Employee
{
 char m_Name[50];
 int m_BirthDateDay;
 int m_BirthDateMonth;
 int m_BirthDateYear;
 int m_Salary;
};

int main(void)
{
 struct Employee X;

 strcpy(X.m_Name, "George Saakadze");
 X.m_BirthDateYear = 1980;
 X.m_BirthDateMonth = 12;
 X.m_BirthDateDay = 22;
```

```
 X.m_Salary = 5000;

 printf("Employee X: %s, Date: %d/%d/%d, Salary: %d\n",
 X.m_Name, X.m_BirthDateDay, X.m_BirthDateMonth,
 X.m_BirthDateYear, X.m_Salary);

 return 0;
}
```

სტრუქტურის ცვლადის შექმნის შემდეგ, მის კონკრეტულ წევრზე მიმართვა ხდება წერტილის (`.`) ოპერატორის საშუალებით. მაგალითად, `X.m_Salary` ნიშნავს `X` სტრუქტურის შიგნით არსებულ `m_Salary` ველს.

შესაძლებელია როგორც სტრუქტურის ცალკეულ ველებთან მუშაობა, ასევე მთლიანი სტრუქტურის მინიჭება იმავე ტიპის სხვა სტრუქტურაზე:

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

struct Employee
{
 char m_Name[50];
 int m_BirthDateDay;
 int m_BirthDateMonth;
 int m_BirthDateYear;
 int m_Salary;
};

int main(void)
{
 struct Employee X, Y, Z;

 strcpy(X.m_Name, "John Doe");
 X.m_BirthDateDay = 22;
 X.m_BirthDateMonth = 12;
 X.m_BirthDateYear = 1980;
 X.m_Salary = 5000;

 /* წევრ-წევრად კოპირება */
 strcpy(Y.m_Name, X.m_Name);
 Y.m_BirthDateDay = X.m_BirthDateDay;
```

```
 Y.m_BirthDateMonth = X.m_BirthDateMonth;
 Y.m_BirthDateYear = X.m_BirthDateYear;
 Y.m_Salary = X.m_Salary;

 /* პირდაპირი მინიჭება - აკოპირებს მთლიან ბაიტურ მეხსიერებას */
 Z = X;

 printf("Z: %s, %d/%d/%d, %d\n", Z.m_Name,
 Z.m_BirthDateDay, Z.m_BirthDateMonth,
 Z.m_BirthDateYear, Z.m_Salary);

 return 0;
}
```

მინიჭების ოპერატორი (`=`): ერთი და იმავე ტიპის სტრუქტურების მინიჭებისას (`Z = X;`), კომპილატორი ავტომატურად ახორციელებს სტრუქტურის ბაიტების პირდაპირ კოპირებას. ეს ბევრად სწრაფი და მოსახერხებელია, ვიდრე ველების თითო-თითოდ კოპირება.

7.2 სტრუქტურის ცვლადების ინიციალიზაცია

სტრუქტურის გამოცხადებისთანავე მისი ველების ინიციალიზაცია შესაძლებელია ფიგურული ფრჩხილების გამოყენებით:

```
#include <stdio.h>

struct Employee
{
 char m_Name[50];
 int m_BirthDateDay;
 int m_BirthDateMonth;
 int m_BirthDateYear;
 int m_Salary;
};

int main(void)
{
 /* მიმდევრობითი ინიციალიზაცია */
 struct Employee X = {"John Doe", 22, 12, 1980, 5000};

 /* C99 დასახელებული ინიციალიზაცია (Designated Initializer) */
 struct Employee Y = {
```

```
    .m_Name = "Alice Smith",
    .m_Salary = 6000,
    .m_BirthDateYear = 1992,
    .m_BirthDateMonth = 5,
    .m_BirthDateDay = 15
  };

  printf("X: %s, Salary: %d\n", X.m_Name, X.m_Salary);
  printf("Y: %s, Salary: %d\n", Y.m_Name, Y.m_Salary);
  return 0;
}
```

7.3 მაგალითი: სტრუქტურების მასივის დალაგება

ქვემოთ მოყვანილ კოდში შემოტანილია `struct Employee` ტიპის მასივი. პროგრამა იღებს მონაცემებს და ახარისხებს მასივს ზრდადობით, დაბადების თარიღის მიხედვით:

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

struct Employee
{
  char m_Name[50];
  int m_BirthDateDay;
  int m_BirthDateMonth;
  int m_BirthDateYear;
  int m_Salary;
};

int main(void)
{
  struct Employee employees[100], tempEmployee;
  int count = 0, i, j;
  char continueChar;

  do
  {
    printf("Enter Employee Full Name: ");
    scanf(" %[^\n]", employees[count].m_Name);

    printf("Enter Birth Date (day month year): ");
```

```
25    scanf("%d %d %d",
26    &employees[count].m_BirthDateDay,
27    &employees[count].m_BirthDateMonth,
28    &employees[count].m_BirthDateYear);
29
30    printf("Enter Salary: ");
31    scanf("%d", &employees[count].m_Salary);
32
33    count++;
34    if (count == 100) break;
35
36    printf("Add another? (y/n): ");
37    scanf(" %c", &continueChar);
38  } while (continueChar == 'y' || continueChar == 'Y');
39
40  /* სორტირება დაბადების თარიღით (Bubble Sort) */
41  for (i = 0; i < count - 1; i++)
42  {
43    for (j = i + 1; j < count; j++)
44    {
45      if (employees[i].m_BirthDateYear > employees[j].m_BirthDateYear ||
46      (employees[i].m_BirthDateYear == employees[j].m_BirthDateYear &&
47      employees[i].m_BirthDateMonth > employees[j].m_BirthDateMonth) ||
48      (employees[i].m_BirthDateYear == employees[j].m_BirthDateYear &&
49      employees[i].m_BirthDateMonth == employees[j].m_BirthDateMonth &&
50      employees[i].m_BirthDateDay > employees[j].m_BirthDateDay))
51      {
52        /* მთლიანი სტრუქტურების გაცვლა */
53        tempEmployee = employees[i];
54        employees[i] = employees[j];
55        employees[j] = tempEmployee;
56      }
57    }
58  }
59
60  printf("\nSorted Employees List:\n");
61  for (i = 0; i < count; i++)
62  {
63    printf("%-20s\t%02d/%02d/%d\t%d\n",
64    employees[i].m_Name,
```

```
65    employees[i].m_BirthDateDay,
66    employees[i].m_BirthDateMonth,
67    employees[i].m_BirthDateYear,
68    employees[i].m_Salary);
69   }
70   return 0;
71  }
```

7.4 ჩადგმული სტრუქტურები (Nested Structs)

სტრუქტურის ველად შესაძლებელია სხვა სტრუქტურის გამოყენებაც. ლოგიკურად დაკავშირებული მონაცემების (მაგალითად, თარიღის) ცალკე სტრუქტურაში გატანა კოდს უფრო ორგანიზებულს ხდის:

```
1  struct TDate
2  {
3   int m_Day;
4   int m_Month;
5   int m_Year;
6  };
7
8  struct Employee
9  {
10  char m_Name[50];
11  struct TDate m_BirthDate;
12  struct TDate m_GraduationDate;
13  int m_Salary;
14 };
```

ჩადგმული სტრუქტურის ინიციალიზაცია და ველებზე მიმართვა:

```
1  struct Employee X = {
2   "John Doe",
3   {22, 12, 1990}, /* m_BirthDate */
4   {2, 7, 2012},   /* m_GraduationDate */
5   5000
6  };
7
8  /* ჩადგმული სტრუქტურის ველზე მიმართვა წერტილების ჯაჭვით */
9  printf("Graduation Year: %d\n", X.m_GraduationDate.m_Year);
```

7.5 სტრუქტურები და ფუნქციები. სტრუქტურის მიმთითებელი

სტრუქტურა ფუნქციას შეიძლება გადაეცეს ორი გზით:

1. **მნიშვნელობით (By Value):** იქმნება სტრუქტურის სრული ასლი. თუ სტრუქტურა დიდია, კოპირება ანელებს პროგრამას და იკავებს მეხსიერებას.
2. **მისამართით/მიმთითებლით (By Pointer):** ფუნქციას გადაეცემა მხოლოდ მეხსიერების მისამართი (4 ან 8 ბაიტი), რაც უზრუნველყოფს მაღალ წარმადობას.

როდესაც გვაქვს მიმთითებელი სტრუქტურაზე, მის ველებზე მიმართვისთვის გამოიყენება ისრის ოპერატორი (`->`):

```c
#include <stdio.h>

struct TDate
{
 int m_Day;
 int m_Month;
 int m_Year;
};

struct Employee
{
 char m_Name[50];
 struct TDate m_BirthDate;
 int m_Salary;
};

/* ფუნქცია იღებს მიმთითებელს - ეფექტურია მეხსიერების თვალსაზრისით */
void PrintEmployee(const struct Employee* pEmp)
{
 printf("Name: %s\n", pEmp->m_Name);
 printf("Birth Date: %02d/%02d/%d\n",
 pEmp->m_BirthDate.m_Day,
 pEmp->m_BirthDate.m_Month,
 pEmp->m_BirthDate.m_Year);
 printf("Salary: %d\n", pEmp->m_Salary);
}

int main(void)
{
 struct Employee X = {"John Doe", {22, 12, 1990}, 5000};
 PrintEmployee(&X); /* გადაეცემა მისამართი */
```

```
32  return 0;
33 }
```

7.6 typedef და struct

typedef საკვანძო სიტყვა საშუალებას იძლევა შევქმნათ ტიპის ალტერნატიული მოკლე სახელი, რათა ყოველ ჯერზე არ მოგვიწიოს struct სიტყვის წერა:

```
1  typedef struct {
2   float x;
3   float y;
4   float z;
5  } Vector3D;
6
7  int main(void)
8  {
9   Vector3D v1 = {1.0f, 2.5f, 3.0f};
10  Vector3D v2 = {0.0f, -1.0f, 4.0f};
11  return 0;
12 }
```

7.6.1 სტრუქტურის ფორმატირებული ბეჭდვა

სივრცული ვექტორისთვის $v = (x, y, z)$ შეგვიძლია განვსაზღვროთ Vector3D სტრუქტურა. ეკრანზე მონაცემების ლამაზად (სვეტებად) გასასწორებლად ვიყენებთ %8.2f ჩანაწერს:

8 — გამოყოფს ზუსტად 8 სიმბოლოს სიგანის ველს (საჭიროებისას მარცხნივ ამატებს სარეებს).

.2 — მძიმის შემდეგ ბეჭდავს 2 ციფრს.

სრული C კოდის მაგალითი:

```
1  #include <stdio.h>
2  #include <complex.h>
3
4  // Vector3D სტრუქტურის განსაზღვრა
5  typedef struct {
6   double x;
7   double y;
8   double z;
9  } Vector3D;
10
11 int main(void) {
12  // 1. კომპლექსური რიცხვის დაბეჭდვა %+.2f დროშით
```

```c
 double complex x2 = 3.5 - 4.2 * I;
 printf("z1 = %.2f %+.2fi\n\n", creal(x2), cimag(x2));

 // 2. Vector3D ცვლადების ინიციალიზაცია
 Vector3D v1 = {12.34, -5.6, 128.9};
 Vector3D v2 = {0.5, 104.12, -7.89};

 // 3. ეკრანზე გამოყვანა 8-პოზიციიანი ველით (%8.2f)
 printf("Vector3D v1: [%8.2f, %8.2f, %8.2f]\n", v1.x, v1.y, v1.z);
 printf("Vector3D v2: [%8.2f, %8.2f, %8.2f]\n", v2.x, v2.y, v2.z);

 return 0;
}
```

7.7 ბიტური ველები (Bit-fields) სტრუქტურაში

C ენაში შესაძლებელია მივუთითოთ, თუ ზუსტად რამდენ ბიტს იკავებს სტრუქტურის ესა თუ ის ცვლადი. ეს გამოსადეგია მეხსიერების მკაცრი ეკონომიისთვის ან აპარატურულ დონეზე (Hardware Registers) მუშაობისას:

კოდი 7.2 ბიტური ველების მაგალითი

```c
#include <stdio.h>

typedef struct {
 unsigned int isVisible   : 1;  // 1 ბიტი (0 ან 1)
 unsigned int isEnabled   : 1;  // 1 ბიტი
 unsigned int drawMode    : 3;  // 3 ბიტი (0-დან 7-მდე)
 unsigned int priority    : 4;  // 4 ბიტი (0-დან 15-მდე)
} ElementFlags;

int main(void) {
 ElementFlags flag = {1, 1, 5, 12};
 printf("Size of struct: %zu bytes\n", sizeof(ElementFlags));
 printf("Flags: Visible=%d, Enabled=%d, Mode=%d, Priority=%d\n",
        flag.isVisible, flag.isEnabled, flag.drawMode, flag.priority)
  ↪ ;
 return 0;
}
```

7.8 სტრუქტურები და დინამიკური მეხსიერება

სტრუქტურის ობიექტებისთვის მეხსიერების გამოყოფა დინამიკურადაც არის შესაძლებელი `malloc` და `free` ფუნქციებით:

კოდი 7.3 დინამიკური სტრუქტურა

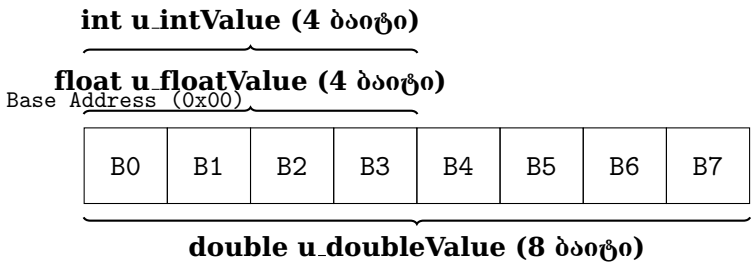
```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
 int id;
 float score;
} Student;

int main(void) {
 int n = 3;
 Student* list = (Student*)malloc(n * sizeof(Student));
 if (list == NULL) return 1;

 for (int i = 0; i < n; i++) {
  list[i].id = i + 1;
  list[i].score = (i + 1) * 25.5f;
 }

 for (int i = 0; i < n; i++) {
  printf("Student %d -> Score: %.2f\n", list[i].id, list[i].score);
 }

 free(list);
 return 0;
}
```

7.9 გაერთიანება - union

`union` (გაერთიანება) არის სპეციალური მონაცემთა ტიპი, სადაც ყველა წევრი **იზიარებს მეხსიერების ერთსა და იმავე არეალს**.

სტრუქტურისგან განსხვავებით, სადაც ზომა არის ველების ზომების ჯამი (პლიუს alignment padding), `union`-ის ზომა უდრის **მისი ყველაზე დიდი წევრის ზომას**.

```c
#include <stdio.h>

enum NUMType { INT, FLOAT, DOUBLE };
```



სურ. 7.1 მეხსიერების გადანაწილება union UNUMValue-ში (ყველა წევრი იწყება ერთი და იმავე მისამართიდან)

```
4
5  union UNUMValue
6  {
7   int u_intValue;
8   float u_floatValue;
9   double u_doubleValue;
10 };
11
12 union UNUMValue Add(union UNUMValue val1, union UNUMValue val2, enum
     ↪ NUMType type)
13 {
14  union UNUMValue result;
15  switch(type)
16  {
17   case INT:
18   result.u_intValue = val1.u_intValue + val2.u_intValue;
19   break;
20   case FLOAT:
21   result.u_floatValue = val1.u_floatValue + val2.u_floatValue;
22   break;
23   case DOUBLE:
24   result.u_doubleValue = val1.u_doubleValue + val2.u_doubleValue;
25   break;
26  }
27  return result;
28 }
29
30 int main(void)
31 {
32  union UNUMValue V1, V2, R;
33  printf("Size of union UNUMValue: %zu bytes\n", sizeof(union UNUMValue
```

```
    ↪ ));
34  V1.u_intValue = 10;
35  V2.u_intValue = 20;
36  R = Add(V1, V2, INT);
37  printf("INT sum: %d\n", R.u_intValue);
38  V1.u_doubleValue = 12.5;
39  V2.u_doubleValue = 7.3;
40  R = Add(V1, V2, DOUBLE);
41  printf("DOUBLE sum: %lf\n", R.u_doubleValue);
42  return 0;
43 }
```

7.10 სპეციალური მაგალითები: Vector3D და RGB/Color Union

union-ისა და ანონიმური სტრუქტურების გაერთიანებით შესაძლებელია ძალიან მოხერხებული მონაცემთა სტრუქტურების შექმნა, სადაც ერთსა და იმავე მონაცემზე წვდომა სხვადასხვა სახით არის შესაძლებელი.

7.10.1 მაგალითი : Vector3D (სახელით და მასივით წვდომა)

ვექტორს შეგვიძლია შევხედოთ როგორც სამი `float` ცვლადის (`x, y, z`) ერთობლიობას ან როგორც `float v[3]` მასივს. `union` საშუალებას იძლევა გვქონდეს ვექტორის წარმოდგენა ორივე სახით:

კოდი 7.4 `union` Vector3D მაგალითი

```
1  #include <stdio.h>
2  typedef union {
3   struct {
4    float x;
5    float y;
6    float z;
7   };
8   float v[3]; /* იზიარებს იმავე მეხსიერებას, რასაც x, y, z */
9  } Vector3D;
10
11 int main(void)
12 {
13  Vector3D vec = {.x = 1.0f, .y = 2.0f, .z = 3.0f};
14
15  /* წვდომა სახელებით */
16  printf("Vector components: x=%.1f, y=%.1f, z=%.1f\n", vec.x, vec.y,
     ↪ vec.z);
```

```
17
18  /* წვდომა ციკლში მასივის ინდექსით */
19  printf("Vector via array loop: ");
20  for(int i = 0; i < 3; i++) {
21   printf("%.1f ", vec.v[i]);
22  }
23  printf("\n");
24
25  return 0;
26 }
```

7.10.2 მაგალითი : RGBA Color (32-ბიტიანი Hex რიცხვი vs 8-ბიტიანი არხები)

ფერი შეიძლება გვქონდეს როგორც ერთი 32-ბიტიანი რიცხვი ან როგორც 4 ბაიტი. union საშუალებას გვაძლევს ამ სტრუქტურას შევხედოთ და გამოვიყენოთ ორივე ფორმით:

კოდი 7.5 rgba ფერი union

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  typedef union {
5   uint32_t rgba; /* მთლიანი 32-ბიტიანი ფერის კოდი */
6   struct {
7    uint8_t r;
8    uint8_t g;
9    uint8_t b;
10   uint8_t a;
11  } channels;
12 } Color;
13
14 int main(void)
15 {
16  Color c;
17  c.rgba = 0xFF8000FF; /* Orange Color (HEX) */
18
19  printf("HEX Color: 0x%08X\n", c.rgba);
20  printf("Red: %d, Green: %d, Blue: %d, Alpha: %d\n",
21  c.channels.r, c.channels.g, c.channels.b, c.channels.a);
22
23  return 0;
24 }
```

7.11 სტრუქტურების და იუნიონების ფაილში ჩაწერა და წაკითხვა

სტრუქტურების ფაილში შენახვა და წაკითხვა შესაძლებელია ორი ძირითადი ფორმატით: **ტექსტური** (ადამიანისათვის მარტივად წაკითხვადი) და **ბინარული** (კომპიუტერისთვის ეფექტური, მეხსიერების ზუსტი ასლი).

7.11.1 ტექსტურ ფორმატში ჩაწერა და წაკითხვა (fprintf / fscanf)

ტექსტურ ფორმატში სტრუქტურის ველები იწერება როგორც სიმბოლოების მიმდევრობა (მაგალითად, ASCII). ეს ფორმატი მოსახერხებელია დებაგინგისთვის და ფაილის ტექსტურ რედაქტორში განსახილველად.

```c
#include <stdio.h>
#include <stdint.h>

typedef struct {
 uint32_t id;
 char name[32];
 float score;
} PlayerData;

int main(void)
{
 PlayerData p1 = {101, "Gamer_One", 98.5f};
 PlayerData p2;

 /* 1. ტექსტურ ფაილში ჩაწერა (fprintf) */
 FILE *txtOut = fopen("player.txt", "w");
 if (txtOut != NULL) {
  fprintf(txtOut, "%u %s %.2f\n", p1.id, p1.name, p1.score);
  fclose(txtOut);
  printf("Data written to text file.\n");
 }

 /* 2. ტექსტური ფაილიდან წაკითხვა (fscanf) */
 FILE *txtIn = fopen("player.txt", "r");
 if (txtIn != NULL) {
  fscanf(txtIn, "%u %31s %f", &p2.id, p2.name, &p2.score);
  fclose(txtIn);
  printf("Text File Read -> ID: %u, Name: %s, Score: %.2f\n",
  p2.id, p2.name, p2.score);
 }
```

```
31
32  return 0;
33 }
```

7.11.2 ბინარულ ფორმატში ჩაწერა და წაკითხვა (fwrite / fread)

ბინარულ ფორმატში სტრუქტურა ფაილში იწერება ზუსტად იმ ბაიტური სახით, როგორადაც ის ოპერატიულ მეხსიერებაშია განთავსებული. ეს მეთოდი ბევრად სწრაფია და იკავებს ნაკლებ ადგილს, თუმცა ფაილი ადამიანისთვის წაკითხვადი არ არის.

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  typedef struct {
5   uint32_t id;
6   char name[32];
7   float score;
8  } PlayerData;
9
10 int main(void)
11 {
12  PlayerData p1 = {101, "Gamer_One", 98.5f};
13  PlayerData p2;
14
15  /* 1. ბინარულ ფაილში ჩაწერა (fwrite) */
16  FILE *binOut = fopen("player.dat", "wb");
17  if (binOut != NULL) {
18   fwrite(&p1, sizeof(PlayerData), 1, binOut);
19   fclose(binOut);
20   printf("Data written to binary file.\n");
21  }
22
23  /* 2. ბინარული ფაილიდან წაკითხვა (fread) */
24  FILE *binIn = fopen("player.dat", "rb");
25  if (binIn != NULL) {
26   fread(&p2, sizeof(PlayerData), 1, binIn);
27   fclose(binIn);
28   printf("Binary File Read -> ID: %u, Name: %s, Score: %.2f\n",
29   p2.id, p2.name, p2.score);
30  }
31
```

```
32  return 0;
33 }
```

დავალება 7.1: სტრუქტურები და გაერთიანებები (პრაქტიკული ამოცანები)

1. დაწერეთ სტრუქტურა კომპლექსური რიცხვისთვის (`Complex`: რეალური და წარმოსახვითი ნაწილებით). შექმენით შეკრების, გამოკლების, გამრავლებისა და მოდულის გამოთვლის ფუნქციები.
2. შექმენით კვადრატული განტოლების ამოხსნის პროგრამა, სადაც ფუნქცია იღებს კოეფიციენტების სტრუქტურას და აბრუნებს ფესვებისა და სტატუსის (`enum`: NO_ROOTS, ONE_ROOT, TWO_ROOTS) შემცველ სტრუქტურას.
3. გადაწერეთ კვადრატული განტოლების ამოხსნა `union`-ის გამოყენებით, სადაც ფესვები შეიძლება იყოს როგორც მთელი, ისე ნამდვილი რიცხვები.
4. **ორი წერტილის მანძილი და 2D პოლიგონი:** შექმენით სტრუქტურა `Point2D` (x, y კოორდინატებით). დაწერეთ ფუნქცია, რომელიც იღებს `Point2D`-ების მასივს (მრავალკუთხედის წვეროებს) და ითვლის მის პერიმეტრს.
5. **სტუდენტების მონაცემთა ბაზა და ბინარული ფაილი:** შექმენით `Student` სტრუქტურა (ID, სახელი, საშუალო ქულა GPA). დაწერეთ პროგრამა, რომელიც N სტუდენტის მონაცემებს ჩაწერს ბინარულ ფაილში (`students.bin`), შემდეგ კი წაიკითხავს ამ ფაილს და ეკრანზე გამოიტანს მხოლოდ იმ სტუდენტებს, ვისი GPA-ც მეტია 3.5-ზე.
6. **Generic Value პატერნი `union`-ით:** შექმენით სტრუქტურა `VariantData`, რომელიც შეიცავს `enum DataType` ტიპის ველს (INT, FLOAT, STRING) და შესაბამის `union`-ს. დაწერეთ უნივერსალური ფუნქცია `PrintVariant(const VariantData* data)`, რომელიც სწორად დაბეჭდავს მონაცემს ველის ტიპის მიხედვით.

თავი 8

ცვლადები და მეხსიერება

8.1 მეხსიერების განაწილება C ენაში:სეგმენტები და მათი მახასიათებლები . 163
8.1.1 Text / Code Segment (კოდის სეგმენტი) 163
8.1.2 Data Segment (ინიციალიზებული მონაცემები) 164
8.1.3 BSS Segment (არაინიციალიზებული მონაცემები) 164
8.1.4 Heap (გროვა) 164
8.1.5 Stack (სტეკი) 164
8.2 ავტომატური ცვლადები და ხილვადობის არე 166
8.3 გლობალური და ლოკალური ცვლადები 167
8.4 გლობალური ცვლადების ხილვადობის არე და `static` ცვლადები 168
8.4.1 მრავალფაილიანი პროექტის მაგალითი 168
8.5 სტატიკური (`static`) ფუნქციები C ენაში. 170
8.6 `inline` და `static inline`-ის განსხვავება C და C++ ენებში. 173
8.7 ცვლადის კვალიფიკატორი `volatile` 178
8.8 ტიპის გარდაქმნა — კასტირება (Type Casting) 178
8.9 `void*` მიმთითებელი (Generic Pointer) 178
8.10 მეხსიერების დინამიური მართვა 179
8.10.1 `realloc()` ფუნქციის პრაქტიკული მაგალითი 180
8.11 დინამიური 2D და 3D მასივები (Flat Array Approach) 181
8.12 3D მასივის სტრუქტურა და ინკაფსულაცია 181
8.13 `goto` ოპერატორი და ნიშნულები (Labels). 182

8.1 მეხსიერების განაწილება C ენაში:სეგმენტები და მათი მახასიათებლები

პროგრამის გაშვებისას ოპერატიული მეხსიერება (RAM) იყოფა რამდენიმე ფუნქციონალურ სეგმენტად. იმის გაგება, თუ როგორ ინახება ცვლადები მეხსიერებაში, ფუნდამენტურია ეფექტური და უსაფრთხო C კოდის დასაწერად.

8.1.1 Text / Code Segment (კოდის სეგმენტი)

დანიშნულება: ინახავს პროგრამის მანქანურ კოდს (შესრულებად ინსტრუქციებს). მხოლოდ კითხვის რეჟიმშია (**Read-Only**).
ზომა: ფიქსირებული. დამოკიდებულია დაკომპილირებული პროგრამის კოდის მოცულობაზე.

სისწრაფე: ექსტრემალურად სწრაფი.

პროცესორთან სიახლოვე: ყველაზე ახლოს. ინსტრუქციები პირდაპირ იტვირთება პროცესორის ინსტრუქციების ქეშში (**I-Cache**).

8.1.2 Data Segment (ინიციალიზებული მონაცემები)

დანიშნულება: ინახავს **ინიციალიზებულ** გლობალურ და სტატიკურ (`static`) ცვლადებს.

ზომა: ფიქსირებული. განისაზღვრება კომპილაციისას.

სისწრაფე: ძალიან სწრაფი.

პროცესორთან სიახლოვე: RAM-ის დონეზე, ეფექტურად იტვირთება პროცესორის მონაცემთა ქეშში (**D-Cache**).

8.1.3 BSS Segment (არაინიციალიზებული მონაცემები)

დანიშნულება: ინახავს **არაინიციალიზებულ** გლობალურ და სტატიკურ ცვლადებს. ავტომატურად უთანაბრდება 0-ს.

ზომა: ფიქსირებული. შესრულებად ფაილში ადგილს არ იკავებს, RAM-ში გამოიყოფა გაშვებისას.

სისწრაფე: ძალიან სწრაფი.

პროცესორთან სიახლოვე: RAM-ის დონეზე / პროცესორის ქეში.

8.1.4 Heap (გროვა)

დანიშნულება: დინამიური მეხსიერების გამოყოფა პროგრამის შესრულებისას (`malloc`, `calloc`, `realloc`, `new`).

ზომა: ძალიან დიდი. შეზღუდულია მხოლოდ სისტემის ხელმისაწვდომი RAM-ით და ვირტუალური მეხსიერებით.

სისწრაფე: შედარებით **ნელი**. მეხსიერების გამოყოფა მოითხოვს OS-ის ალგორითმების ჩართვას.

პროცესორთან სიახლოვე: შორს პროცესორისგან. მონაცემები გაბნეულია, რის გამოც მაღალია ქეშის აცდენის (**Cache Miss**) ალბათობა.

8.1.5 Stack (სტეკი)

დანიშნულება: ინახავს ლოკალურ ცვლადებს, ფუნქციის პარამეტრებს და დაბრუნების მისამართებს (LIFO პრინციპი).

ზომა: შეზღუდული და მცირე (ნაგულისხმევად 1MB – 8MB).

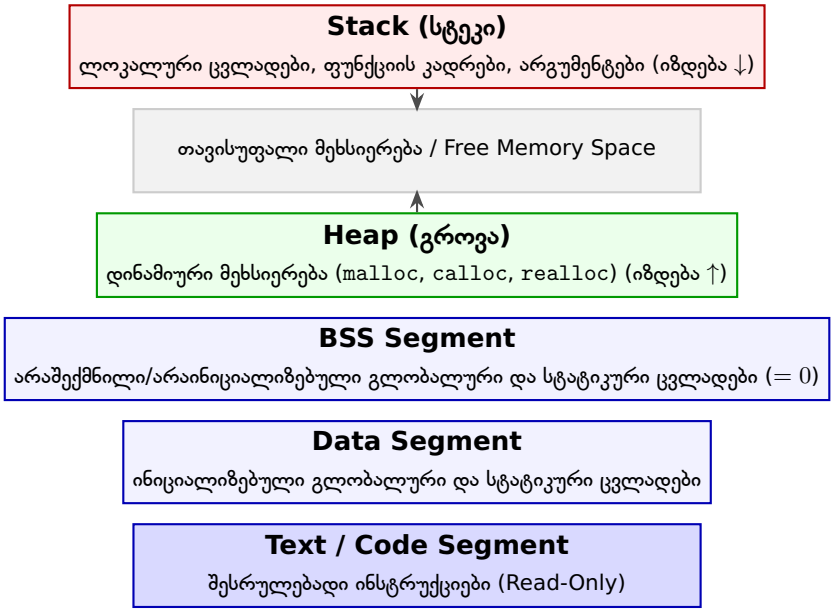
სისწრაფე: ძალიან სწრაფი. გამოყოფა ხდება უბრალოდ **Stack Pointer** რეგისტრის წანაცვლებით.

პროცესორთან სიახლოვე: ძალიან ახლოს. თითქმის ყოველთვის იმყოფება პროცესორის **L1/L2 ქეშში**.

ოპერაციული სისტემა პროგრამას გამოუყოფს ვირტუალურ მისამართთა სივრცეს. მისამართები იწყება დაბალი მნიშვნელობებიდან (მაგ. 0x00000000) და მიდის მაღალი მნიშვნელობებისკენ:

Stack (სტეკი) იზრდება ქვემოთ (↓): იწყება მაღალი მისამართებიდან. ახალი მონა-

ცემების დამატებისას ან ფუნქციების გამოძახებისას მისამართები მცირდება.
Heap (გროვა) იზრდება ზემოთ (↑): იწყება დაბალი მისამართებიდან (BSS სეგმენტის შემდეგ). დინამიური მეხსიერების გამოყოფისას მისამართები იზრდება.
მიზანი: სტეკი და ჰიპი მოძრაობენ ერთმანეთის მიმართულებით, რათა მათ შორის არსებული **თავისუფალი მეხსიერება (Free Memory Space)** მაქსიმალურად ეფექტურად და მოქნილად იქნას გამოყენებული.



სურ. 8.1 C პროგრამის მეხსიერების განაწილების სტრუქტურა (RAM Memory Layout)

კოდი 8.1 memory_layout_example.c

```c
#include <stdio.h>
#include <stdlib.h>

// 1. Data Segment: ინიციალიზებული გლობალური ცვლადი
int global_init_var = 42;

// 2. BSS Segment: არაინიციალიზებული გლობალური ცვლადი (= 0)
int global_uninit_var;

void demo_function(int param) { // param -> Stack
 // Data Segment: ინიციალიზებული სტატიკური ცვლადი
 static int static_init_var = 100;

 // BSS Segment: არაინიციალიზებული სტატიკური ცვლადი
 static int static_uninit_var;

 // Stack: ლოკალური ცვლადი
```

```
18   int local_var = 5;
19
20   // Heap: დინამიურად გამოყოფილი მეხსიერება
21   // თვითონ 'heap_ptr' ინახება Stack-ზე, გამოყოფილი მეხსიერება - Heap-ზე.
22   int *heap_ptr = (int *)malloc(sizeof(int) * 10);
23
24   if (heap_ptr != NULL) {
25    heap_ptr[0] = 77;
26    free(heap_ptr); // Heap მეხსიერების გათავისუფლება
27   }
28  }
29
30  // Text / Code Segment: კოდი და main ფუნქცია ინახება კოდის სეგმენტში
31  int main() {
32   demo_function(10);
33   return 0;
34  }
```

8.2 ავტომატური ცვლადები და ხილვადობის არე

როგორც ადრე აღვნიშნეთ, `int a = 3;` ტიპის ბრძანება ნიშნავს, რომ იქმნება `int` ტიპის ცვლადი a, რომელსაც ენიჭება მნიშვნელობა 3.

ასეთი გზით გამოცხადებულ/შექმნილ ცვლადებს **ავტომატური ცვლადები** ეწოდებათ, ვინაიდან ისინი იქმნებიან მეხსიერებაში (სტეკზე) ავტომატურად. ავტომატური ცვლადებისათვის შეიძლება გამოყენებულ იქნას სპეციფიკატორი `auto`, მაგრამ ეს ნაგულისხმევია და, როგორც წესი, გამოიტოვება.

როდის „ქრებიან“ ეს ცვლადები? ანუ როდის თავისუფლდება მეხსიერება მათგან? პასუხი „პროგრამის მუშაობის დამთავრებისას“ ყოველთვის სწორი არაა.

ფრჩხილებში `{...}` მოთავსებული ბრძანებები ქმნიან ცალკე ბლოკს. ავტომატური ცვლადები არსებობენ მხოლოდ იმ ბრძანებათა ბლოკში, სადაც ისინი შეიქმნენ. როგორც კი პროგრამა შესრულებისას გადის ბრძანებათა ბლოკიდან, მეხსიერება ავტომატურად თავისუფლდება.

კოდი 8.2 ავტომატური ცვლადები და ბლოკური ხილვადობა

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 3;
6      int b = 4;
7      printf("\na*b = %d * %d = %d", a, b, a * b);
```

```
8
9       {
10          int a = 5; // ფარავს გარეთა 'a' ცვლადს ამ ბლოკში
11          int b = 6;
12          printf("\nqveblokshi a*b = %d * %d = %d", a, b, a * b);
13          int c = 8;
14          printf("\nqveblokshi c = %d", c);
15      } // 'c' და შიდა 'a', 'b' ცვლადები აქ ნადგურდებიან
16
17      printf("\nisev gare bloki a*b = %d * %d = %d", a, b, a * b);
18      puts("\n");
19      return 0;
20  }
```

ასეთ ცვლადებს სხვაგვარად **ლოკალურ ცვლადებსაც** უწოდებენ. ბლოკის გარეთ განმარტებული ცვლადები ხელმისაწვდომია შიდა ბლოკში, ხოლო შიდა ბლოკში განმარტებული ცვლადები — მხოლოდ ამ ბლოკში.

გასათვალისწინებელია, რომ ცვლადი ხელმისაწვდომია მხოლოდ მისი დეკლარირების (ინიციალიზაციის) შემდეგ. არაინიციალიზებული ცვლადის გამოყენება შეცდომაა. მაგალითად, თუ ზემოთ მოყვანილ კოდში ქვებლოკის გარეთ დავწერთ `c = 7;`, ეს იქნება შეცდომა, ვინაიდან ასეთი ცვლადი ამ არეში არ არსებობს.

8.3 გლობალური და ლოკალური ცვლადები

ცვლადი შეიძლება შემოტანილ იქნას `main()` ფუნქციის გარეთაც. `main()` ან ნებისმიერ ფუნქციამდე განმარტებულ ცვლადებს **გლობალური ცვლადები** ეწოდებათ, ხოლო `{...}` ბლოკში განმარტებულ ცვლადებს — **ლოკალური**.

კოდი 8.3 გლობალური და ლოკალური ცვლადების გამოყენება

```
1  #include <stdio.h>
2
3  int a = 3; // გლობალური ცვლადი
4  int b = 4; // გლობალური ცვლადი
5
6  int mult_and_add(int _x, int _y)
7  {
8      return _x * _y + a; // იყენებს გლობალურ 'a'-ს
9  }
10
11 int main(void)
12 {
```

```
13      printf("\na*b = %d * %d = %d", a, b, a * b);
14      {
15          int tmpa = 5;
16          int tmpb = 6;
17          printf("\nqveblokshi tmpa*tmpb = %d * %d = %d", tmpa, tmpb,
    ↪ tmpa * tmpb);
18          int c = 8;
19          printf("\nqveblokshi c = %d", c);
20      }
21      printf("\nisev gare bloki a*b = %d * %d = %d", a, b, a * b);
22      puts("\n");
23      printf("\nResult: %d\n", mult_and_add(3, 5));
24      return 0;
25  }
```

გლობალური ცვლადები მიეკუთვნებიან *სტატიკური შენახვის ხანგრძლივობის* კლასს (*Static Storage Duration*), ხოლო ლოკალური ცვლადები — *ავტომატური შენახვის ხანგრძლივობის* კლასს (*Automatic Storage Duration*). გლობალური ცვლადები ავტომატურად ინიციალიზდებიან ნულოვანი მნიშვნელობით (0).

8.4 გლობალური ცვლადების ხილვადობის არე და `static` ცვლადები

საკვანძო სიტყვა `static` ნიშნავს, რომ ცვლადი განაგრძობს არსებობას ბრძანებათა ბლოკიდან პროგრამის გასვლის შემდეგაც. იგი არ ნადგურდება და არსებობს პროგრამის დასრულებამდე.

იმის მიხედვით, თუ სად გამოცხადდა ცვლადი, ის შეიძლება იყოს ლოკალური ან გლობალური:

ლოკალური `static` ცვლადი: ინარჩუნებს მნიშვნელობას ფუნქციის გამოძახებებს შორის, თუმცა ხილვადია მხოლოდ იმ ფუნქციის შიგნით.

გლობალური `static` ცვლადი: ხილვადია მხოლოდ იმ ფაილში (ტრანსლირების ერთეულში), სადაც გამოცხადდა (*Internal Linkage*).

სხვა ფაილში არსებული გლობალური ცვლადის გამოსაყენებლად გამოიყენება სიტყვა `extern` (*External Linkage*).

8.4.1 მრავალფაილიანი პროექტის მაგალითი

```
1  #include <stdio.h>
2  #include "util.h"
3
4  int d = 5;           // External linkage (ხელმისაწვდომია სხვა ფაილებშიც)
5  static int e = 10;   // Internal linkage (მხოლოდ ამ ფაილისთვის)
6
```

```c
extern int b;        // გამოცხადებულია სხვა ფაილში (util.c)

int main(void)
{
    b = 8;
    a = 7; // util.h-ში არსებული static ცვლადი

    {
        int a = 5;
        int b = 6;
        int c = 8;
        printf("\nqveblokshi a*b = %d * %d = %d", a, b, a * b);
        printf("\nqveblokshi c = %d", c);
    }

    puts("\n");
    printf("\n1st call: %d\n", mult_and_add(3, 5));
    printf("\n2nd call: %d\n", mult_and_add(3, 5));
    return 0;
}
```

კოდი 8.5 util.h სათაო ფაილი

```c
#ifndef TEST_UTIL_H
#define TEST_UTIL_H

static int a = 3; // ყოველი #include შექმნის თავის ლოკალურ ასლს

#ifdef __cplusplus
extern "C" {
#endif

int mult_and_add(int _x, int _y);

#ifdef __cplusplus
}
#endif

#endif
```

კოდი 8.6 util.c იმპლემენტაციის ფაილი

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

int b = 4; // გლობალური ცვლადი
extern int d; // main.c-ში გამოცხადებული d ცვლადი

int mult_and_add(int _x, int _y)
{
    static int lb = 0; // ინარჩუნებს მნიშვნელობას გამოძახებებს შორის
    ++lb;
    d = 8;
    b = 14;
    a += 1;
    return _x * _y + a + lb;
}
```

სიცოცხლის ხანგრძლივობა vs. ხილვადობის არე: სიცოცხლის ხანგრძლივობა განსაზღვრავს, არის თუ არა ობიექტი მეხსიერებაში, ხოლო ხილვადობის არე განსაზღვრავს, შეგვიძლია თუ არა მასზე მიმართვა სახელით კოდის მოცემულ წერტილში.

ავტომატურ ცვლადებს განეკუთვნება ასევე ცვლადები კვალიფიკატორით `register` (მაგ. `register int c;`). ეს არის მინიშნება კომპილატორისთვის, რომ ცვლადი განათავსოს პროცესორის რეგისტრში სწრაფი წვდომისთვის. თანამედროვე კომპილატორები ამ ოპტიმიზაციას ავტომატურად ახორციელებენ.

8.5 სტატიკური (`static`) ფუნქციები C ენაში

C ენაში `static` საკვანძო სიტყვის გამოყენება ცვლადებთან და ფუნქციებთან სრულიად განსხვავებულ ფუნქციურ დატვირთვას ატარებს.

როდესაც `static` საკვანძო სიტყვას ვუწერთ **ფუნქციას**, ეს არ ეხება მისი მეხსიერების სიცოცხლისუნარიანობას (როგორც ცვლადების შემთხვევაშია), არამედ განსაზღვრავს ფუნქციის **ხილვადობის არეს (Scope / Linkage)**.

1. კავშირის ტიპები: Global (External) vs Static (Internal) Linkage

ნაგულისხმევად (Default), C ენაში ყველა ფუნქციას აქვს **გარეგანი კავშირი (External Linkage)**. ეს ნიშნავს, რომ ფუნქციის გამოძახება შესაძლებელია პროექტის ნებისმიერი სხვა `.c` ფაილიდან.

ფუნქციისთვის `static` მოდიფიკატორის მიწერა მას ანიჭებს **შინაგანი კავშირის (Internal Linkage)** სტატუსს:

- ფუნქცია ხდება **„კერძო“ (Private)** მხოლოდ იმ ტრანსლირების ერთეულისთვის (`.c`

ფაილისთვის), სადაც ის არის აღწერილი.

- სხვა `.c` ფაილებიდან ამ ფუნქციის პირდაპირი გამოძახება შეუძლებელია (ლინკერი ამოაგდებს `Unresolved External Symbol` შეცდომას).

2. რატომ და როდის იყენებენ `static` ფუნქციებს?

ა) ინკაფსულაცია და ინფორმაციის დამალვა (Encapsulation)

ობიექტზე ორიენტირებული ენების (C++, Java) მსგავსად, სადაც გვაქვს `private` მეთოდები, C ენაში `static` ფუნქციები გამოიყენება დამხმარე (Helper) ლოგიკის დასამალად. API-ის მომხმარებელი ხედავს მხოლოდ საჯარო (Public) ფუნქციებს, ხოლო შიდა დამხმარე ფუნქციები დაცულია.

ბ) სახელების კონფლიქტის აცილება (Avoiding Name Collisions)

დიდ პროექტებში ან ბიბლიოთეკებში ხშირია ზოგადი სახელების მქონე დამხმარე ფუნქციები (მაგ. `validate()`, `swap()`, `parse_input()`). თუ ორ სხვადასხვა `.c` ფაილში გვაქვს ერთსა და იმავე სახელის მქონე გლობალური ფუნქცია, ლინკერი წარმოქმნის კონფლიქტს (`duplicate symbol`). `static`-ად გამოცხადებისას, ორივე ფაილში შეიძლება არსებობდეს იდენტური დასახელების ფუნქციები ერთმანეთთან კონფლიქტის გარეშე.

გ) კომპილატორის ოპტიმიზაცია (Inlining & Code Size)

რადგან კომპილატორმა ზუსტად იცის, რომ `static` ფუნქცია არასოდეს გამოიძახება ამ `.c` ფაილის გარეთ:

— მას შეუძლია ფუნქციის კოდი პირდაპირ ჩასვას გამოძახების ადგილას (Inlining), ფუნქციური გამოძახების (Function Call Overhead) აცილების მიზნით.

— თუ `static` ფუნქცია არსად არ გამოიყენება ფაილში, კომპილატორი მას საერთოდ ამოიღებს საბოლოო ბინარული ფაილიდან (Dead Code Elimination).

3. პრაქტიკული კოდის მაგალითი

განვიხილოთ მრავალფაილიანი პროექტის სტრუქტურა:

კოდი 8.7 დამხმარე მოდული

```
#include <stdio.h>

// static ფუნქცია: ხილვადია მხოლოდ ამ ფაილში (Private)
static double clamp_value(double val, double min, double max) {
 if (val < min) return min;
 if (val > max) return max;
 return val;
}

```

```
// ჩვეულებრივი ფუნქცია: ხილვადია ყველგან (Public API)
double normalize_signal(double raw_val) {
 // იყენებს შიდა static ფუნქციას
 return clamp_value(raw_val, 0.0, 1.0);
}
```

კოდი 8.8 მთავარი პროგრამა

```
#include <stdio.h>

// პროტოტიპების გამოცხადება
double normalize_signal(double raw_val);
// double clamp_value(double val, double min, double max); // ERROR!

int main(void) {
 double signal = 1.5;

 // სწორია: საჯარო ფუნქციის გამოძახება
 double clean_signal = normalize_signal(signal);
 printf("Normalized: %f\n", clean_signal); // დაბეჭდავს 1.000000

 // შეცდომა! ლინკერის შეცდომა (Linker Error):
 // clamp_value(signal, 0.0, 1.0);

 return 0;
}
```

ცხრილი 8.1 სტატიკური ცვლადი vs სტატიკური ფუნქცია

მახასიათებელი	**Static** ცვლადი (ფუნქციაში)	**Static** ფუნქცია
რა იცვლება?	მეხსიერების ხანგრძლივობა (Static Storage Duration).	ხილვადობის არე / კავშირი (Internal Linkage).
არსებობის ვადა	ინარჩუნებს მნიშვნელობას პროგრამის დასრულებამდე.	პროგრამის კომპილაციის/მიკავშირების(linking) ეტაპი.
დანიშნულება	მდგომარეობის (State) შენარჩუნება გამოძახებებს შორის.	ლოგიკის დამალვა, ინკაფსულაცია, სახელთა კონფლიქტის აცილება.

8.6 inline და static inline-ის განსხვავება C და C++ ენებში

პრაქტიკულად, C-ში `static inline` ფუნქცია ჰედერ ფაილში აკეთებს იმავეს, რასაც C++-ში უბრალო `inline` (საშუალებას იძლევა ფუნქცია განვსაზღვროთ `.h` ფაილში მრავალჯერადი ჩართვის შეცდომის გარეშე). თუმცა, ენების დონეზე ლინკერისა და მეხსიერების მართვის მექანიზმები განსხვავებულია.

1. `inline` C++ ენაში

C++-ში `inline` საკვანძო სიტყვა ანიჭებს ფუნქციას **გარეგან კავშირს (External Linkage)**, მაგრამ ავალდებულებს კომპილატორსა და ლინკერს, დაარღვიონ ODR (*One Definition Rule*) წესი:

- ჰედერ (`.h`) ფაილში ჩაწერილი `inline` ფუნქცია შეიძლება ჩართული (include) იყოს ათობით `.cpp` ფაილში.
- ლინკერი ხედავს ამ მრავალჯერად განსაზღვრებას, თუმცა შეცდომას (`multiple definition`) არ ამოაგდებს — ის **ყველა ასლს აერთიანებს ერთ გლობალურ მისამართზე**.
- *შედეგი:* C++-ში ამ ფუნქციის მისამართი (`&foo`) ყველა `.cpp` ფაილში ზუსტად **იდენტურია**.

2. `inline` C ენაში (C99 სტანდარტიდან)

C ენაში მხოლობითად ჩაწერილი `inline` ნიშნავს მხოლოდ „ინლაინ განსაზღვრებას“ (*Inline Definition*) და მას **არ მოჰყვება გლობალური განსაზღვრება (*External Definition*)**.

თუ C-ის ჰედერ ფაილში ჩაწერთ მხოლოდ `inline void foo() { ... }`:

1. **თუ კომპილატორი ჩასვამს კოდს ინლაინად:** პროგრამა უპრობლემოდ დაკომპილირდება.
2. **თუ კომპილატორმა კოდი ინლაინად არ ჩასვა:** (მაგ. Debug რეჟიმში ან თუ ფუნქცია დიდია), კომპილატორი დაიწყებს ამ ფუნქციის **გლობალური (External) სიმბოლოს ძებნას**.
3. **შედეგი (*Linker Error*):** თუ გლობალური სიმბოლო არცერთ `.c` ფაილში არ წერია, მიიღებთ შეცდომას: `Unresolved external symbol`. ხოლო თუ მას რამდენიმე `.c` ფაილში ჩაწერთ გლობალურად, მიიღებთ: `multiple definition`.

C-ში უბრალო `inline`-ის სწორი გამოყენება C-ში უბრალო `inline`-ის გამოსაყენებლად საჭიროა ჰედერში ინლაინ განსაზღვრება, ხოლო ერთ-ერთ `.c` ფაილში `extern`-ით გლობალური სიმბოლოს შექმნა:

კოდი 8.9 math_utils.h

```
1 // math_utils.h
2 inline int add(int a, int b) {
3  return a + b;
```

```
4 }
```

კოდი 8.10 math_utils.c

```
1 // math_utils.c
2 #include "math_utils.h"
3 extern int add(int a, int b); // ქმნის გლობალურ სიმბოლოს ლინკერისთვის!
```

ეს ნიშნავს შემდეგს:

ა) თუ კომპილატორმა გაუკეთა ფუნქციას "inlining"(მაგ. სამ ადგილას) მოხდება სამ სხვადასხვა ადგილას ფუნქციის ჩაშენება (ინლაინინგი როგორც სი++ში)

ბ) თუ კომპილატორმა გადაწყვიტა რომ არ ჩააშენოს, მაშინ ლინკერი დაიწყებს გლობალური ფუნქციის ძებნას და ეს იქნება `extern`-ით აღწერილი გლობალური ფუნქცია და მოხდება გლობალური, ჩაუშენებელი ფუნქციის გამოძახება ერთი მისამართიდან. ანუ შესაძლებელია კოდის ერთ ან რამდენიმე ნაწილში ჩააშენოს ფუნქცია და სადაც არ ჩააშენებს იქ მოხდება გლობალური ფუნქციის გამოძახება.

ესაა C99 სტანდარტის ოფიციალური გზა.

3. რატომ არის C-ში აუცილებელი `static inline`?

რადგან C-ში უბრალო `inline` ცალკე `extern` გამოცხადებას მოითხოვს, ჰედერ-ფაილებში იწერება **`static inline`** წყვილი:

- `static` — ანიჭებს ფუნქციას **შინაგანი კავშირის (Internal Linkage)** სტატუსს. ყოველი `.c` ფაილი იღებს თავის ლოკალურ ასლს, რის გამოც ლინკერი გლობალურ სიმბოლოს აღარ ეძებს და `multiple definition` შეცდომაც გამოირიცხება.
- `inline` — ეუბნება კომპილატორს, ეცადოს ამ ლოკალური ასლის პირდაპირ ჩასმას გამოძახების ადგილას.
- *ტექნიკური განსხვავება C++-თან:* თუ კომპილატორმა კოდი ინლაინად არ ჩასვა, C-ში სხვადასხვა `.c` ფაილში ამ ფუნქციის მისამართი (`&foo`) **განსხვავებული იქნება**, რადგან ისინი დამოუკიდებელი ლოკალური ასლებია.

საკვანძო სიტყვა `static` ინლაინინგს **არ კრძალავს** — `static inline`-ის დროსაც **ინლაინინგი ჩვეულებრივად ხდება**. `static` განსაზღვრავს მხოლოდ „სათადარიგო გეგმას“ (*Fallback Plan*) იმ შემთხვევისთვის, თუ კომპილატორი ინლაინინგს ვერ/არ განახორციელებს.

სცენარი ა: კომპილატორმა გააკეთა ინლაინინგი (სტანდარტული შემთხვევა)

- ფუნქციის ბრძანებები ჩაჯდა პირდაპირ გამოძახების ადგილას.
- მეხსიერებაში არანაირი ლოკალური ასლი არ იქმნება და არც მისამართი არსებობს.

სცენარი ბ: კომპილატორმა ვერ/არ გააკეთა ინლაინინგი (ან აიღეთ მისამართი `&func`)

- ამოქმედდება `static` მექანიზმი.

- კომპილატორი ამ კონკრეტული `.c` ფაილისთვის ქმნის ფუნქციის ლოკალურ ასლს 1 მისამართზე და ყველა არა-ინლაინ გამოძახება მიმართავს ამ ასლს.

შედარებითი ანალიზი: უბრალო `inline` vs `static inline`

ორივე შემთხვევაში პირველადი მიზანი ინლაინინგია. განსხვავება ვლინდება **მხოლოდ სათადარიგო გეგმაში** (როცა ინლაინინგი არ განხორციელდა ან ფუნქციის მისამართი აიღეთ):

ცხრილი 8.2 `inline` და `static inline` ყოფაქცევა

მოდიფიკატორი	თუ მოხდა ინლაინინგი	თუ არ მოხდა ინლაინინგი (სათადარიგო გეგმა)
`inline`	კოდი ჩააშენა გამოძახების ადგილას.	ეძებს **1 გლობალურ ფუნქციას** (`.c` ფაილში `extern`-ით შექმნილს).
`static inline`	კოდი ჩააშენა გამოძახების ადგილას.	ქმნის **ლოკალურ ასლს** თითოეული `.c` ფაილისთვის თავის მისამართზე.

შეჯამება:

ბრძანებების ჩასმა: ორივე შემთხვევაში (`inline` და `static inline`) ინლაინინგი ნიშნავს ფუნქციის ინსტრუქციების ჩასმას გამოძახების ადგილას.
`static`-ის როლი: `static` არის დაზღვევა. თუ კომპილატორმა ვერ ჩააშენა ფუნქცია, იმ `.c` ფაილში შეიქმნება ლოკალური ასლი, რათა ლინკერის შეცდომა არ წარმოიქმნას.
C++-ში: `inline` საკმარისია ჰედერში დასაწერად (ლინკერი თავად აერთიანებს სიმბოლოებს).
C-ში: უბრალო `inline` მოითხოვს ცალკე `extern`-ით გლობალური ვერსიის გამოცხადებას.
დასკვნა: C ენაში დეველოპერები ჰედერ ფაილებში ყოველთვის იყენებენ **`static inline`**-ს, რადგან ის უზრუნველყოფს ლოკალურ ასლებს და თავიდან აცილებს ლინკერის შეცდომებს.

4. `static inline` C++-ში

C++-ში inline ფუნქციის წინ static-ის დაწერა ფუნქციონალურად პრაქტიკულად არაფერს ცვლის და თანამედროვე C++-ში ამის გაკეთება არარეკომენდებულიცაა (Bad Practice).

C++-ში inline საკვანძო სიტყვას აქვს ძლიერი, მკაფიოდ განსაზღვრული მნიშვნელობა (Linkage-ის დონეზე):

ODR (One Definition Rule) გამონაკლისი: ის ენის დონეზე რთავს ნებას, რომ ფუნქციის განმარტება (definition) იყოს მრავალ სხვადასხვა .cpp ფაილში (header-ის საშუალებით), და ლინკერი (Linker) არ განაცხადებს შეცდომას (multiple definition error).

External Linkage (გარე კავშირი): ლინკერი იცნობს ამ ფუნქციას, იღებს ყველა .cpp-დან შემოსულ ერთნაირ ვერსიას და მათ ერთიანად აერთიანებს.

თუ C++-ში დაწერთ static inline-ს, static საკვანძო სიტყვა ფუნქციას ანიჭებს `Internal Linkage` (შიდა კავშირს):

იზოლაცია: ფუნქცია ხდება ხილვადი მხოლოდ იმ კონკრეტული .cpp ფაილისთვის, რომელშიც ის `#include`-ით შევიდა.

დუბლირება მეხსიერებაში: თუ ეს `Header` ფაილი 10 სხვადასხვა .cpp ფაილშია შეტანილი (`included`), კომპილატორი შექმნის ამ ფუნქციის 10 სრულიად დამოუკიდებელ ასლს მეხსიერებაში (თითოეულს თავისი უნიკალური მისამართი ექნება).

ლინკერი ვერ გააერთიანებს: უბრალოდ `inline`-ისგან განსხვავებით, ლინკერი ამ 10 ასლს ერთმანეთთან ვერ გააერთიანებს, რადგან `static`-მა მათ შიდა (`private`) სტატუსი მიანიჭა.

C++-ში Header ფაილებში ვწერთ `inline`. ეს სრულიად საკმარისია, უსაფრთხოა და ეხმარება ლინკერს კოდის დუბლირების თავიდან აცილებაში.

C++-ში `static inline`-ის დაწერა ჩვევად მხოლოდ C ენიდან გადმოდის, სადაც C-ის სპეციფიკური მიკავშირების წესების გამო `static inline` აუცილებლობა იყო. C++-ში კი `static`-ი უბრალოდ ართმევს ფუნქციას გლობალურ გაერთიანების უნარს.

ცხრილი 8.3 `inline` და `static inline` C/C++

ფუნქციის ტიპი	C ენაში	C++ ენაში
`inline`	არ არის თვითკმარი (ჭირდება extern ან იმავე ფაილში გამოყენება, თორემ ლინკერის შეცდომას იძლევა).	სტანდარტული გზა Header-ებში! უზრუნველყოფს ODR-ს და ლინკერი აერთიანებს დუბლიკატებს.
static inline	სტანდარტული გზა C Header-ებში! იცავს Linker error-ისგან და საშუალებას აძლევს ოპტიმიზაციას.	ზედმეტია / არარეკომენდებულია. ფუნქციას ხდის "ლოკალურს" თითოეული .cpp-სთვის და შეიძლება გაზარდოს ბინარული ფაილის ზომა.

კროს-კომპილატორული კოდი

შეიძლება ერთი და იგივე კოდი გამოყენებულ იქნას როგორც C ასევე C++ კომმპილატორისთვის(პროექტისთვის), inline-მ რომ სწორად იმუშაოს შეგვიძლია გავაკეთოთ შემდეგი

```
#ifndef DYN_INLINE
#ifdef __cplusplus
#define DYN_INLINE inline
#else
#define DYN_INLINE static inline
#endif
#endif
```

1. **C კომპილატორისთვის (static inline):**
   C ენაში უბრალო inline Header ფაილში იწვევს undefined reference ლინკერის შეცდომას (თუ სადმე extern არ დაუწერე). static inline უზრუნველყოფს, რომ C-ში კოდი უპრობლემოდ დაკომპილირდეს ყოველ .c ფაილში.
2. **C++ კომპილატორისთვის (inline):**
   C++-ში უბრალო inline იდეალურად მუშაობს Header-ებში. ის იცავს ODR (One Definition Rule) წესს და ლინკერს აძლევს უფლებას, სხვადასხვა .cpp ფაილიდან შემოსული ფუნქციის დუბლიკატები გააერთიანოს (Avoid Code Bloat).

გამოყენების მაგალითი Header ფაილში (my_header.h), როდესაც ფუნქციას განვსაზღვრავთ ორივე ენისთვის თავსებად Header-ში:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

#ifdef __cplusplus
extern "C" {
#endif

 // მაკროს განმარტება
 #ifndef DYN_INLINE
  #ifdef __cplusplus
  #define DYN_INLINE inline
  #else
  #define DYN_INLINE static inline
  #endif
 #endif

 // უნივერსალური ფუნქცია C და C++ -ისთვის
```

```
18  DYN_INLINE float cube(const float x)
19  {
20   return x*x*x;
21  }
22
23  #ifdef __cplusplus
24 }
25 #endif
26
27 #endif // MY_HEADER_H
```

8.7 ცვლადის კვალიფიკატორი volatile

კვალიფიკატორი volatile აცნობებს კომპილატორს, რომ ცვლადის მნიშვნელობა შეიძლება შეიცვალოს პროგრამის გარეთ ნებისმიერ მომენტში (მაგალითად, აპარატურული ინტერრაპტით, ტაიმერით ან სხვა ნაკადით). კომპილატორს ეკრძალება ასეთი ცვლადის ოპტიმიზაცია (ქეშირება პროცესორის რეგისტრში).

```
volatile int hw_flag = 0; // შეიძლება შეიცვალოს აპარატურულად
```

8.8 ტიპის გარდაქმნა — კასტირება (Type Casting)

ხშირად საჭიროა ერთი ტიპის ცვლადის გადაყვანა სხვა ტიპში:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 5;
6      int b = 10;
7      float f;
8
9      f = (float)a / b;  // explicit type cast float-ში
10     printf("%.2f\n", f); // დაბეჭდავს 0.50
11     return 0;
12 }
```

8.9 void* მიმთითებელი (Generic Pointer)

void* არის უტიპო მიმთითებელი, რომელსაც შეუძლია მიუთითოს ნებისმიერი ტიპის მონაცემზე მეხსიერებაში. მასზე პირდაპირ დერეფერენცირება (*p) ან არითმეტიკული ოპერაციები დაუშვებელია ტიპის გარდაქმნის (*casting*) გარეშე.

კოდი 8.11 void* მიმთითებლის გამოყენების მაგალითი

```c
#include <stdio.h>
void *pointer_array[10]; // 10-ელემენტიანი void* მასივი
int idx = 0;
void append_pointer(void *p)
{
    pointer_array[idx++] = p;
}

void *get_pointer(int i)
{
    return pointer_array[i];
}

int main(void)
{
    char *s = "some data!";
    int a = 10;
    int *b = &a;
    append_pointer(s);
    append_pointer(b);
    char *s2 = (char*)get_pointer(0);
    int *b2 = (int*)get_pointer(1);
    printf("b2 value: %d\n", *b2);
    printf("s2 string: %s\n", s2);
    return 0;
}
```

8.10 მეხსიერების დინამიური მართვა

მეხსიერების დინამიური გამოყოფა (Heap-ზე) პროგრამას აძლევს საშუალებას მოითხოვოს და გაათავისუფლოს მეხსიერება უშუალოდ შესრულების დროს (*Runtime*).

C ენის სტანდარტული ბიბლიოთეკა (`<stdlib.h>`) გვაწვდის 4 ძირითად ფუნქციას:

`malloc(nsize)` — გამოყოფს `nsize` ბაიტ მეხსიერებას (არაინიციალიზებულს).

`calloc(num, nsize)` — გამოყოფს მეხსიერებას და ავსებს ნულებით (0).

`realloc(ptr, new_size)` — ცვლის არსებული მეხსიერების ბლოკის ზომას.

`free(ptr)` — ათავისუფლებს გამოყოფილ მეხსიერებას.

8.10.1 realloc() ფუნქციის პრაქტიკული მაგალითი

```c
#include <stdio.h>
#include <stdlib.h>
#define INITIAL_SIZE 5
#define BUCKET_SIZE 5
int main(void)
{
    int count = 0;
    int capacity = INITIAL_SIZE;
    int *data = malloc(capacity * sizeof(int));
    if (data == NULL)
    {
        fprintf(stderr, "Memory allocation failed!\n");
        return EXIT_FAILURE;
    }
    // 23 ელემენტის დამატება
    for (int i = 0; i < 23; i++)
    {
        if (count == capacity)
        {
            capacity += BUCKET_SIZE;
            int *temp = realloc(data, capacity * sizeof(int));
            if (temp == NULL)
            {
                fprintf(stderr, "Reallocation failed!\n");
                free(data);
                return EXIT_FAILURE;
            }
            data = temp;
        }
        data[count++] = i;
    }
    printf("Successfully added %d elements. Capacity: %d\n", count,
    ↪ capacity);
    free(data); // მეხსიერების გათავისუფლება
    return 0;
}
```

8.11 დინამიური 2D და 3D მასივები (Flat Array Approach)

მრავალგანზომილებიანი მასივების შესაქმნელად Heap-ზე ყველაზე ეფექტურია 1D ბრტყელი მასივის (*Flat Array*) გამოყენება.

ცხრილი 8.4 კლასიკური vs. 1D ბრტყელი დინამიური მასივი

პარამეტრი	კლასიკური (`int**`)	1D ბრტყელი (`int*`)
მეხსიერება	ფრაგმენტირებული.	აბსოლუტურად უწყვეტი (Contiguous).
ალოკაცია / წაშლა	სჭირდება ციკლები (`for`).	მხოლოდ 1 `malloc` და 1 `free`.
სისწრაფე (Cache)	უფრო ნელი (Cache miss-ები).	ძალიან სწრაფი (Cache-friendly).
ინდექსაცია	`arr[i][j]`	`arr[i * cols + j]`

8.12 3D მასივის სტრუქტურა და ინკაფსულაცია

მონაცემთა მიმთითებლისა და განზომილებების გაერთიანება სტრუქტურაში უზრუნველყოფს კოდის უსაფრთხოებასა და სისუფთავეს.

კოდი 8.13 3D დინამიური მასივის სტრუქტურული რეალიზაცია

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    float *data;
    size_t x_dim;
    size_t y_dim;
    size_t z_dim;
} DynArray3D;
DynArray3D* create_dyn_array3d(size_t x, size_t y, size_t z)
{
    DynArray3D *arr = malloc(sizeof(DynArray3D));
    if (!arr) return NULL;
    arr->x_dim = x;
    arr->y_dim = y;
    arr->z_dim = z;
    arr->data = malloc(x * y * z * sizeof(float));

```

```
18      if (!arr->data) {
19          free(arr);
20          return NULL;
21      }
22      return arr;
23  }
24  static inline float get_elem3d(const DynArray3D *arr, size_t x, size_t
        ↪  y, size_t z)
25  {
26      return arr->data[z * (arr->x_dim * arr->y_dim) + y * arr->x_dim +
        ↪ x];
27  }
28  void free_dyn_array3d(DynArray3D *arr)
29  {
30      if (arr) {
31          free(arr->data);
32          free(arr);
33      }
34  }
```

8.13 goto ოპერატორი და ნიშნულები (Labels)

goto არის უპირობო გადასვლის ოპერატორი. ის პროგრამის ნაკადს გადაისვრის მითითებულ ნიშნულზე (label:) იმავე ფუნქციაში.

კოდი 8.14 ჩაშენებული ციკლებიდან გამოსვლა goto-თი

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int matrix[3][3] = {
5          {1, 2, 3},
6          {4, -99, 6},
7          {7, 8, 9}
8      };
9      for (int i = 0; i < 3; ++i) {
10         for (int j = 0; j < 3; ++j) {
11             if (matrix[i][j] < 0) {
12                 printf("Found negative value %d at [%d][%d]\n", matrix
       ↪ [i][j], i, j);
13                 goto found; // გამოდის ორივე ციკლიდან
14             }
```

```
        }
    }
found:
    printf("Search finished.\n");
    return 0;
}
```

კოდი 8.15 რესურსების გასუფთავება (Cleanup Pattern) C ენაში

```c
#include <stdio.h>
#include <stdlib.h>
int process_data(const char *filename)
{
    FILE *file = fopen(filename, "r");
    if (!file) return -1;
    int *buf1 = malloc(100 * sizeof(int));
    if (!buf1) goto cleanup_file;
    int *buf2 = malloc(100 * sizeof(int));
    if (!buf2) goto cleanup_buf1;

    // ძირითადი სამუშაო ლოგიკა...
    free(buf2);
    free(buf1);
    fclose(file);
    return 0;

cleanup_buf1:
    free(buf1);
cleanup_file:
    fclose(file);
    return -1;
}
```

დავალება 8.1: დინამიური 2D მასივის სტრუქტურა

შექმენით `DynArray2D` სტრუქტურა `float* data`, `size_t rows`, `size_t cols` ველებით. დაწერეთ შექმნის, გათავისუფლების, ელემენტზე წვდომისა და ფაილში ჩაწერა/წაკითხვის ფუნქციები.

დავალება 8.2: მეხსიერების გაჟონვის პოვნა

გააანალიზეთ მოცემული ფუნქციები და შეასწორეთ მეხსიერების გაჟონვის (*Memory Leak*) შეცდომები `free()`-ის სწორი გამოყენებით.

დავალება 8.3: სტატიკური ცვლადების ტრეკერი

დაწერეთ ფუნქცია `int get_next_id(void)`, რომელიც ყოველ გამოძახებაზე დააბრუნებს უნიკალურ, ზრდად ID-ს $(1, 2, 3, \dots)$ `static` ცვლადის გამოყენებით.

თავი 9

ბიტური ოპერაციები და ბიტური ველები

9.1 ბიტური ლოგიკური ოპერაციები (AND, OR, XOR, NOT) 186
9.2 მარცხნივ და მარჯვნივ წანაცვლების ოპერაციები 187
9.3 ბიტური ოპერაციების პრიორიტეტი (Precedence) 188
9.4 ბიტური ოპერაციების მექანიზმი ორობით (0 და 1) ენაზე 189
9.4.1 ბიტური ნიღაბი 189
9.5 ბიტური ნიღბები (Bit Masks): 8, 16 და 32-ბიტიანი სისტემები . . . 191
9.5.1 8, 16 და 32-ბიტიანი ნიღბების განსაზღვრა 191
9.5.2 ბიტური ოპერაციების მაგალითი 192
9.5.3 უნივერსალური ბიტური მაკროსები (8, 16, 32, 64 ბიტისთვის) . . . 193
9.6 პრაქტიკული გამოყენება: აპარატურული რეგისტრებისა და LED-ების მართვა 194
9.6.1 ლაბორატორიული მაგალითები: LED-ების მართვის სცენარები . . 194
9.7 ბიტური ველები (Bit Fields) 197
9.7.1 8, 16 და 32-ბიტიანი სტრუქტურის მაგალითები 197
9.7.2 უსახელო ველები (Unnamed Bit Fields) და მეხსიერების გასწორება 198
9.7.3 K&R კრიტიკული გაფრთხილება: `signed` vs `unsigned` ბიტური ველები 199
9.7.4 Unions + Bit Fields: აპარატურული რეგისტრების იდეალური მოდელი 200
9.7.5 ბიტური ველების შეზღუდვები და არაპორტაბელურობა 202

C ენაში გვაქვს ბიტებზე მოქმედების ექვსი ძირითადი ოპერატორი: ბიტური „და“ (AND) `&` ↪ , ბიტური „ან“ (OR) `|`, ექსკლუზიური „ან“ (XOR) `^`, ბიტური უარყოფა „არა“ (NOT) `~`, მარცხნივ წანაცვლება (left-shift) `<<` და მარჯვნივ წანაცვლება (right-shift) `>>`.

ეს ოპერაციები გამოიყენება მხოლოდ მთელრიცხოვან ტიპებზე: `char`, `short`, `int`, `long` ↪ , აგრეთვე C99 სტანდარტის ზუსტი სიგანის ტიპებზე (`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`). არ შეიძლება ამ ოპერაციების გამოყენება მცოცავწერტილიან ტიპებზე (`float`, `double`).

ოპერატორებს `&`, `|`, `^`, `<<` და `>>` აქვთ შესაბამისი მინიჭების ოპერატორები: `&=`, `=|`, `^=` ↪ , `<<=` და `>>=`. ეს ოპერატორები მოქმედებენ მათემატიკური მინიჭების ოპერატორების

ანალოგიურად. მაგალითად:

```
z &= x | y;
```

იგივეა, რაც:

```
z = z & (x | y);
```

მნიშვნელოვანია გვახსოვდეს, რომ ბიტური ოპერატორები (&, |, ~) პრინციპულად განსხვავდებიან ლოგიკური ოპერატორებისგან (&&, ||, !). ლოგიკური ოპერატორები აფასებენ მთლიან გამოსახულებას ჭეშმარიტებად (1) ან მცდარობად (0), ხოლო ბიტური ოპერატორები პარალელურად მოქმედებენ ცვლადის თითოეულ ბიტზე.

9.1 ბიტური ლოგიკური ოპერაციები (AND, OR, XOR, NOT)

ჩაშენებული სისტემების პროგრამირებაში მიკროკონტროლერის პინებისა და რეგისტრების მართვა ეფუძნება 4 ფუნდამენტურ ოპერაციას: AND, OR, XOR, NOT.

ოპერატორები &, |, ^ და ~ საშუალებას იძლევა ბიტებზე ჩატარდეს ლოგიკური ოპერაციები. უარყოფის ოპერატორი ~ უნარულია — ის 0-ს ცვლის 1-ით და პირიქით. დანარჩენი ოპერატორები ბინარულია. ისინი ადარებენ ორ ბიტს და იძლევიან შედეგს შემდეგი წესის მიხედვით:

ცხრილი 9.1 ბინარული ბიტური ოპერაციები

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

მნიშვნელოვანი გაფრთხილება (ბიტების ნუმერაცია): პროგრამირებასა და ციფრულ ელექტრონიკაში ბიტების ინდექსაცია (ნუმერაცია) **ყოველთვის იწყება 0-იდან**:

- **Bit 0 (ბიტი 0):** $2^0 = 1$ (0b0000 0001)
- **Bit 1 (ბიტი 1):** $2^1 = 2$ (0b0000 0010)
- **Bit 2 (ბიტი 2):** $2^2 = 4$ (0b0000 0100)
- **Bit 3 (ბიტი 3):** $2^3 = 8$ (0b0000 1000)

ამრიგად, როდესაც ვამბობთ **„მე-3 ბიტი“ (Bit 3)**, ყოველთვის იგულისხმება ბიტი **ინდექსით 3** (ანუ `1U << 3` $\rightarrow 1000_2$ / 0x08), და არა მარჯვნიდან მესამე პოზიციაზე მდგომი ბიტი!

განვიხილოთ 8-ბიტიანი (1 ბაიტი) მაგალითი:

```
1 unsigned char x = 55; /* 55 (dec) = 0x37 (hex) = 0011 0111 (binary) */
2 unsigned char y = 25; /* 25 (dec) = 0x19 (hex) = 0001 1001 (binary) */
3 unsigned char z;
```

1. განვიხილოთ ბიტური „და“ — `z = x & y`:

	128	64	32	16	8	4	2	1	
	7	6	5	4	3	2	1	0	
	0	0	1	1	0	1	1	1	$_2 = \mathbf{55}_{10} = \texttt{0x37}_{16}$
&	0	0	0	1	1	0	0	1	$_2 = \mathbf{25}_{10} = \texttt{0x19}_{16}$
	0	0	0	1	0	0	0	1	$_2 = \mathbf{17}_{10} = \texttt{0x11}_{16}$

   ბიტური „და“ (`&`) შედეგის ბიტს აქცევს 1-ად მხოლოდ მაშინ, თუ `x` და `y`-ის შესაბამისი ორივე ბიტი 1-ის ტოლია. ეს ოპერატორი ხშირად გამოიყენება კონკრეტული ბიტების განულებისა და შერჩეული ბიტების შემოწმებისთვის (Masking).

2. ბიტური „ან“ ოპერაციის `z = x | y` შედეგად `z` იქნება 63:

	128	64	32	16	8	4	2	1	
	7	6	5	4	3	2	1	0	
	0	0	1	1	0	1	1	1	$_2 = \mathbf{55}_{10} = \texttt{0x37}_{16}$
|	0	0	0	1	1	0	0	1	$_2 = \mathbf{25}_{10} = \texttt{0x19}_{16}$
	0	0	1	1	1	1	1	1	$_2 = \mathbf{63}_{10} = \texttt{0x3F}_{16}$

   ბიტური „ან“ (`|`) შედეგის ბიტს აქცევს 1-ად, თუ `x` ან `y`-ის შესაბამისი ბიტებიდან ერთი მაინც არის 1. ეს ოპერატორი ხშირად გამოიყენება სასურველი ბიტების ჩასართავად (1-ად გასახდომად).

3. მესამე ოპერაციის — ექსკლუზიური „ან“ (`z = x ^ y`) შედეგად `z` იქნება 46:

	128	64	32	16	8	4	2	1	
	7	6	5	4	3	2	1	0	
	0	0	1	1	0	1	1	1	$_2 = \mathbf{55}_{10} = \texttt{0x37}_{16}$
^	0	0	0	1	1	0	0	1	$_2 = \mathbf{25}_{10} = \texttt{0x19}_{16}$
	0	0	1	0	1	1	1	0	$_2 = \mathbf{46}_{10} = \texttt{0x2E}_{16}$

   ბიტური XOR (`^`) შედეგის ბიტს აქცევს 1-ად მაშინ, როცა შესატყვისი ბიტებიდან განსხვავებული მნიშვნელობისაა (მხოლოდ ერთიანია რომელიმე ერთი). იგი გამოიყენება ბიტების მდგომარეობის ინვერტირებისთვის (Toggle).

4. მეოთხე ოპერაცია `~` (ბიტური ინვერსია / NOT) 0-ს ცვლის 1-ით და 1-ს 0-ით:

	128	64	32	16	8	4	2	1	
	7	6	5	4	3	2	1	0	
~	0	0	1	1	0	1	1	1	$_2 = \mathbf{55}_{10} = \texttt{0x37}_{16}$
	1	1	0	0	1	0	0	0	$_2 = \mathbf{200}_{10} = \texttt{0xC8}_{16}$

9.2 მარცხნივ და მარჯვნივ წანაცვლების ოპერაციები

წანაცვლების ოპერატორები `<<` და `>>` ანაცვლებენ მთელი ცვლადის ბიტებს მარცხნივ ან მარჯვნივ.

მარცხნივ წანაცვლება (`x << n`) ანაცვლებს `x` ცვლადის ბიტებს `n` პოზიციით მარცხნივ.

მარცხნიდან თანრიგს გარეთ გასული უფროსი ბიტები იკარგება, ხოლო მარჯვნიდან შემოსული უმცროსი ბიტები ყოველთვის 0-ებია.

```
1 unsigned char x = 0x19; /* 0001 1001 binary (25 decimal) */
2 unsigned char y = x << 2; /* 0110 0100 binary (100 decimal) */
```

გაფრთხილება: უარყოფითი სიდიდით წანაცვლება ან წანაცვლება რაოდენობით, რომელიც მეტია ან ტოლი ცვლადის ბიტების სიგანეზე, იწვევს განუსაზღვრელ ქცევას (Undefined Behavior) და დაუშვებელია!

მარჯვნივ წანაცვლებისას (`x >> n`):

- თუ `x` *უნიშნო* ტიპისაა (`unsigned`), მარცხნიდან ყოველთვის შემოდის 0-ები (*ლოგიკური წანაცვლება*).
- თუ `x` *ნიშნიანი* ტიპისაა (`signed`), ქცევა დამოკიდებულია არქიტექტურასა და კომპილატორზე. ზოგ სისტემაში შემოდის 0-ები, ხშირად კი ინარჩუნებს ნიშნის ბიტს (*არითმეტიკული წანაცვლება*).

მაგალითი ნიშნიანი ცვლადისთვის:

```
1 signed char x = -75;   /* 1011 0101 binary */
2 signed char y = x >> 2;/* 0010 1101 (45 dec) - ლოგიკური წანაცვლებისას */
3                        /* 1110 1101 (-19 dec) - არითმეტიკული წანაცვლებისას */
```

არაპორტაბელური ქცევის თავიდან ასაცილებლად, ბიტური წანაცვლების ოპერაციები უმჯობესია **ყოველთვის უნიშნო (`unsigned`) ტიპებზე** შესრულდეს.

უნიშნო (`unsigned`) ტიპის რიცხვების წანაცვლება ექვივალენტურია ორის ხარისხზე გამრავლება/გაყოფისა:

```
x * 2    იგივეა, რაც   x << 1
x / 16   იგივეა, რაც   x >> 4
x % 8    იგივეა, რაც   x & 7
```

თანამედროვე ოპტიმიზატორ-კომპილატორები მათემატიკურ გამრავლებას/გაყოფას თავად გარდაქმნიან ბიტურ ოპერაციებად, ამიტომ კოდის წაკითხვადობის დასათმობად ამგვარი ოპტიმიზაცია ხელით ყოველთვის საჭირო არ არის.

9.3 ბიტური ოპერაციების პრიორიტეტი (Precedence)

ბიტური ოპერატორების პრიორიტეტი („წონა“) უფრო დაბალია, ვიდრე არითმეტიკული ოპერატორებისა (გარდა უნარული `~`-ისა).

წანაცვლების ოპერატორებს (`<<`, `>>`) უფრო მაღალი პრიორიტეტი აქვთ, ვიდრე შედარების ოპერატორებს (`<`, `>`), მაგრამ ბიტურ ლოგიკურ ოპერატორებს (`&`, `^`, `|`) შედარების

ოპერატორებზე დაბალი პრიორიტეტი აქვთ! პრიორიტეტების იერარქია:

`~` > `<<, >>` > `&` > `^` > `|`

ვინაიდან ბიტური ოპერატორების პრიორიტეტი ხშირად ინტუიციას არ ემთხვევა, **პროგრამირების საუკეთესო პრაქტიკაა ყოველთვის ფრჩხილების გამოყენება**:

```
if ((flags & MASK) == 0) /* ფრჩხილები (flags & MASK) აუცილებელია! */
```

9.4 ბიტური ოპერაციების მექანიზმი ორობით (0 და 1) ენაზე

იმისათვის, რომ უკეთ გავიაზროთ, თუ რატომ მუშაობს ბიტური მაკროსები და რეგისტრების მართვა, განვიხილოთ თითოეული ოპერაცია უშუალოდ ორობით (0-ებისა და 1-ების) დონეზე.

9.4.1 ბიტური ნიღაბი

ბიტური ნიღბის შექმნა (`1U << n`): ეს არის ყველაზე გავრცელებული ხერხი ბიტური ნიღბის (Bit Mask) შესაქმნელად. ჩანაწერი `1U << n` ქმნის რიცხვს, სადაც **მხოლოდ ინდექსით n ბიტია** 1, ხოლო ყველა დანარჩენი — 0.

საწყისი (`1U`): `0 0 0 0  0 0 0 1` $(1 \cdot 2^0 = 1)$

`1U << 1:` `0 0 0 0  0 0 1 0` (1-ლი ბიტი გახდა $1 \to 1 \cdot 2^1 = 2$)

`1U << 3:` `0 0 0 0  1 0 0 0` (მე-3 ბიტი გახდა $1 \to 1 \cdot 2^3 = 8$)

`1U << 7:` `1 0 0 0  0 0 0 0` (მე-7 ბიტი გახდა $1 \to 1 \cdot 2^7 = 128$)

შენიშვნა: `U` სუფიქსი (`1U`) ნიშნავს *Unsigned*-ს (უნიშნო მთელ რიცხვს). დაბალი დონის პროგრამირებაში 1-იანის უნიშნოდ გამოცხადება აუცილებელია, რათა თავიდან ავიცილოთ ნიშნიანი ბიტის (Sign bit) გადაფარვა და გაურკვეველი ქცევა (Undefined Behavior) 32/64-ბიტიან სისტემებში.

1. ყველა ბიტის გასუფთავება / განულება (`status &= ~0xFFFFFFFFUL`)

თექვსმეტობითი კონსტანტა `0xFFFFFFFFUL` წარმოადგენს 32-ბიტიან რიცხვს, სადაც ყველა ბიტი 1-ის ტოლია:

$$\texttt{0xFFFFFFFFUL} = 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111_2$$

როდესაც ვიყენებთ ინვერსიის (NOT) `~` ოპერატორს, ყველა 1 ხდება 0:

$$\texttt{\~{}0xFFFFFFFFUL} = 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_2$$

ახლა განვიხილოთ ბიტური „და“ (`&`) ოპერაცია ნებისმიერი საწყისი `status` მნიშვნელობისთვის (x აღნიშნავს ნებისმიერ ბიტს — 0-ს ან 1-ს):

```
status:          x x x x  x x x x  x x x x  x x x x   (საწყისი მდგომარეობა)
~0xFFFFFFFFUL: & 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0   (ყველა ბიტი ნულია)
-----------------------------------------------------
შედეგი:          0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0   (ყველა ბიტი განულდა)
```

რადგან $x \,\&\, 0 = 0$ (ნებისმიერი ბიტის „და“ ოპერაცია 0-თან იძლევა 0-ს), რეგისტრის ყველა ბიტი გარანტირებულად ნულდება.

შენიშვნა: `status &= ~0xFFFFFFFFUL` ექვივალენტურია `status = 0`-ის, თუმცა დაბალი დონის პროგრამირებაში ასეთი ჩანაწერი ხაზს უსვამს იმას, რომ ჩვენ მიზანმიმართულად ვასუფთავებთ 32-ბიტიანი რეგისტრის ყველა ბიტურ დროშას.

2. კონკრეტული ბიტის ჩართვა (Set - `|=`)

ბიტის 1-ად გასახდომად ვიყენებთ ბიტურ „ან“ (`|`) ოპერაციას 1-თან, რადგან:

$$x \mid 1 = 1 \quad \text{(ნებისმიერი ბიტი „ან“ 1 ყოველთვის იძლევა 1-ს)}$$

$$x \mid 0 = x \quad \text{(ნებისმიერი ბიტი „ან“ 0 ინარჩუნებს თავის მნიშვნელობას)}$$

დავუშვათ, გვინდა მე-2 ბიტის ჩართვა (`1U << 2`):

```
საწყისი status:    0 1 0 0  1 0 0 0   (მაგალითად)
ნიღაბი (1U << 2): | 0 0 0 0  0 1 0 0   (მხოლოდ მე-2 ბიტია 1)
-----------------------------------------
ახალი status:      0 1 0 0  1 1 0 0   (მე-2 ბიტი გახდა 1, სხვა არ შეიცვალა)
```

3. კონკრეტული ბიტის გამორთვა (Clear - `&= ~`)

ბიტის 0-ად გასახდომად (ისე, რომ სხვა ბიტები არ დავაზიანოთ) ვიყენებთ ბიტურ „და“ (`&`) ოპერაციას ინვერტირებულ ნიღაბთან.

თუ გვინდა მე-3 ბიტის გამორთვა:

1. ნიღაბი `(1U << 3)` არის: `0000 1000`
2. ინვერტირებული ნიღაბი `~(1U << 3)` არის: `1111 0111`

რადგან $x \,\&\, 1 = x$ და $x \,\&\, 0 = 0$, ვიღებთ:

```
საწყისი status:    0 1 0 0  1 1 0 0   (მე-3 ბიტი ანთებულია)
ნიღაბი ~(1U << 3): & 1 1 1 1  0 1 1 1   (მხოლოდ მე-3 ბიტია 0)
-----------------------------------------
ახალი status:      0 1 0 0  0 1 0 0   (მე-3 ბიტი ჩაქრა, სხვა არ შეიცვალა)
```

4. ბიტის მდგომარეობის ინვერტირება / გადართვა (Toggle - ^=)

ბიტის საპირისპირო მდგომარეობაში გადასაყვანად ვიყენებთ ექსკლუზიურ „ან“ (^) ოპერაციას:

$$x \hat{} 1 = \neg x \quad (0 \hat{} 1 = 1, \quad 1 \hat{} 1 = 0)$$

$$x \hat{} 0 = x \quad (\text{ინარჩუნებს უცვლელად})$$

საწყისი status:	0 1 0 0 1 1 0 0	(მე-2 ბიტი იყო 1)
ნიღაბი (1U << 2):	^ 0 0 0 0 0 1 0 0	(მე-2 ბიტის ტოგლი)
ახალი status:	0 1 0 0 1 0 0 0	(მე-2 ბიტი გახდა 0)

9.5 ბიტური ნიღბები (Bit Masks): 8, 16 და 32-ბიტიანი სისტემები

ბიტური ოპერაციების მთავარი დანიშნულებაა:

1. მეხსიერების დაზოგვა და მდგომარეობების (Flags/Status) შენახვა.
2. აპარატურულ რეგისტრებთან (Hardware Registers, GPIO, Timers, UART) ურთიერთქმედება ჩაშენებულ სისტემებში (Embedded Systems).

ბიტური ნიღაბი (Bit Mask) არის წინასწარ განსაზღვრული რიცხვითი მუდმივა, სადაც მხოლოდ კონკრეტული ბიტი (ან ბიტების ჯგუფია) 1-ის ტოლი.

9.5.1 8, 16 და 32-ბიტიანი ნიღბების განსაზღვრა

თითოეული HEX ციფრი შეესაბამება ზუსტად 4 ბიტს (Nibble[1]). ამიტომ:

8-ბიტიანი ტიპი (uint8_t): მოიცავს 2 HEX ციფრს (0x00 – 0xFF).

16-ბიტიანი ტიპი (uint16_t): მოიცავს 4 HEX ციფრს (0x0000 – 0xFFFF).

32-ბიტიანი ტიპი (uint32_t): მოიცავს 8 HEX ციფრს (0x00000000 – 0xFFFFFFFF).

კოდი 9.1 16 და 32-ბიტიანი ნიღბების განსაზღვრა enum და წანაცვლებით

```c
#include <stdint.h>

/* 16-ბიტიანი ნიღბები (მაგალითად, ქსელური პაკეტის ან მოწყობილობის დროშები) */
enum Flag16 : uint16_t {
 FLAG16_NONE  = 0x0000U,   /* 0b00000000 00000000 */
 FLAG16_READ  = 1U << 0,   /* Bit 0  (0b00000000 00000001) */
 FLAG16_WRITE = 1U << 1,   /* Bit 1  (0b00000000 00000010) */
 FLAG16_EXEC  = 1U << 2,   /* Bit 2  (0b00000000 00000100) */
 FLAG16_ERROR = 1U << 15,  /* Bit 15 (0b10000000 00000000) */
 FLAG16_ALL   = 0xFFFFU    /* ყველა 16 ბიტი 1-ია */
};

```

[1] ნიბლი (Nibble ან Nybble) კომპიუტერულ მეცნიერებაში ეწოდება 4-ბიტიან ჯგუფს.

```
/* 32-ბიტიანი ნიღბები (მაგალითად, 32-ბიტიანი მიკროპროცესორის რეგისტრი) */
#define SYS_ENABLE     (1UL << 0)    /* 0x00000001 -> Bit 0 */
/* (0b00000000...00000001) */
#define SYS_READY      (1UL << 1)    /* 0x00000002 -> Bit 1 */
/* (0b00000000...00000010) */
#define SYS_MODE_MASK (0x000000F0UL) /* 0x000000F0 -> Bits 4..7 */
/* (0b0000...11110000) */
#define SYS_IRQ_BIT    (1UL << 31)  /* 0x80000000 -> Bit 31 */
/* (0b10000000...00000000) */
#define SYS_ALL_MASK   (0xFFFFFFFFUL)
```

მნიშვნელოვანი წესი: 32-ბიტიანი ნიღბების წანაცვლებისას აუცილებლად გამოიყენეთ `1UL << n` (ან `1U << n`), რადგან უბრალო `1 << 31` ჩვეულებრივ `signed int`-ად აღიქმება და 31-ე ბიტზე გადასვლისას გამოიწვევს ნიშნის გადავსებას (Undefined Behavior)!

9.5.2 ბიტური ოპერაციების მაგალითი

განვიხილოთ ბიტური ოპერაციების მაგალითი:

კოდი 9.2 32-ბიტიანი დროშების მართვა

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

/* თითოეული ბიტის პოზიცია ორობით (Binary) და თექვსმეტობით (Hex) ფორმატში */
#define BIT_POWER  (1UL << 0)  /* 0x00000001 -> Bit 0 */
/* (0b00000000 00000000 00000000 00000001) */
#define BIT_WIFI   (1UL << 4)  /* 0x00000010 -> Bit 4 */
/* (0b00000000 00000000 00000000 00010000) */
#define BIT_BLE    (1UL << 5)  /* 0x00000020 -> Bit 5 */
/* (0b00000000 00000000 00000000 00100000) */
#define BIT_ALARM  (1UL << 31) /* 0x80000000 -> Bit 31 */
/* (0b10000000 00000000 00000000 00000000) */

int main(void)
{
 uint32_t status = 0; /* 0000 ... 0000 */

 /* 1. ჩართვა (Set) - ბიტური OR (|=) */
 status |= (BIT_POWER | BIT_WIFI); /* ჩართავს POWER და WIFI ბიტებს */

```

```
21  /* 2. გამორთვა (Clear/Reset) - ბიტური AND + NOT (&= ~) */
22  status &= ~BIT_WIFI; /* გამორთავს WIFI ბიტს, დანარჩენს არ ეხება */
23
24  /* 3. გადართვა (Toggle) - ბიტური XOR (^=) */
25  status ^= BIT_BLE; /* თუ BLE ეწერა 0, გახდება 1; თუ 1 იყო - 0 */
26
27  /* 4. შემოწმება (Test/Check) - ბიტური AND (&) */
28  if ((status & BIT_POWER) != 0)
29  {
30   printf("Device power is ON.\n");
31  }
32
33  if ((status & (BIT_WIFI | BIT_BLE)) == 0)
34  {
35   printf("Wireless modules are OFF.\n");
36  }
37
38  /* ყველა ბიტის გასუფთავება */
39  status &= ~0xFFFFFFFFUL; /* ან status = 0; */
40
41  return 0;
42 }
```

9.5.3 უნივერსალური ბიტური მაკროსები (8, 16, 32, 64 ბიტისთვის)

ხშირად გამოიყენება უნივერსალური მაკროსები, რომლებიც მუშაობენ ნებისმიერი სიგანის ტიპებთან:

```
1  /* უსაფრთხო მაკროსები (1UL უზრუნველყოფს 32/64 ბიტიან მხარდაჭერას) */
2  #define BitSet(arg, posn) ((arg) |= (1UL << (posn)))
3  #define BitClr(arg, posn) ((arg) &= ~(1UL << (posn)))
4  #define BitFlp(arg, posn) ((arg) ^= (1UL << (posn)))
5  #define BitTst(arg, posn) (((arg) & (1UL << (posn))) != 0)
```

გამოყენების მაგალითი:

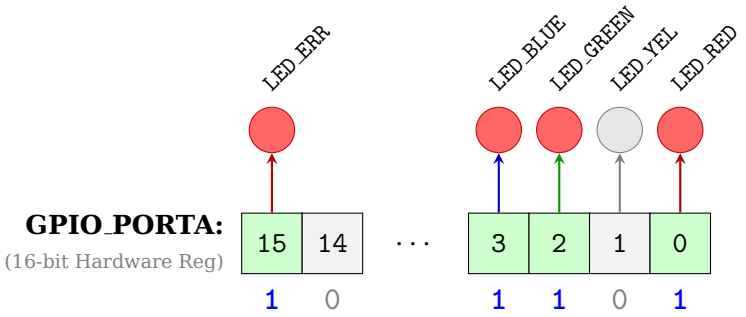
```
1  uint32_t reg = 0;
2
3  BitSet(reg, 0);  /* ჩართავს 0-ოვან ბიტს */
4  BitSet(reg, 15); /* ჩართავს 15-ე ბიტს (16-ბიტიანი ზღვარი) */
5  BitSet(reg, 31); /* ჩართავს 31-ე ბიტს (32-ბიტიანი ზღვარი) */
```

```
6
7  if (BitTst(reg, 15))
8  {
9   BitClr(reg, 15); /* გამორთავს 15-ე ბიტს */
10 }
```

9.6 პრაქტიკული გამოყენება: აპარატურული რეგისტრებისა და LED-ების მართვა

მიკროპროცესორებსა და მიკროკონტროლერებში (მაგალითად, ARM Cortex-M, STM32, AVR, ESP32) ფიზიკური პინები (GPIO - General Purpose Input/Output), რომლებზეც მიერთებულია შუქდიოდები (LED), ღილაკები და სენსორები, იმართება **16 ან 32-ბიტიანი აპარატურული რეგისტრების** ბიტების ცვლილებით.

როდესაც რეგისტრის შესაბამის ბიტში ჩავწერთ 1-ს, პინზე მიეწოდება ძაბვა (მაგ. +3.3V) და LED აინთება, ხოლო 0-ის ჩაწერისას — ჩაქვრება.



სურ. 9.1 16-ბიტიანი აპარატურული რეგისტრის ბიტების მიერთება ფიზიკურ LED-ებთან

9.6.1 ლაბორატორიული მაგალითები: LED-ების მართვის სცენარები

განვიხილოთ 5 პრაქტიკული მაგალითი 16 და 32-ბიტიანი რეგისტრებისთვის.

მაგალითი 1: კონკრეტული LED-ის ჩართვა (Bit Set / Bitwise OR) თუ გვინდა ჩავრთოთ მხოლოდ 0-ოვანი ბიტი (წითელი LED), ისე რომ რეგისტრის დანარჩენი 15 ბიტის მდგომარეობა არ შეიცვალოს:

```
1  #define LED_RED_PIN   (1U << 0)   /* 0x0001 -> Bit 0 */
2  /* (0b00000000 00000001) */
3
4  uint16_t GPIO_PORTA = 0x0000;    /* თავდაპირველად ყველა LED ჩაქრობილია */
5
6  /* წითელი LED-ის ანთება */
7  GPIO_PORTA |= LED_RED_PIN;       /* 0x0000 | 0x0001 = 0x0001 */
```

მაგალითი 2: LED-ის გამორთვა (Bit Clear / Bitwise AND + NOT) თუ გვინდა LED_RED ↪ -ის ჩაქრობა ისე, რომ სხვა LED-ებს ხელი არ ვახლოთ, ვიყენებთ &= ~MASK კომბინაციას.

GPIO_PORTA:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
~LED_RED_PIN:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
შედეგი (&):	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

სურ. 9.2 ბიტის გამორთვის ლოგიკა &= ~MASK ოპერაციით

```
/* წითელი LED-ის ჩაქრობა */
GPIO_PORTA &= ~LED_RED_PIN;      /* 0x0001 & 0xFFFE = 0x0000 */
```

მაგალითი 3: LED-ის მდგომარეობის შეცვლა/ციმციმი (Bit Toggle / Bitwise XOR) თუ გვინდა, რომ LED-მა იციმციმოს (ყოველ გამოძახებაზე აინთოს თუ ჩაქრობილია, ან ჩაქრეს თუ ანთებულია), ვიყენებთ XOR (^=) ოპერატორს:

```
#define LED_BLUE_PIN (1U << 3)  /* 0x0008 -> Bit 3 */
/* (0b00000000 00001000) */

/* ლურჯი LED-ის გადართვა (Toggle) */
GPIO_PORTA ^= LED_BLUE_PIN;      /* თუ ეწერა 0, გახდება 1; თუ 1 იყო, გახდე
  ↪ ბა 0 */
```

მაგალითი 4: რამდენიმე LED-ის ერთდროულად მართვა (Group Masking) 32-ბიტიან სისტემებში (მაგ. STM32) ხშირად საჭიროა LED-ების ჯგუფური მართვა. განვიხილოთ 32-ბიტიანი რეგისტრი:

კოდი 9.3 32-ბიტიანი GPIO რეგისტრის ჯგუფური მართვა

```
#include <stdint.h>

#define LED_RED_PIN    (1UL << 0)   /* Bit 0 */
/* (0b00000000 00000000 00000000 00000001) */
#define LED_GREEN_PIN  (1UL << 2)   /* Bit 2 */
/* (0b00000000 00000000 00000000 00000100) */
#define LED_BLUE_PIN   (1UL << 3)   /* Bit 3 */
/* (0b00000000 00000000 00000000 00001000) */
#define LED_ERR_PIN    (1UL << 15)  /* Bit 15 */
/* (0b00000000 00000000 10000000 00000000) */
#define SYSTEM_ALARM   (1UL << 31)  /* Bit 31 */
/* (0b10000000 00000000 00000000 00000000) */
```

```
13
14 #define ALL_LEDS_MASK  (LED_RED_PIN | LED_GREEN_PIN | LED_BLUE_PIN |
     ↪ LED_ERR_PIN)
15
16 int main(void)
17 {
18  uint32_t GPIO_PORT = 0;
19
20  /* ა) ყველა LED-ის ერთდროულად ანთება */
21  GPIO_PORT |= ALL_LEDS_MASK;
22
23  /* ბ) მხოლოდ GREEN და BLUE LED-ების ჩაქრობა */
24  GPIO_PORT &= ~(LED_GREEN_PIN | LED_BLUE_PIN);
25
26  /* გ) განგაშის (ALARM) ბიტის ჩართვა (31-ე ბიტი) */
27  GPIO_PORT |= SYSTEM_ALARM;
28
29  /* დ) ყველა LED-ის ჩაქრობა, მაგრამ ALARM ბიტის შენარჩუნება */
30  GPIO_PORT &= ~ALL_LEDS_MASK;
31
32  return 0;
33 }
```

მაგალითი 5: ღილაკის/სენსორის მდგომარეობის წაკითხვა (Bit Testing) დავუშვათ, 16-ბიტიანი რეგისტრის მე-7 ბიტზე მიერთებულია ფიზიკური ღილაკი. გვინდა შევამოწმოთ, დააჭირა თუ არა მომხმარებელმა თითი ღილაკს:

```
1  #define BUTTON_PIN  (1U << 7)   /* Bit 7: 0x0080 */
2  /* (0b00000000 10000000) */
3
4  uint16_t GPIO_INPUT_REG = 0x0080; /* მე-7 ბიტი 1-ია (ღილაკი დაჭერილია) */
5
6  /* შემოწმება */
7  if ((GPIO_INPUT_REG & BUTTON_PIN) != 0)
8  {
9   /* ღილაკი დაჭერილია - ავანთოთ წითელი LED */
10  GPIO_PORTA |= LED_RED_PIN;
11 }
12 else
13 {
```

```
14   /* ღილაკი აშვებულია - ჩავაქროთ წითელი LED */
15   GPIO_PORTA &= ~LED_RED_PIN;
16  }
```

შეჯამება მომავალი მიკროპროცესორების კურსისთვის:

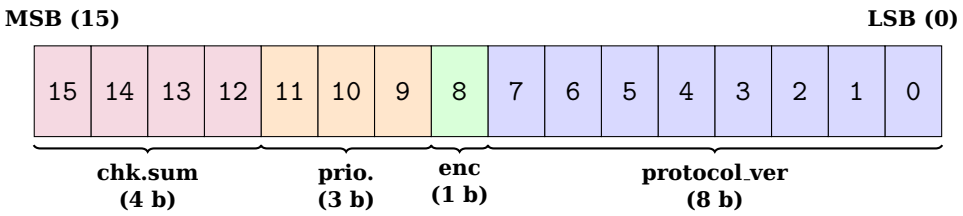
- `REG |= (1 << n);` → **ბიტის ჩართვა** (LED-ის ანთება).
- `REG &= ~(1 << n);` → **ბიტის გამორთვა** (LED-ის ჩაქრობა).
- `REG ^= (1 << n);` → **ბიტის გადართვა** (LED-ის ციმციმი).
- `if (REG & (1 << n))` → **ბიტის შემოწმება** (ღილაკის წაკითხვა).

9.7 ბიტური ველები (Bit Fields)

ბიტური ველები (Bit Fields) საშუალებას იძლევა `struct`-ის შიგნით ცვლადებს განვუსაზღვროთ კონკრეტული ზომა **ბიტების ზუსტი რაოდენობით** და არა სტანდარტული ბაიტებით. ეს არის ბიტური ნიღბების ალტერნატივა, რომელიც კოდს უფრო კითხვადს ხდის.

სინტაქსი:

```
struct StructName {
 type field_name : width_in_bits;
};
```



სურ. 9.3 16-ბიტიანი სტრუქტურის (`HeaderFlags16`) ბიტების განლაგება მეხსიერებაში

9.7.1 8, 16 და 32-ბიტიანი სტრუქტურის მაგალითები

```
1  #include <stdint.h>
2
3  /* 8-ბიტიანი სტრუქტურა (ჯამში 8 ბიტი = 1 ბაიტი) */
4  struct StatusRegister8 {
5   uint8_t enable   : 1; /* 1 ბიტი: 0 ან 1 */
6   uint8_t ready    : 1; /* 1 ბიტი */
7   uint8_t mode     : 3; /* 3 ბიტი: მნიშვნელობები 0..7 */
8   uint8_t error    : 1; /* 1 ბიტი */
9   uint8_t reserved : 2; /* 2 ბიტი (რეზერვი) */
10 };
11
```

```c
/* 16-ბიტიანი სტრუქტურა (ჯამში 16 ბიტი = 2 ბაიტი) */
struct HeaderFlags16 {
 uint16_t protocol_ver : 8; /* 0..255 */
 uint16_t is_encrypted : 1; /* 0..1 */
 uint16_t priority     : 3; /* 0..7 */
 uint16_t checksum     : 4; /* 0..15 */
};

/* 32-ბიტიანი ტაიმერის აპარატურული რეგისტრი */
struct TimerControlReg32 {
 uint32_t timer_enable : 1;  /* ბიტი 0: ტაიმერის ჩართვა */
 uint32_t clock_source : 3;  /* ბიტები 1-3: საათის წყარო (0..7) */
 uint32_t interrupt_en : 1;  /* ბიტი 4: ინტერრაპტის ნებართვა */
 uint32_t prescaler    : 8;  /* ბიტები 5-12: გამყოფი (0..255) */
 uint32_t reserved     : 18; /* ბიტები 13-30: რეზერვი */
 uint32_t overflow_flag: 1;  /* ბიტი 31: გადავსების დროშა */
};
```

9.7.2 უსახელო ველები (Unnamed Bit Fields) და მეხსიერების გასწორება

C ენაში ბიტურ ველებში დაშვებულია **სახელის გამოტოვება (Unnamed Bit Fields)**. მათი დანიშნულებაა კომპილატორის მართვა და მეხსიერების განლაგების (Layout) ზუსტი რეგულირება.

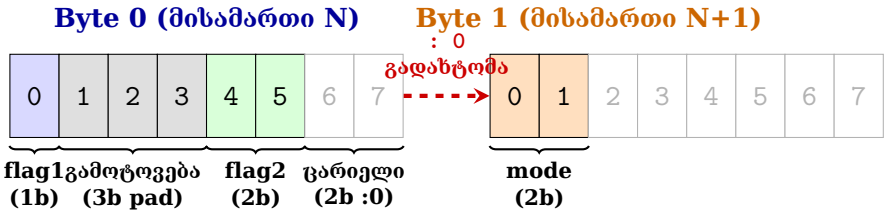
არსებობს უსახელო ველების ორი ძირითადი ტიპი:

1. **დადებითი სიგანის უსახელო ველი (`type : N`):** გამოიყენება მეხსიერებაში ბიტების გამოტოვებისთვის (Padding/Reserved). კოდიდან ამ ბიტებზე მიმართვა შეუძლებელია.
2. **ნულოვანი სიგანის უსახელო ველი (`type : 0`):** არ იკავებს ბიტებს. ის არის სპეციალური ბრძანება კომპილატორისთვის: *„დაუყოვნებლივ დახურე მიმდინარე ბაიტი/სიტყვა და მომდევნო ველი დაიწყე ახალი მეხსიერების ერთეულის (Byte/Word) ზღვარიდან!“*

კოდი 9.5 უსახელო ველების და :0-ის გამოყენება

```c
#include <stdint.h>

struct PaddedRegister {
 /* --- Byte 0 (Address N) --- */
 uint8_t flag1 : 1; /* Bit 0 */
 uint8_t       : 3; /* გამოტოვებს 3 ბიტს (Bits 1..3 - Padding) */
 uint8_t flag2 : 2; /* Bits 4..5 (Byte 0-ში შეივსო მხოლოდ 6 ბიტი) */

```

```
    /* :0 აიძულებს კომპილატორს დატოვოს დარჩენილი 2 ბიტი (Bits 6..7)
    და გადავიდეს Byte 1-ზე */
    uint8_t        : 0;

    /* --- Byte 1 (Address N+1) --- */
    uint8_t mode   : 2; /* იწყება ახალი ბაიტის თავიდან (Bits 0..1) */
};
```



სურ. 9.4 ნულოვანი სიგანის ბიტველის (: 0) მიერ ახალ ბაიტზე გადახტომის სქემა

ნულოვანი სიგანის ბიტველის (: 0) ქცევის წესები:

- **ნაწილობრივ შევსებულ ბაიტზე:** ხურავს მიმდინარე ბაიტს/სიტყვას (ტოვებს დარჩენილ ბიტებს გამოუყენებლად) და გადადის მომდევნო ახალ ბაიტზე.
- **უკვე ახალ/ცარიელ ბაიტზე:** თუ მიმდინარე ბაიტში ჯერ არცერთი ბიტი არ არის ჩაწერილი, : 0 არაფერს აკეთებს (იგნორირდება).

```
struct Test {
    /* --- Byte 0 --- */
    uint8_t flag1 : 1; /* Byte 0, Bit 0 */
    uint8_t       : 0; /* ხურავს Byte 0-ს -> გადადის Byte 1-ზე */

    /* --- Byte 1 --- */
    uint8_t flag2 : 1; /* Byte 1, Bit 0 (დაიკავა 1 ბიტი) */
    uint8_t       : 0; /* რადგან Byte 1-ში ბიტი ჩაჯდა, ახლა ხურავს Byte 1-ს
     ↪ -> გადადის Byte 2-ზე! */

    /* --- Byte 2 --- */
    uint8_t flag3 : 4; /* Byte 2, Bits 0..3 */
};
```

9.7.3 K&R კრიტიკული გაფრთხილება: signed vs unsigned ბიტური ველები

ხშირია სერიოზული ლოგიკური შეცდომა **1-ბიტიანი ნიშნიანი (signed) ველის** გამოყენებისას:

```
struct BadStruct {
```

```
 signed int flag : 1; /* შეცდომის წყარო! */
};

struct BadStruct s;
s.flag = 1;

if (s.flag == 1) {
 /* ეს პირობა ARASODES შესრულდება! */
}
```

რატომ ხდება ეს შეცდომა? 1-ბიტიან **ნიშნიან** (signed) ცვლადში უფროსი (და ერთადერთი) ბიტი მიჩნეულია ნიშნის ბიტად. დამატებით კოდში (Two's Complement) 1-ბიტიან სივრცეში ეტევა მხოლოდ ორი მნიშვნელობა:

- 0b0 → 0
- 0b1 → -1

როდესაც s.flag-ს მივანიჭებთ 1-ს, მეხსიერებაში ჩაიწერება ბიტი 1, რაც ნიშნიანი ინტერპრეტაციით არის **-1**! შესაბამისად, s.flag == 1 შემოწმება იძლევა -1 == 1-ს, რაც მცდარია (false).

```
struct GoodStruct {
 unsigned int flag : 1; /* სწორია! მნიშვნელობები: 0 ან 1 */
};
```

წესი: ბიტური ველებისთვის **ყოველთვის გამოიყენეთ unsigned ტიპები** (uint8_t, uint16_t, uint32_t ან unsigned int), თუ უარყოფითი რიცხვების შენახვა სპეციალურად არ გჭირდებათ.

9.7.4 Unions + Bit Fields: აპარატურული რეგისტრების იდეალური მოდელი

ჩაშენებულ სისტემებში (Embedded Systems) ყველაზე გავრცელებული და პროფესიონალური პრაქტიკაა **union-ისა და ბიტური ველების გაერთიანება**.

ეს იძლევა ორმაგ უპირატესობას:

1. ცალკეულ აპარატურულ დროშებს მივმართოთ სახელით (მაგ. reg.bits.tx_enable = 1).
2. მთლიანი რეგისტრი წავიკითხოთ ან გავგზავნოთ რიცხვის/ბაიტის სახით (მაგ. SPI_Send(reg.raw)).

კოდი 9.6 Unions და Bit Fields-ის კომბინაცია რეგისტრებისთვის

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

```

```c
/* 16-ბიტიანი UART კონტროლერის რეგისტრის მოდელი */
typedef union {
 uint16_t raw; /* მთლიანი 16-ბიტიანი რეგისტრი (0x0000 - 0xFFFF) */

 struct {
  /* Byte 0 */
  uint8_t tx_enable   : 1; /* Bit 0: TX Enable */
  uint8_t rx_enable   : 1; /* Bit 1: RX Enable */
  uint8_t             : 2; /* Bits 2..3: Reserved */
  uint8_t parity_mode : 2; /* Bits 4..5: Parity configuration */
  uint8_t             : 2; /* Bits 6..7: Reserved */

  uint8_t             : 0; /* იძულებითი გადაყვანა Byte 1-ზე */

  /* Byte 1 */
  uint8_t baud_rate   : 4; /* Bits 8..11: Baud rate selection */
  uint8_t status_flag : 4; /* Bits 12..15: Status flags */
 } bits;
} UART_ControlRegister;

int main(void)
{
 UART_ControlRegister reg;
 reg.raw = 0x0000; /* რეგისტრის სრული გასუფთავება */

 /* ცალკეულ ველებზე მიმართვა */
 reg.bits.tx_enable = 1;
 reg.bits.rx_enable = 1;
 reg.bits.parity_mode = 2;   /* 10 binary */
 reg.bits.baud_rate = 9;     /* 9600 baud rate code */

 /* HEX მნიშვნელობის დაბეჭდვა */
 printf("Register Raw HEX Value: 0x%04X\n", reg.raw);
 printf("Struct Size: %zu Byte(s)\n", sizeof(UART_ControlRegister));

 return 0;
}
```

9.7.5 ბიტური ველების შეზღუდვები და არაპორტაბელურობა

მიუხედავად იმისა, რომ ბიტური ველები მოსახერხებელია, მათ გააჩნიათ სერიოზული შეზღუდვები:

1. **ჩალაგების მიმართულება (Endianness & Bit Ordering):** C სტანდარტი არ განსაზღვრავს, ბიტები მარცხნიდან მარჯვნივ ჩალაგდება მეხსიერებაში (MSB first) თუ მარჯვნიდან მარცხნივ (LSB first). ეს სრულიად დამოკიდებულია არქიტექტურასა და კომპილატორზე.
2. **მისამართის აღების შეუძლებლობა:** ბიტურ ველზე **არ შეიძლება** მისამართის მიმთითებლის აღება (`&reg.bits.tx_enable` არაკომპილირებადია!), რადგან პროცესორის მისამართები გაიცემა მხოლოდ ბაიტის დონეზე.
3. **მასივების აკრძალვა:** ბიტური ველებისგან **შეუძლებელია მასივის შექმნა** (მაგ. `uint8_t flags[10] : 1;` სინტაქსურად აკრძალულია).
4. **მაქსიმალური სიგანე:** ველის სიგანე არ უნდა აღემატებოდეს საბაზო ტიპის ბიტების სიგანეს (მაგ. `uint16_t`-ში ველის სიგანე ვერ იქნება 17).

დასკვნა: დაბალი დონის კრიტიკულ პროგრამირებაში, სადაც მეხსიერებაში ბიტების ზუსტი განლაგება აუცილებელია (მაგ. ქსელური ოქმები, დისკური ფაილების სათაურები, დრაივერები), **ბიტური ნიღბები** (`&`, `|`, `~`) უფრო საიმედო და პორტაბელურია, ვიდრე ბიტური ველები.

თავი 10

მონაცემების დამუშავება და მაღალი წარმადობის გამოთვლები

10.1 FISO სენსორით გაზომილი მონაცემების წაკითხვა 203
10.2 მასივების სტრუქტურა თუ სტრუქტურების მასივი (AoS vs SoA) . . . 209
10.3 Fortran vs C/C++: წარმადობა, IEEE 754 და მეხსიერების არქიტექტურა 210
10.3.1 `restrict` საკვანძო სიტყვა და Pointer Aliasing 210
10.3.2 IEEE 754 სტანდარტი და კომპლექსური გამრავლება 211
10.3.3 მათემატიკური მატრიცა და მეხსიერების ლეიოუტი: Row-Major vs Column-Major 211
10.3.4 წარმადობა (Performance) და მატრიცების გამრავლება 213
10.3.5 მაგალითი: OpenGL (Column-Major) მატრიცის ჩალაგება, ინიციალიზაცია და დაბეჭდვა C-ში 215
10.4 Fortran და C კოდები, ჩალაგება და წარმადობა. 217
10.5 GFLOPS-ის გამოთვლის ფორმულის დეტალური ანალიზი 225
10.6 ახალი მათემატიკური ფუნქციები 226
10.7 C დროის ფუნქციები vs `C++ std::chrono` 226

10.1 FISO სენსორით გაზომილი მონაცემების წაკითხვა

კოდი **10.1** მონაცემების წაკითხვა

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
bool loadMeasuredT_C(const char* filename, float** xData, float**
    ↪ yData, size_t* data_count);
int main(int argc, char *argv[]) {
    // 1. ინპუტ ფაილის განსაზღვრა (ნაგულისხმევად: sensor_data.txt)
    const char *input_filename = "sensor_data.txt";
```

```
    if (argc > 1) {
        input_filename = argv[1];
    }

    const char *output_filename = "converted.txt";

    float *xData = NULL;
    float *yData = NULL;
    size_t count = 0;

    printf("Reading from '%s'...\n", input_filename);

    // 2. ფაილის წაკითხვა
    if (loadMeasuredT_C(input_filename, &xData, &yData, &count)) {
        printf("Successfully read %zu records.\n", count);

        // 3. თუ წაკითხვა წარმატებულია, გახსნის converted.txt-ს ჩასაწერად
        FILE *pfile_write = fopen(output_filename, "wt");
        if (!pfile_write) {
            puts("Error: Could not open output file for writing.");
            free(xData);
            free(yData);
            return EXIT_FAILURE;
        }

        // პირველ ხაზზე N # <რაოდენობა>
        fprintf(pfile_write, "N # %zu\n", count);
        fprintf(pfile_write, "Time_Sec\tTemperature\n");

        // მონაცემების ჩაწერა (t წამებში და data)
        for (size_t i = 0; i < count; i++) {
            fprintf(pfile_write, "%.6f\t%.3f\n", xData[i], yData[i]);
        }

        fclose(pfile_write);
        printf("Data written to '%s' successfully.\n", output_filename
    ↪ );

        // მეხსიერების გასუფთავება
        free(xData);
```

```
48          free(yData);
49      } else {
50          printf("Failed to read input file. Output file '%s' was NOT
    ↪ created.\n", output_filename);
51          return EXIT_FAILURE;
52      }
53
54      return EXIT_SUCCESS;
55  }
56
57  bool loadMeasuredT_C(const char* filename, float** xData, float**
    ↪ yData, size_t* data_count)
58  {
59      if (!filename || !xData || !yData || !data_count) {
60          return false;
61      }
62
63      *xData = NULL;
64      *yData = NULL;
65      *data_count = 0;
66
67      FILE *pfile = fopen(filename, "rt");
68      if (!pfile) {
69          printf("Error: Could not open input file '%s'.\n", filename);
70          return false;
71      }
72
73      char filedata[512];
74
75      // 1. [Data]  შემოწმება
76      if (fgets(filedata, sizeof(filedata), pfile) != NULL) {
77          filedata[strcspn(filedata, "\r\n")] = 0;
78          if (strcmp(filedata, "[Data]") != 0) {
79              puts("Error: Missing [Data] header.");
80              fclose(pfile);
81              return false;
82          }
83      } else {
84          puts("Error: File is empty.");
85          fclose(pfile);
```

```
86          return false;
87      }
88
89      //  მონაცემების დასაწყისის დამახსოვრება
90      long data_start_pos = ftell(pfile);
91
92      char strdate[32], strvalid[32];
93      int hour, minute;
94      double second;
95      double valT = -1.0;
96      size_t linecount = 0;
97
98      // ========================================
99      //  1: ხაზების დათვლა მეხსიერების ალოკაციისთვის
100     // ========================================
101     while (fgets(filedata, sizeof(filedata), pfile) != NULL) {
102         filedata[strcspn(filedata, "\r\n")] = 0;
103         if (strlen(filedata) == 0) continue;
104
105         int numread = sscanf(filedata, "%s %d:%d:%lf %lf %s",
106                              strdate, &hour, &minute, &second, &valT,
    ↪ strvalid);
107         if (numread == 6) {
108             linecount++;
109         }
110     }
111
112     if (linecount == 0) {
113         puts("Warning: No valid data lines found.");
114         fclose(pfile);
115         return false;
116     }
117
118     // ========================================
119     // მეხსიერების გამოყოფა
120     // ========================================
121     *xData = (float*)malloc(linecount * sizeof(float));
122     *yData = (float*)malloc(linecount * sizeof(float));
123
124     if (!*xData || !*yData) {
```

```
125         puts("Error: Memory allocation failed.");
126         free(*xData);
127         free(*yData);
128         fclose(pfile);
129         return false;
130     }
131 
132     // ========================================
133     //  2: მონაცემების ამოკითხვა წამებში
134     // ========================================
135     fseek(pfile, data_start_pos, SEEK_SET);
136 
137     double start_total_seconds = -1.0;
138     size_t index = 0;
139 
140     while (fgets(filedata, sizeof(filedata), pfile) != NULL && index <
    ↪  linecount) {
141         filedata[strcspn(filedata, "\r\n")] = 0;
142         if (strlen(filedata) == 0) continue;
143 
144         int numread = sscanf(filedata, "%s %d:%d:%lf %lf %s",
145                              strdate, &hour, &minute, &second, &valT,
    ↪ strvalid);
146         if (numread != 6) continue;
147 
148         // სრული წამები დღის დასაწყისიდან
149         double current_total_seconds = (hour * 3600.0) + (minute *
    ↪ 60.0) + second;
150 
151         if (start_total_seconds < 0) {
152             start_total_seconds = current_total_seconds;
153         }
154 
155         // ათვლა 0-დან წამებში
156         double elapsed_seconds = current_total_seconds -
    ↪ start_total_seconds;
157 
158         (*xData)[index] = (float)elapsed_seconds; // წამები
159         (*yData)[index] = (float)valT;
160 
```

```
161         index++;
162     }
163
164     fclose(pfile);
165     *data_count = index;
166     return true;
167 }
```

დავალება 10.1: რთული ფორმატის ფაილის წაკითხვა

რეალურ ფაილში [Data] ხაზამდე წერია სხვა ინფორმაცია. ამოცანაა „გადავახტეთ“ ამ ინფორმაციას. ვიპოვოთ [Data] და ამის შემდეგ წავიკითხოთ მონაცემები. თუ [Data] ვერ მოიძებნა, მაშინ გამოდის შეტყობინება, რომ [Data] ვერ იქნა ნაპოვნი და პროგრამა ამთავრებს მუშაობას. ქვემოთ მოყვანილია ეს კოდი.

შეიტანეთ შესაბამისი ცვლილებები მონაცემების წაკითხვის პროგრამაში და დარწმუნდით, რომ პროგრამა სწორად მუშაობს.

```
1  FILE *fp = fopen("data.txt", "r");
2  if (!fp) {
3   perror("ver gaikhsna faili");
4   return 1;
5  }
6
7  char line[256];
8  bool found = false;
9
10 // 1. ეძებს "[Data]" სექციას
11 while (fgets(line, sizeof(line), fp)) {
12  if (strstr(line, "[Data]") != NULL) {
13   found = true;
14   break; // [Data] ნაპოვნია, გამოვდივართ ძებნის ციკლიდან
15  }
16 }
17
18 // შემოწმება: თუ [Data] ვერ მოიძებნა
19 if (!found) {
20  fprintf(stderr, "Shecdoma: failshi [Data] ver moidzebna!\n");
21  fclose(fp);
22  return 1;
23 }
24
```

```c
// 2. მონაცემების წაკითხვა (მხოლოდ იმ შემთხვევაში, თუ [Data] ნაპოვნია)
char date[20], time[20], status[20];
double val;

while (fscanf(fp, "%19s %19s %lf %19s", date, time, &val, status) ==
  ↪ 4) {
 printf("Date: %s | Time: %s | Val: %.3f | Status: %s\n", date, time,
  ↪ val, status);
}
// აქ უკვე წაკითხული გვაქვს მონაცემები და შეგვიძლია
// დავამუშავოთ როგორც ზემოთ იყო კოდში მოცემული
fclose(fp);
return 0;
```

დავალება 10.2: კონვერტირებული ფაილის წაკითხვა

დაწერეთ კოდი, რომელიც წაიკითხავს მონაცემებს ახალი ფორმატის `converted.txt` ფაილიდან.

10.2 მასივების სტრუქტურა თუ სტრუქტურების მასივი (AoS vs SoA)

მონაცემთა ორგანიზებისას არსებობს ორი ფუნდამენტური მიდგომა:

1. **AoS (Array of Structures - სტრუქტურების მასივი):** ობიექტზე ორიენტირებული მიდგომა.
2. **SoA (Structure of Arrays - მასივების სტრუქტურა):** მონაცემებზე ორიენტირებული დიზაინი (Data-Oriented Design).

```c
/* 1. AoS (Array of Structures) */
typedef struct {
 float x, y, z;
 int id;
} Particle;

Particle particles[1000]; /* ობიექტების მასივი */

/* 2. SoA (Structure of Arrays) */
typedef struct {
 float x[1000];
 float y[1000];
 float z[1000];
 int id[1000];
} ParticleSystem;
```

```
16
17 ParticleSystem system_particles;
```

როდის რომელი მიდგომა ავირჩიოთ?

ცხრილი 10.1 AoS და SoA არქიტექტურების შედარება

მახასიათებელი	AoS (Array of Structs)	SoA (Structure of Arrays)
დიზაინი	ობიექტზე ორიენტირებული	მონაცემებზე ორიენტირებული
წვდომა	მოსახერხებელია 1 ობიექტისთვის	მოსახერხებელია 1 თვისებისთვის
Cache Locality	ცუდი (თუ მხოლოდ X კოორდინატს ვამუშავებთ)	შესანიშნავი (მეხსიერება უწყვეტია)
SIMD/Vectorization	რთულად ვექტორიზებადი	იდეალურია CPU/GPU SIMD ინსტრუქციებისთვის

დასკვნა: თუ თქვენ აპროგრამებთ თამაშის ძრავას, გრაფიკას ან სამეცნიერო სიმულაციას, სადაც ათასობით ნაწილაკის მხოლოდ X კოორდინატი უნდა განახლდეს ციკლში, **SoA** იძლევა კოლოსალურ უპირატესობას პროცესორის ქეშის (L1/L2 Cache) ეფექტური გამოყენებისა და SIMD ვექტორიზაციის გამო. ხოლო ჩვეულებრივი ბიზნეს-ლოგიკისთვის **AoS** უფრო მარტივი და წაკითხვადია.

10.3 Fortran vs C/C++: წარმადობა, IEEE 754 და მეხსიერების არქიტექტურა

C89 (ANSI C / ISO C90) იყო ენის პირველი ოფიციალური სტანდარტი. მას შემდეგ C ენამ განვითარების რამდენიმე მნიშვნელოვანი ეტაპი გაიარა: **C99**, **C11**, **C17** და **C23**.

მთავარი მიზანი იყო მაღალი წარმადობის (HPC) გამოთვლების მხარდაჭერა, ტიპების უსაფრთხოება, მრავალნაკადიანობა (Multi-threading) და აპარატურულ დონეზე ოპტიმიზაციის შესაძლებლობა.

ისტორიულად, Fortran-ი ითვლებოდა სამეცნიერო გამოთვლებისა და მრავალგანზომილებიანი (3D/4D) ვექტორული ოპერაციების შეუდარებელ ლიდერად. C-სა და C++-ში კომპლექსური ტიპების გამოყენებისას ხშირად შეიმჩნეოდა წარმადობის მნიშვნელოვანი ჩამორჩენა. ეს გამოწვეული იყო ენობრივი სტანდარტებისა და მეხსიერების მართვის სამი ფუნდამენტური განსხვავებით.

10.3.1 `restrict` საკვანძო სიტყვა და Pointer Aliasing

Fortran ენის ისტორიული უპირატესობა სამეცნიერო გამოთვლებში განპირობებული იყო იმით, რომ Fortran კომპილატორს აქვს გარანტია: სხვადასხვა მასივები მეხსიერებაში არასოდეს გადაფარავენ ერთმანეთს (**No Pointer Aliasing**). C-ში პოინტერების გამო

კომპილატორს ეშინია, რომ $A[i]$-ში შენახვამ შეიძლება შეცვალოს $B[i]$ მონაცემი, ამის გამო კომპილატორი უარს ამბობს აგრესიულ SIMD (AVX/AVX-512) ვექტორიზაციაზე, რის გამოც გადადის უსაფრთხო, მაგრამ ნელ რეჟიმში.

C99-ში დამატებულმა `restrict` მოდიფიკატორმა ეს პრობლემა გადაჭრა და C ენას მისცა Fortran-ის იდენტური ოპტიმიზაციის გარანტია:

```
void add_vectors(double * restrict A, const double * restrict B);
```

`restrict` ეუბნება კომპილატორს, რომ მიმთითებელი არის ერთადერთი გზა ამ მეხსიერების ბლოკზე წვდომისთვის, რაც რთავს **SIMD / AVX2 / AVX-512** ავტომატურ ვექტორიზაციას.

10.3.2 IEEE 754 სტანდარტი და კომპლექსური გამრავლება

C99-ში ჩაშენდა `_Complex` ტიპი (`<complex.h>`). მეხსიერებაში $z = a+bi$ ინახება როგორც ორი მიმდევრობითი $f64$ ან $f32$ რიცხვი, ზუსტად ისევე, როგორც Fortran-ში.

თუმცა, C/C++ სტანდარტი ითხოვს IEEE 754-ის მკაცრ დაცვას $\infty \cdot NaN$ შემთხვევებისთვის. კომპლექსური გამრავლებისას $z_1 \cdot z_2$

$$z_1 \cdot z_2 = (ac - bd) + i(ad + bc)$$

შედეგში შეიძლება მივიღოთ NaN ან ∞. C კომპილატორი ვალდებულია შეამოწმოს $0 \cdot \infty$ შემთხვევები (მაგალითად, როცა ნამდვილი ნაწილის მნიშვნელობა NaN-ია, მაგრამ წარმოსახვითი ნაწილი სასრულია). ეს ყოველ გამრავლებაზე ამატებს ლოგიკურ განშტოებებს (branching), რაც გამოთვლებს 2–4-ჯერ ანელებს.

Fortran-ი ნაგულისხმევად იყენებს უბრალო ალგებრულ ფორმულას დამატებითი შემოწმებების გარეშე. C/C++-ში Fortran-ის სიჩქარის მისაღწევად აუცილებელია კომპილატორის დროშები:

- `-O3` — მაქსიმალური ოპტიმიზაცია;
- `-march=native` — პროცესორის ლოკალური SIMD ინსტრუქციების ჩართვა: იყენებს პროცესორის ლოკალურ AVX2/AVX-512 ინსტრუქციებს;
- `-ffast-math` — რთავს სწრაფ მათემატიკურ გამოთვლებს IEEE შემოწმებების გვერდის ავლით (*ეს არაა რეკომენდებული. როცა იტერაციების რიცხვი დიდია, გროვდება ცდომილება*);
- `-fcx-limited-range` — თიშავს კომპლექსურ გამრავლებაზე NaN/∞ სპეციალურ დამუშავებას;

10.3.3 მათემატიკური მატრიცა და მეხსიერების ლეიოუტი: Row-Major vs Column-Major

წრფივ ალგებრაში („ქაღალდზე“ ან წიგნებში) $M \times N$ ზომის A მატრიცა ჩაიწერება ორგანზომილებიანი ცხრილის სახით, სადაც პირველი ინდექსი i აღნიშნავს **სტრიქონს** ($i = 0, \ldots, M-1$), ხოლო მეორე ინდექსი j — **სვეტს** ($j = 0, \ldots, N-1$):

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

რადგან კომპიუტერის ოპერატიული მეხსიერება (RAM) არის **წრფივი (1D) სტრუქტურა**, ეს ორგანზომილებიანი ცხრილი მეხსიერების მისამართებში უწყვეტ 1D ბლოკად უნდა გაიშალოს. ამ გაშლის ორი ფუნდამენტური გზა არსებობს:

1. **Row-Major (სტრიქონების მიხედვით):** მეხსიერებაში გვერდიგვერდ ეწყობა ერთი სტრიქონის ელემენტები. ბოლო ინდექსი (j) იცვლება ყველაზე სწრაფად.
   - მეხსიერების მიმდევრობა: $[a_{0,0}, a_{0,1}, a_{0,2}, \quad a_{1,0}, a_{1,1}, a_{1,2}, \quad a_{2,0}, a_{2,1}, a_{2,2}]$
   - 1D ინდექსის ფორმულა (N სვეტისთვის): index $= i \cdot N + j$
   - **გამოიყენება:** C, C++, Python (NumPy default).
2. **Column-Major (სვეტების მიხედვით):** მეხსიერებაში გვერდიგვერდ ეწყობა ერთი სვეტის ელემენტები. პირველი ინდექსი (i) იცვლება ყველაზე სწრაფად.
   - მეხსიერების მიმდევრობა: $[a_{0,0}, a_{1,0}, a_{2,0}, \quad a_{0,1}, a_{1,1}, a_{2,1}, \quad a_{0,2}, a_{1,2}, a_{2,2}]$
   - 1D ინდექსის ფორმულა (M სტრიქონისთვის): index $= j \cdot M + i$
   - **გამოიყენება:** Fortran, MATLAB, R, OpenGL (GLSL).

როგორ ავაგოთ ორივე ჩალაგება(layout) C-ში და Fortran-ში?

მნიშვნელოვანია გავიაზროთ: **ენის ნაგულისხმევი წესი არ გვიზღუდავს მეორე ტიპის ჩალაგების გამოყენებას.**

- **C / C++-ში:**
  - ჩვეულებრივი ორმაგი მასივი `m[i][j]` ავტომატურად ინახება როგორც **Row-Major**.
  - თუ 1D მასივს `mplane[9]` შევავსებთ `mplane[j * M + i]` ფორმულით, C-ში მივიღებთ **Column-Major** მეხსიერებას.
- **Fortran-ში:**
  - ჩვეულებრივი ორმაგი მასივი `A(i,j)` ავტომატურად ინახება როგორც **Column-Major**.
  - თუ გამოვიყენებთ `A(j,i)` ინდექსაციას ან შესაბამის გარდაქმნას, Fortran-შიც შეგვიძლია სიმულირება გავუკეთოთ **Row-Major** განლაგებას.

კოდი **10.2** c_row_col_comparison

```c
#include <stdio.h>

int main() {
 // მათემატიკური მატრიცა A (3x3)
 // [ 1.0  2.0  3.0 ]
 // [ 4.0  5.0  6.0 ]
 // [ 7.0  8.0  9.0 ]
```

```
8
9   float mplane_row[9]; // Row-Major ჩალაგება C-სთვის
10  float mplane_col[9]; // Column-Major ჩალაგება Fortran/OpenGL-სთვის
11
12  int rows = 3, cols = 3;
13
14  // შევსება მათემატიკური (i=row, j=col) ინდექსაციით
15  for (int i = 0; i < rows; ++i) {
16   for (int j = 0; j < cols; ++j) {
17    float val = (i + 1) * 10.0f + (j + 1); // მაგ. 11, 12, 13...
18
19    // Row-Major: ბოლო ინდექსი j იცვლება სწრაფად
20    mplane_row[i * cols + j] = val;
21
22    // Column-Major: პირველი ინდექსი i იცვლება სწრაფად
23    mplane_col[j * rows + i] = val;
24   }
25  }
26
27  printf("Row-Major mekhsiereba (C naguliskhmevi): ");
28  for (int k = 0; k < 9; ++k) printf("%.0f ", mplane_row[k]);
29  // 11 12 13  21 22 23  31 32 33
30
31  printf("\nColumn-Major მეხსიერება (Fortran naguliskhmevi): ");
32  for (int k = 0; k < 9; ++k) printf("%.0f ", mplane_col[k]);
33  // 11 21 31  12 22 32  13 23 33
34  printf("\n");
35  return 0;
36 }
```

10.3.4 წარმადობა (Performance) და მატრიცების გამრავლება

ორი მატრიცის გამრავლებისას $C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j}$, პროცესორის **L1/L2 ქეშის ეფექტურობა** მთლიანად დამოკიდებულია ციკლების თანმიმდევრობასა და მეხსიერების ლეიოუტზე.

პროცესორს მეხსიერებიდან მონაცემები მოაქვს **Cache Line**-ის ბლოკებით (ჩვეულებრივ 64 ბაიტი, ე.ი. 16 ცალი `float`). თუ შემდეგი ოპერაციისთვის საჭირო რიცხვი უკვე ჩატვირთულ Cache Line-შია, ეს არის **Cache Hit** (სწრაფი). თუ სხვა მისამართზე ნახტომი გვიწევს — **Cache Miss** (ნელი).

1. C / C++ ენა (Row-Major-ისთვის ოპტიმალური ციკლი) თუ C-ში დავწერთ სტანდარტულ $i \to j \to k$ ციკლს:

კოდი 10.3 c_matrix_mult_slow

```
// ნელი ვარიანტი C-ში (i -> j -> k): Cache Miss-ები B მატრიცაზე!
for (size_t i = 0; i < N; ++i) {
 for (size_t j = 0; j < N; ++j) {
  for (size_t k = 0; k < N; ++k) {
   C[i][j] += A[i][k] * B[k][j]; // B[k][j] ითხოვს ნახტომებს სვეტის გასწვ
   ↪ რივ!
  }
 }
}
```

აქ $B[k][j]$ შიდა ციკლში k-ის ცვლილებისას მეხსიერებაში დახტუნავს N ელემენტის ბიჯით (Stride-N access), რაც იწვევს კატასტროფულ **Cache Miss**-ებს.

სწორი (ოპტიმიზებული) მიდგომა C-ში ($i \to k \to j$): თუ შიდა ციკლად გავხდით j-ს, $B[k][j]$ დაიწყებს მეხსიერებაში მიმდევრობით (Stride-1) კითხვას, რაც წარმადობას 5–10-ჯერ ზრდის!

კოდი 10.4 c_matrix_mult_fast

```
// სწრაფი ვარიანტი C-ში (i -> k -> j):Row-Major ქეშის მაქსიმალური გამოყენება
for (size_t i = 0; i < N; ++i) {
 for (size_t k = 0; k < N; ++k) {
  for (size_t j = 0; j < N; ++j) {
   C[i][j] += A[i][k] * B[k][j]; // B[k][j] იკითხება მიმდევრობით!
  }
 }
}
```

2. Fortran ენა (Column-Major-ისთვის ოპტიმალური ციკლი) რადგან Fortran-ში პირველი ინდექსი იცვლება ყველაზე სწრაფად, შიდა ციკლში **პირველი ინდექსი (i) უნდა იცვლებოდეს!**

Fortran-ის ოპტიმალური ციკლების თანმიმდევრობაა $j \to k \to i$:

კოდი 10.5 Fortran-ის ოპტიმალური ციკლების თანმიმდევრობა

```
! სწრაფი ვარიანტი Fortran-ში (j -> k -> i).
!Column-Major ქეშის ოპტიმიზაცია
do j = 1, N
do k = 1, N
do i = 1, N
C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

```
! A(i,k) და C(i,j) იკითხება მიმდევრობით!
end do
end do
end do
```

ცხრილი 10.2 ოპტიმალური ჩალაგება მატრიცაში

ენა	ნაგულისხმევი განლაგება	ოპტიმალური შიდა ინდექსი	საუკეთესო ციკლი
C / C++	Row-Major	ბოლო ინდექსი (j) იცვლება შიდა ციკლში	$i \to k \to j$
Fortran	Column-Major	პირველი ინდექსი (i) იცვლება შიდა ციკლში	$j \to k \to i$

10.3.5 მაგალითი: OpenGL (Column-Major) მატრიცის ჩალაგება, ინიციალიზაცია და დაბეჭდვა C-ში

OpenGL-ში ყველა 4×4 მატრიცა (მაგალითად Model, View, Projection მატრიცები) მეხსიერებაში ინახება **Column-Major** ფორმატში. ეს ნიშნავს, რომ 16-ელემენტიან 1D მასივში პირველი 4 ელემენტი წარმოადგენს **პირველ სვეტს**, შემდეგი 4 — **მეორე სვეტს** და ა.შ.

თუ C-ში გვინდა OpenGL-ის მატრიცასთან მუშაობა, ორი ძირითადი გზა არსებობს:

1. **1D მასივით (OpenGL-ის ბუნებრივი გზა):** მატრიცა ინახება 16-ელემენტიან `float gl_matrix[16]` მასივში. ელემენტზე წვდომა (i-სტრიქონი, j-სვეტი) ხდება ფორმულით: `index = j * 4 + i`.
2. **2D მასივში გარდაქმნა (C Row-Major მატრიცაში ჩალაგება):** თუ C-ის ჩვეულებრივ 4×4 ორმაგ მასივში `c_matrix[row][col]` გვინდა ჩავალაგოთ OpenGL-ის სვეტები, მოგვიწევს ინდექსების ტრანსპონირება: `c_matrix[row][col] = gl_matrix[col * 4 + row]`.

ქვემოთ მოყვანილია C კოდის სრული მაგალითი, სადაც ნაჩვენებია OpenGL წანაცვლების (Translation) მატრიცის ინიციალიზაცია, C-ს 2D მატრიცაში ჩალაგება და ეკრანზე დაბეჭდვა:

კოდი 10.6 c_opengl_matrix_example

```
#include <stdio.h>

int main() {
// 1. OpenGL-ის 4 × 4 Column-Major მატრიცა (1D მასივი RAM-ში)
// მათემატიკურად ეს არის წანაცვლების (Translation) მატრიცა (Tx=10, Ty=20,
  ↪ Tz=30):
//
// |  1   0   0  10 |   <- სვეტი 0: {1, 0, 0, 0}
```

```
// |  0   1   0  20 |   <- სვეტი 1: {0, 1, 0, 0}
// |  0   0   1  30 |   <- სვეტი 2: {0, 0, 1, 0}
// |  0   0   0   1 |   <- სვეტი 3: {10, 20, 30, 1} (Translation ვექტორი)

float gl_matrix[16] = {
 1.0f,  0.0f,  0.0f,  0.0f,  // Column 0
 0.0f,  1.0f,  0.0f,  0.0f,  // Column 1
 0.0f,  0.0f,  1.0f,  0.0f,  // Column 2
 10.0f, 20.0f, 30.0f,  1.0f   // Column 3 (Tx, Ty, Tz)
};

// 2. ჩალაგება C-ს 2D მასივში Row-Major ფორმატში (მათემატიკური ბადე
  ↪ c_matrix[row][col])
float c_matrix[4][4];

for (int col = 0; col < 4; ++col) {
 for (int row = 0; row < 4; ++row) {
  // OpenGL-ის Column-Major ინდექსიდან (col * 4 + row)
  // C-ს Row-Major მატრიცაში გადატანა
  c_matrix[row][col] = gl_matrix[col * 4 + row];
 }
}

// 3. დაბეჭდვა - მეხსიერების რეალური მიმდევრობა (როგორც RAM-შია ჩალაგებულ
  ↪ ი OpenGL-ისთვის)
printf("OpenGL mekhsierebis ganlageba RAM-shi (1D Column-Major):\n");
for (int k = 0; k < 16; ++k) {
 printf("%5.1f ", gl_matrix[k]);
}
printf("\n\n");

// 4. დაბეჭდვა - მათემატიკური 4 × 4 ბადე (როგორც ქაღალდზე ვხედავთ)
printf("matematikuri 4x4 bade C-shi (Row-Major representation):\n");
for (int r = 0; r < 4; ++r) {
 for (int c = 0; c < 4; ++c) {
  printf("%5.1f ", c_matrix[r][c]);
 }
 printf("\n"); // ახალი სტრიქონის გადაყვანა \textbackslash n
}

```

```
46  return 0;
47 }
```

დავალება 10.3: Row-Major ↔ Column-Major

1. **Row-Major ↔ Column-Major ფაილური გარდაქმნა:** დაწერეთ C პროგრამა, რომელიც 4×4 მატრიცას შეინახავს ბინარულ ფაილში `matrix_row.bin` როგორც **Row-Major** (გამოიყენეთ `fwrite`). შემდეგ წაიკითხეთ ეს ფაილი `fread()`-ით და ჩაწერეთ მეორე ბინარულ ფაილში `matrix_col.bin` **Column-Major** ფორმატში ისე, რომ Fortran-ის ან OpenGL-ის პროგრამამ შეძლოს მისი პირდაპირ წაკითხვა.
2. **ტექსტურ ფაილში ფორმატირებული გარდაქმნა `\n` სიმბოლოებით:** დაწერეთ ფუნქცია, რომელიც 3×3 მატრიცას ჩაწერს ტექსტურ ფაილში `matrix3x3.txt`. ყოველი სტრიქონის ბოლოს გამოიყენეთ `\n` ახალი სტრიქონის გადაყვანის სიმბოლო.

10.4 Fortran და C კოდები, ჩალაგება და წარმადობა

კოდი 10.7 C მასივები restrict ოპციით

```
1  #include "modern_c_features.h"
2  //კროსპლატფორმული ტაიმერ ფუნქციები
3  #if defined(_WIN32) || defined(_WIN64)
4  #include <windows.h>
5
6  static double get_time_sec(void) {
7      LARGE_INTEGER frequency, count;
8      QueryPerformanceFrequency(&frequency);
9      QueryPerformanceCounter(&count);
10     return (double)count.QuadPart / (double)frequency.QuadPart;
11 }
12 #else
13 #include <time.h>
14
15 static double get_time_sec(void) {
16     struct timespec ts;
17     clock_gettime(CLOCK_MONOTONIC, &ts);
18     return (double)ts.tv_sec + (double)ts.tv_nsec * 1e-9;
19 }
20 #endif
21
22 /* 1e3 = მილიწამები, 1e6 = მიკროწამები, 1e9 = ნანოწამები */
23 static inline double get_time_mili_sec(void)  { return get_time_sec()
    ↪ * 1e3; }
```

```c
static inline double get_time_micro_sec(void) { return get_time_sec()
    ↪ * 1e6; }
static inline double get_time_nano_sec(void)  { return get_time_sec()
    ↪ * 1e9; }
// Fortran-equivalent high-speed 3D complex multiplication in C
void fast_complex_3d_mult(size_t dim1, size_t dim2, size_t dim3,
                          double complex * restrict A,
                          const double complex * restrict B,
                          const double complex * restrict C) {
    size_t total_elements = dim1 * dim2 * dim3;
    for (size_t i = 0; i < total_elements; ++i) {
        A[i] = B[i] * C[i];
    }
}

void vector3d_add_array(size_t n,
                        Vector3D * restrict res,
                        const Vector3D * restrict a,
                        const Vector3D * restrict b) {
    for (size_t i = 0; i < n; ++i) {
        res[i].x = a[i].x + b[i].x;
        res[i].y = a[i].y + b[i].y;
        res[i].z = a[i].z + b[i].z;
    }
}

double vector3d_length(Vector3D v) {
    return hypot(hypot(v.x, v.y), v.z);
}

void demonstrate_math_features(void) {
    printf("--- Modern math.h & complex.h Features ---\n");

    double a = 2.5, b = 4.0, c = 1.5;
    printf("FMA operation fma(%.1f, %.1f, %.1f) = %.2f\n", a, b, c,
    ↪ fma(a, b, c));
    printf("hypot(3.0, 4.0) = %.2f\n", hypot(3.0, 4.0));
    printf("Gamma function tgamma(5.0) [4!] = %.2f\n", tgamma(5.0));

    double invalid_val = 0.0 / 0.0;
```

```c
    if (isnan(invalid_val)) {
        printf("fpclassify: Value is NaN!\n");
    }

    double complex z1 = 3.0 + 4.0 * I;
    double complex z2 = 1.0 - 2.0 * I;
    double complex z_prod = z1 * z2;

    printf("Complex multiplication: (3+4i)*(1-2i) = %.2f %+.2fi\n",
           creal(z_prod), cimag(z_prod));
    printf("Complex magnitude |z1| = %.2f\n\n", cabs(z1));
}

// ========================================================
//   1. CACHE OPTIMIZATION BENCHMARK (Row-Major vs Column-Major)
// ========================================================
void run_cache_benchmark_3d(size_t dim1, size_t dim2, size_t dim3) {
    size_t total_elements = dim1 * dim2 * dim3;
    size_t bytes = total_elements * sizeof(double);
    printf("=====================================================\n");
    printf("[TEST 1] 3D Cache Benchmark (Allocated: %.2f MB)\n", bytes
    ↪  / (1024.0 * 1024.0));
    printf("=====================================================\n");

    double *matrix = (double *)malloc(bytes);
    if (!matrix) {
        printf("Error: Failed to allocate memory!\n");
        return;
    }

    // 1.1 Cache-Friendly traversal
    double start_time = get_time_sec();
    double sum_good = 0.0;

    for (size_t i = 0; i < dim1; ++i) {
        for (size_t j = 0; j < dim2; ++j) {
            for (size_t k = 0; k < dim3; ++k) {
                size_t idx = i * (dim2 * dim3) + j * dim3 + k;
                matrix[idx] = i + j + k;
                sum_good += matrix[idx];
```

```
            }
        }
    }
    double time_good = get_time_sec() - start_time;
    printf("1. Row-Major (Cache-Friendly / Optimal):   %.4f seconds\n"
    ↪ , time_good);

    // 1.2 Cache-Unfriendly traversal
    start_time = get_time_sec();
    double sum_bad = 0.0;

    for (size_t k = 0; k < dim3; ++k) {
        for (size_t j = 0; j < dim2; ++j) {
            for (size_t i = 0; i < dim1; ++i) {
                size_t idx = i * (dim2 * dim3) + j * dim3 + k;
                sum_bad += matrix[idx];
            }
        }
    }
    double time_bad = get_time_sec() - start_time;
    printf("2. Column-Major (Cache-Misses / Suboptimal): %.4f seconds\
    ↪ n", time_bad);

    // პრევენცია Dead Code Elimination-ის წინააღმდეგ:
    // რადგან ცვლადებს ვბეჭდავთ, კომპილატორი იძულებული გახდება ორივე ციკ
    ↪ ლი ბოლომდე ჩაატაროს!
    if (sum_good == sum_bad) {
        printf("   (Checksum verified: %.0f)\n", sum_good);
    }

    if (time_good > 0) {
        printf("--> Result: Row-Major cache locality is %.2fx faster!\
    ↪ n\n", time_bad / time_good);
    }

    free(matrix);
}

// ========================================================
//   2. RESTRICT & FAST-MATH BENCHMARK (Large Memory ~2.88 GB)
```

```
136 // ====================================================
137 void run_complex_restrict_benchmark(size_t element_count) {
138     size_t bytes_per_array = element_count * sizeof(double complex);
139     double total_gb = (bytes_per_array * 3.0) / (1024.0 * 1024.0 *
    ↪ 1024.0);
140 
141     printf("================================================\n");
142     printf("[TEST 2] Restrict & Fast-Math Complex Multiplication\n");
143     printf("Element Count: %zu (Total Memory: %.2f GB)\n",
    ↪ element_count, total_gb);
144     printf("================================================\n");
145 
146     printf("Allocating memory in RAM...\n");
147     double complex *A = (double complex *)malloc(bytes_per_array);
148     double complex *B = (double complex *)malloc(bytes_per_array);
149     double complex *C = (double complex *)malloc(bytes_per_array);
150 
151     if (!A || !B || !C) {
152         printf("Error: Memory allocation failed for %.2f GB!\n",
    ↪ total_gb);
153         if (A) free(A);
154         if (B) free(B);
155         if (C) free(C);
156         return;
157     }
158 
159     // Initialize arrays
160     for (size_t i = 0; i < element_count; ++i) {
161         B[i] = (i + 1.0) + 2.0i;
162         C[i] = 3.0 - 1.0i;
163     }
164 
165     printf("Executing computation (AVX SIMD Vectorization)...\n");
166     double start_time = get_time_sec();
167 
168     // Call vector function with restrict qualifiers
169     fast_complex_3d_mult(1, 1, element_count, A, B, C);
170 
171     double elapsed_time = get_time_sec() - start_time;
172 
```

```
    // Calculate GFLOPS (6 operations per complex multiplication)
    double gflops = (element_count * 6.0) / (elapsed_time * 1e9);

    printf("Execution Time: %.4f seconds\n", elapsed_time);
    printf("Performance:    %.2f GFLOPS\n", gflops);
    printf("First element:  A[0] = %.2f %+.2fi\n", creal(A[0]), cimag(
    ↪ A[0]));
    printf("--> Note: With -ffast-math & -O3, SIMD vectorization
    ↪ achieves maximum throughput.\n\n");

    free(A);
    free(B);
    free(C);
}

// ========================================================
//   MAIN FUNCTION
// ========================================================
int main(void) {
    bool is_running = true;
    if (is_running) {
        printf("Modern C (C99/C11/C23) Performance & Optimization Test
    ↪  Suite\n\n");
    }

    // 1. Basic feature demonstration
    Vector3D v1 = {1.0, 2.0, 3.0};
    Vector3D v2 = {4.0, 5.0, 6.0};
    Vector3D v_out[1];

    vector3d_add_array(1, v_out, &v1, &v2);
    printf("Vector3D Result: [%.2f, %.2f, %.2f]\n", v_out[0].x, v_out
    ↪ [0].y, v_out[0].z);
    printf("Vector Length via _Generic: %.2f\n\n", vector_len(v1));

    demonstrate_math_features();

    // 2. [TEST 1] Cache Benchmark (500 x 500 x 500 = 125M elements ~ 1 GB
    ↪ RAM)
    run_cache_benchmark_3d(500, 500, 500);
```

```
208
209     // 3. [TEST 2] Restrict + IEEE Fast-Math Benchmark (60M complex numbers ~
        ↪  2.88 GB RAM)
210     run_complex_restrict_benchmark(60000000);
211
212     return 0;
213 }
```

კოდი 10.8 F90 მასივები

```
1  program complex_benchmark
2      implicit none
3
4      ! Double Precision (64-bit real / 128-bit complex) ტიპის განსაზღვრა
5      integer, parameter :: dp = kind(0.0d0)
6
7      integer(8) :: element_count
8      complex(dp), allocatable :: A(:), B(:), C(:)
9
10     integer(8) :: start_count, end_count, count_rate
11     real(dp) :: elapsed_time, gflops, total_ops
12     integer(8) :: i
13
14     ! ელემენტების რაოდენობა (60 მილიონი - ზუსტად C-ს იდენტური)
15     element_count = 60000000_8
16
17     print *, "=================================================="
18     print *, "  Fortran 90 Restrict & SIMD Performance Suite"
19     print *, "=================================================="
20
21     write(*, '(A, I0, A)') " Element Count: ", element_count, " (Total
       ↪ Memory: 2.68 GB)"
22     print *, " Allocating memory in RAM..."
23
24     ! მეხსიერების გამოყოფა დინამიურ მეხსიერებაში (Heap Allocation)
25     allocate(A(element_count))
26     allocate(B(element_count))
27     allocate(C(element_count))
28
29     ! მასივების ინიციალიზაცია (B = 3.0 + 4.0i, C = 1.0 - 2.0i)
```

```
30    do i = 1, element_count
31        B(i) = cmplx(3.0_dp, 4.0_dp, kind=dp)
32        C(i) = cmplx(1.0_dp, -2.0_dp, kind=dp)
33    end do
34
35    print *, " Executing computation (Fortran Vectorized Array Syntax)
      ↪ ..."
36
37    ! მაღალი სიზუსტის ტაიმერის ჩართვა (System Clock)
38    call system_clock(start_count, count_rate)
39
40    ! ------------------------------------------------------
41    ! მთავარი ოპერაცია. Fortran-ის ელემენტარული მასივური გამრავლება
42    ! (კომპილატორი მას ავტომატურად უკეთებს SIMD/AVX ვექტორიზაციას
      ↪ )
43    ! ------------------------------------------------------
44    A = B * C
45
46    ! ტაიმერის გაჩერება
47    call system_clock(end_count, count_rate)
48
49    ! დროის გაანგარიშება წამებში
50    elapsed_time = real(end_count - start_count, dp) / real(count_rate,
      ↪  dp)
51
52    ! GFLOPS ფორმულა (6 ოპერაცია * ელემენტების რაოდენობა / (დრო * 1
      ↪ e9))
53    total_ops = real(element_count, dp) * 6.0_dp
54    gflops = total_ops / (elapsed_time * 1.0e9_dp)
55
56    ! შედეგების დაბეჭდვა
57    print *, "------------------------------------------------------"
58    write(*, '(A, F8.4, A)') " Execution Time: ", elapsed_time, "
      ↪ seconds"
59    write(*, '(A, F8.2, A)') " Performance:    ", gflops, " GFLOPS"
60    write(*, '(A, F5.2, A, F5.2, A)') " First element:  A(1) = ", real(
      ↪ A(1)), " + ", aimag(A(1)), "i"
61    print *, "------------------------------------------------------"
62
63    ! მეხსიერების გათავისუფლება
```

```
64      deallocate(A, B, C)
65
66  end program complex_benchmark
```

10.5 GFLOPS-ის გამოთვლის ფორმულის დეტალური ანალიზი

პროგრამაში გამოყენებული ფორმულა ზომავს პროცესორის წარმადობას **GFLOPS**-ებში (*Giga Floating-Point Operations Per Second* — მილიარდი მცურავწერტილიანი ოპერაცია წამში):

$$\text{GFLOPS} = \frac{\text{element_count} \times 6.0}{\text{elapsed_time} \times 10^9}$$

1. რატომ მრავლდება 6.0-ზე? (კომპლექსური გამრავლება)

ორი კომპლექსური რიცხვის გამრავლებისას გამოიყენება ალგებრის სტანდარტული წესი:

$$(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$$

მცურავწერტილიანი ოპერაციების (*FLOP*) რაოდენობა ითვლება შემდეგნაირად:

ნამდვილი ნაწილის გამოთვლა ($ac - bd$): $a \cdot c$ (გამრავლება), $b \cdot d$ (გამრავლება), $(ac) - (bd)$ (გამოკლება) $\rightarrow$ **სულ 3 ოპერაცია**.

წარმოსახვითი ნაწილის გამოთვლა ($ad + bc$): $a \cdot d$ (გამრავლება), $b \cdot c$ (გამრავლება), $(ad) + (bc)$ (მიმატება) $\rightarrow$ **სულ 3 ოპერაცია**.

დასკვნა: თითოეული 1 კომპლექსური გამრავლება მოითხოვს ზუსტად **6 მცურავწერტილიან ოპერაციას**.

2. რატომ მრავლდება `element_count`-ზე?

თუ მასივში არის $\text{element_count} = 60,000,000$ ელემენტი, ჩატარებული მათემატიკური ოპერაციების ჯამური რაოდენობა შეადგენს:

$$\text{Total FLOPs} = 60,000,000 \times 6.0 = 360,000,000 \text{ ოპერაციას}$$

3. რატომ იყოფა `elapsed_time`-ზე?

ოპერაციების საერთო რაოდენობის გაყოფით შესრულების დროზე (`elapsed_time` წამებში), მიიღება პროცესორის სიჩქარე წამში (**FLOPS**):

$$\text{FLOPS} = \frac{\text{Total FLOPs}}{\text{elapsed_time}}$$

4. რატომ მრავლდება მნიშვნელი 10^9-ზე (`1e9`)?

რადგან $1 \text{ Giga} = 10^9 = 1,000,000,000$ (1 მილიარდი), ჩვეულებრივი **FLOPS** სიდიდის **GigaFLOPS**-ში გადასაყვანად შედეგი იყოფა 10^9-ზე:

$$\text{GFLOPS} = \frac{\text{FLOPS}}{10^9} = \frac{\text{element_count} \times 6.0}{\text{elapsed_time} \times 10^9}$$

რიცხვითი მაგალითი:

თუ 60 მილიონ კომპლექსურ რიცხვს პროცესორი დაამუშავებს 0.35 წამში:

1. სულ ჩატარდა: $60,000,000 \times 6 = 360,000,000$ ოპერაცია.
2. სიჩქარე წამში: $\frac{360,000,000}{0.35} \approx 1,028,571,428$ FLOPS.
3. გადაყვანა GFLOPS-ში: $\frac{1,028,571,428}{1,000,000,000} \approx \mathbf{1.03}$ GFLOPS.

10.6 ახალი მათემატიკური ფუნქციები

C-ში დამატებულია ოპტიმიზებული მათემატიკური ფუნქციები. მათ შორის:

`fma(x, y, z)` — Fused Multiply-Add ($x \cdot y + z$), სრულდება პროცესორის 1 ინსტრუქციით.

`hypot(x, y)` — $\sqrt{x^2 + y^2}$-ის გამოთვლა Overflow/Underflow-სგან დაცვით.

`erf(x)`, `tgamma(x)` — შეცდომის ფუნქცია და გამა ფუნქცია $\Gamma(x)$.

`log2(x)`, `exp2(x)`, `cbrt(x)` — ფუძე 2 ლოგარითმი, ხარისხი და კუბური ფესვი.

`round(x)`, `trunc(x)`, `nearbyint(x)` — დამრგვალების რეჟიმები.

10.7 C დროის ფუნქციები vs C++ `std::chrono`

კოდში ხშირად საჭიროა დროის შუალედის გაზომვა. ამის შესაძლებლობები არის როგორც C-ში, ასევე *C++* ენაში.

სისწრაფის მხრივ (*overhead/performance*) ორივე მიდგომა აბსოლუტურად იდენტურია. C++-ის `std::chrono` წარმოადგენს "*zero-overhead abstraction*"-ს — კომპილაციისას ის სრულად ოპტიმიზდება და ასემბლერში ზუსტად იმავე დაბალი დონის სისტემურ ინსტრუქციებზე გადადის, რასაც C-ს შესაბამისი ფუნქციები.

`std::chrono`-ს არქიტექტურული უპირატესობები:

სისწრაფე: ტოლია. ორივე შემთხვევაში ოპერაციული სისტემის მაღალი სიზუსტის ტაიმერი გამოიძახება.

ტიპების უსაფრთხოება (*Type Safety*): `std::chrono` გამორიცხავს შეცდომებს — კომპილატორი არ მოგვცემს უფლებას მილიწამები და წამები შეცდომით ავურიოთ. C-ში სტრუქტურებიდან (`timespec`, `timeval`) წამებისა და ნანოწამების ხელით გადაყვანა გვჭირდება.

მონოტონურობა: `std::chrono::steady_clock` გარანტიას იძლევა, რომ ტაიმერი არასოდეს წავა უკან (მაგალითად, თუ ოპერაციულმა სისტემამ ინტერნეტით დრო გაასწორა). C-ში ამისთვის POSIX-სპეციფიკური `CLOCK_MONOTONIC` გვჭირდება.

კროს-პლატფორმულობა: `std::chrono` ერთნაირად მუშაობს Linux-ზე, Windows-სა და macOS-ზე. C-ს მაღალი სიზუსტის ტაიმერები OS-ზეა დამოკიდებული (`clock_gettime` Linux-ზე, `QueryPerformanceCounter` Windows-ზე).

პრაქტიკული რეკომენდაციები:

C++ კოდში: უმჯობესია `std::chrono::steady_clock`-ის გამოყენება — ის ერთდროულად არის მაქსიმალურად სწრაფი, ზუსტი და პორტაბელური.

C კოდში: `clock_gettime(CLOCK_MONOTONIC, ...)` (POSIX/Linux-ისთვის) და/ან ჩვენ მიერ დაწერილი (იხ. კოდში ზემოთ) `get_time_sec()`, რომელიც არის კროსპლატფორ-

მული ფუნქცია. ის:

- მალავს პლატფორმულ სირთულეს: შიგნით იძახებს დაბალი დონის სწრაფ ტაიმერს (`clock_gettime(CLOCK_MONOTONIC, ...)` Linux-ზე ან `QueryPerformanceCounter` Windows-ზე).
- ამარტივებს კოდს: აბრუნებს `double` ტიპის წამებს, რისი წყალობითაც დროის სხვაობის გამოთვლა (`end - start`) კოდში ძალიან სუფთად და მარტივად იწერება.
- ინარჩუნებს მაღალ სიზუსტეს: უზრუნველყოფს ნანოწამების/მიკროწამების სიზუსტეს `clock()`-ის დაბალი სიზუსტისგან განსხვავებით.

C პროექტში ბენჩმარკინგისთვის ეს მიდგომა ყველაზე ოპტიმალური და მოსახერხებელია!

გაფრთხილება (`clock()`): ერიდეთ C-ს `clock()` ფუნქციას. მას აქვს დაბალი სიზუსტე და *multi-threaded* კოდში ითვლის ყველა ბირთვის მიერ დახარჯულ ჯამურ CPU დროს და არა რეალურად გასულ დროს (*wall-clock time*).