# დაპროგრამება C++-ში

შედგენილია სახელმძღვანელოს მიხედვით, მაგალითები აღებულია სახელმძღვანელოდან: Bruce Eckel. Thinking in C++

## 1-2 ლექცია

### ობიექტებზე ორიენტირებული დაპროგრამება(OOP), მისი ძირითადი ცნებები და აგების ობიექტები

კომპიუტერები არ არიან მანქანები, რადგან ისინი არიან აზროვნების გაფართოების საშუალებები და არიან მანქანებისაგან განსხვავებული გამოსახვის საშუალება. როგორც შედეგი, ისინი უფრო ჰგვანან ჩვენი აზროვნების ნაწილებს ვიდრე მანქანებს. ისინი წააგავენ გამოსახვის საშუალებებს, როგორიცაა დამწერლობა, მხატვრობა, სკულპტურა, მულტიპლიკაციები და კინოხელოვნება. ობიექტებზე ორიენტირებული დაპროგრამება არის ამ მიმართულების ნაწილი გამოიყენოს კომპიუტერები, როგორც გამოსახვის საშუალება. ამ ლექციაში შემოვიტანთ OOP-ს ძირითად ცნებებს და მიმოვიხილავთ მის მეთოდებს.

**აბსტრაქციის განვითარება.** ყოველი დაპროგრამების ენა ითვალისწინებს აბსტრაქციას. იგი არის არგუმენტირებული იმ პრობლემების სირთულით, რომლებიც უნდა იქნეს გადაწყვეტილი და დამოკიდებულია აბსტრაქციის სახეობასა და ხარისხზე. იმპერატიული ენები წარმოადგენენ ასამბლერის აბსტრაქციას, მაგრამ ამ ენებში ისევ ვრჩებით, რომ ვიაზროვნოთ კომპიუტერის სრუქტურის ჩარჩოებში. C++ ენას გადავყავართ, რომ ვიაზროვნოთ გადასაწყვეტი პრობლემის სივრცეში.

ობიექტებზე ორიენტირებულ დაპროგრამებას გააჩნია შემდეგი ძირითადი მახასიათებლები:

1. ყველაფერი არის ობიექტი;
2. პროგრამა არის ობიექტების ერთობლიობა და ობიექტები შეტყობინების საშუალებით გადასცემენ ერთიმეორეს თუ რა უნდა გაკეთდეს;
3. ყოველ ობიექტს აქვს საკუთარი მეხსიერება, რომელიც შექმნილია სხვა ობიექტებზე დაყრდნობით;
4. ყოველ ობიექტს აქვს რაიმე ტიპი;
5. კონკრეტული ტიპის ყოველ ობიექტს შეუძლია მიიღოს ერთი და იგივე შეტყობინება.

**ობიექტს აქვს ინტერფეისი.** არისტოტელე იყო ალბათ პირველი, რომელმაც შეისწავლა ტიპის ცნება. იგი განიხილავდა თევზებისა და ფრინველების კლასებს. კლასის ყოველ ობიექტს აქვს თვისებები, რომლებიც საერთოა კლასის ყოველი ობიექტისათვის. რადგანაც კლასი აღწერს ობიექტების ერთობლიობას, რომლებსაც აქვთ ერთნაირი მახასიათებლები(მონაცემები) და ქცევები(ფუნქციონირება), კლასი არის რეალურად მონაცემთა ტიპი, როგორც ნამდვილი რიცხვი, რომელიც განსაზღვრავს რიცხვების სიმრავლეს და ქცევებს. განსხვავებაა იმაში, რომ თქვენ განსაზღვრავთ საკუთარ ტიპებს თქვენი ამოცანის საჭიროებისათვის. დაპროგრამების სისტემა ხელს უწყობს ახალი ტიპების შემოტანას და ზრუნავს ტიპების სწორად გამოყენების შემოწმებაზე.

ობიექტებზე ორიენტირებული დაპროგრამება ამარტივებს პრობლემის წარმოდგენას, მაგრამ ისმის საკითხი, თუ როგორ უნდა გააკეთოს ობიექტმა თქვენთვის სასარგებლო სამუშაო. ამისათვის უნდა არსებობდეს საშუალება, რომ მივმართოთ ობიექტს შეასრულოს რაიმე სამუშაო. ყოველ ობიექტს უნდა შეეძლოს რაიმე სამუშაოს შესრულება. მიმართვა ობიექტზე განსაზღვრულია ინტერფეისის საშუალებით და ტიპი არის ის, რაც განსაზღვრავს ინტერფეისს. მარტივ მაგალითს წარმოადგენს ელექტრონული ნათურა:

| ტიპის სახელი<br><br>ინტერფეისი | Light |
|---|---|
| | on()<br>of()<br>brithen()<br>dim() |

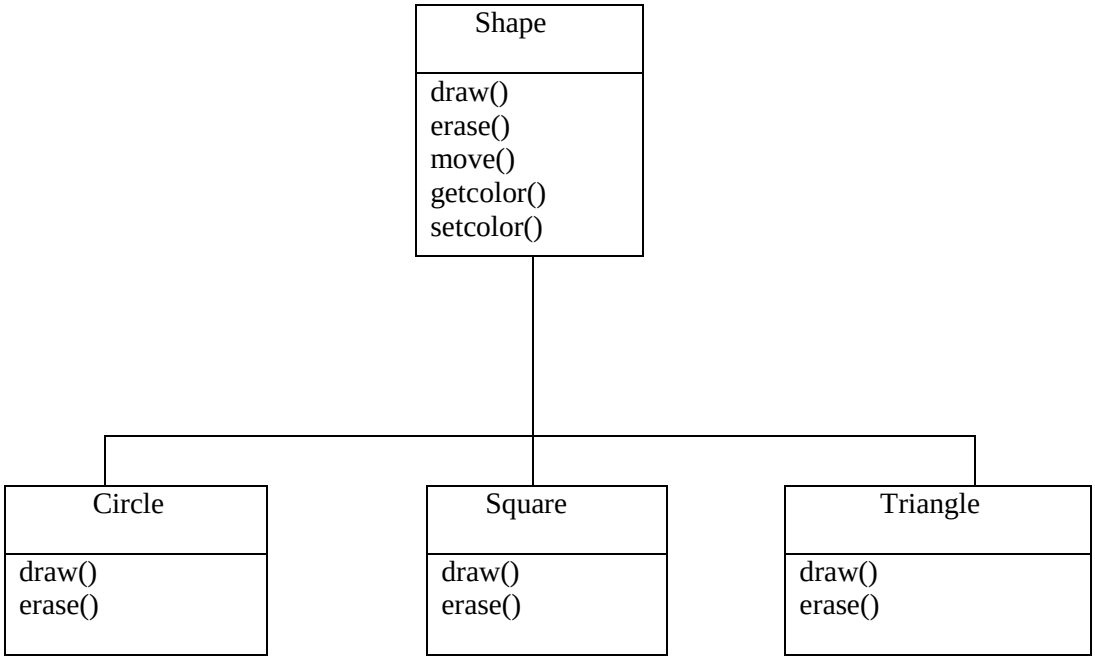
ამ მაგალითში ტიპის სახელია Light და ინტერფეისი შედგება ოთხი ფუნქციისაგან. რომ შევქმნათ ამ ტიპის ობიექტი უნდა დავწეროთ Light.lt; სადაც lt ობიექტის სახელია. ასეთ გრაფიკულ წარმოდგენას ჰქვია მოდელირების ერთიანი ენა UML.

**დაფარული რეალიზაცია.** კლასის გამომყენებლის(კლიენტ პროგრამის) მიზანია მოაგროვოს ყველა კლასები და გამოიყენოს ისინი საკუთარი პრობლემის გადასაწყვეტად, ხოლო კლასის შემქმნელის მიზანია შექმნას კლასი და გამოაჩინოს მხოლოდ ის, რაც აუცილებელია პროგრამისტისთვის. ამიტომ, რაც საჭიროა მხოლოდ კლასის შესაქმნელად და არ სჭირდება პროგრამისტს მიზანშეწონილია დაფარული იყოს, რომ ვინმემ არ დააზიანოს კლასი. ასეთი ელემენტების დაფარვა იცავს პროგრამას შეცდომებისაგან. მეორე მიზეზი არის ის, რომ კლასის შემქმნელს შეეძლოს კლასის გადაკეთება, მაგალითად კლასის ოპტიმიზაციის მიზნით, ისე, რომ არ დააზიანოს კლიენტის პროგრამა.

C++ იყენებს ამისათვის გასაღებ სიტყვებს კლასის საზღვრების დასადგენად. ესენია public, private და protected. public ნიშნავს, რომ მომდევნო განსაზღვრება მისაწვდომია ყველასთვის. private გასაღები სიტყვა ნიშნავს, რომ ამ სიტყვის მომდევნო განსაზღვრებასთან არავის არა აქვს წვდომა ამ კლასის შემქმნელის გარდა. protected გასაღებ სიტყვას აქვს იგივე მნიშვნელობა იმ გამონაკლისით, რომ მის შემდგომ განსაზღვრებაზე წვდომა აქვთ აგრეთვე მემკვიდრე კლასის წევრებს.

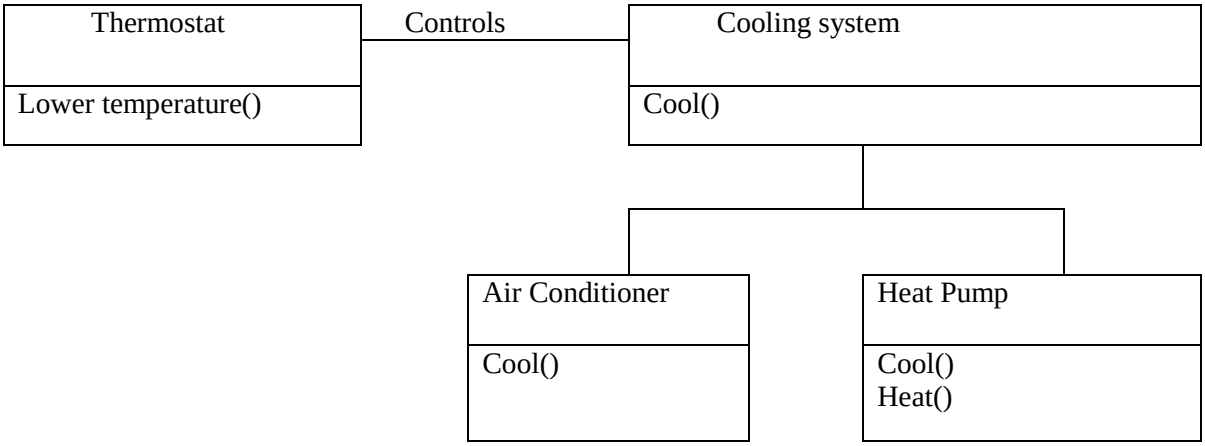
**რეალიზაციის ხელმეორედ გამოყენება.** როცა შევქმნით რაიმე კლასს და გავმართავთ მის მუშაობას, სასურველია შექმნილი კოდი გამოყენებულ იქნეს ახალი კლასის შესაქმნელად. კოდის ხელმეორედ გამოყენება არის ობიექტებზე ორიენტირებული დაპროგრამების ერთ-ერთი უპირატესობა. შესაძლებელია, როგორც მთლიანად შექმნილი კლასის გამოყენება ახალ კლასში, ასევე შექმნილი კლასის ობიექტების გამოყენება ახალ კლასში. ამას ეწოდება წევრი ობიექტის შექმნა. ახალი კლასის შექმნას არსებული კლასების გამოყენებით ჰქვია კომპოზიცია ან უფრო ზოგადად აგრეგაცია. კომპოზიციას მოაქვს პროგრამის დიდი მოქნილობა და გამოყენებული ობიექტები ჩვეულებრივ არიან დაფარული, რომ იყონ მიუწვდომელი კლიენტ პროგრამისტისთვის.

**მემკვიდრეობითობა, როგორც ინტერფეისის ხელმეორედ გამოყენება.** ახალ კლასს შეუძლია გამოიყენოს უკვე შექმნილი კლასის ინტერფეისი, თუ იგი არის შექმნილი კლასის მემკვიდრე. ადრე შექმნილ კლასს ჰქვია წინაპარი კლასი.ასეთი კლასი იყენებს წინაპარი კლასის ინტერფეისს და შეუძლია დაუმატოს ახალი ფუნქციები და მონაცემები. შესაძლებელია ასევე სხვანაირად განვსაზღვროთ უკვე არსებული ფუნქცია. ასეთ ფუნქციას ენიჭება უპირატესობა მოცემულ კლასში წინაპარ ფუნქციასთან შედარებით. მოვიყვანოთ მაგალითი:




ამ მაგალითში კლასების საშუალებით წარმოდგენილია ფიგურა ფორმა(Shape), რომელიც არის წრეწირის, კვადრატისა და სამკუთხედის განზოგადება. Shape(ფორმა) კლასს აქვს ხუთი მეთოდი. ამ კლასის მემკვიდრე კლასებია: Circle(წრეწირი), Square (კვადრატი) და Triangle(სამკუთხედი). მათში მემკვიდრეობითობით განსაზღვრულია Shape-ის მეთოდები, მაგრამ მემკვიდრე კლასებში სხვადასხვანაირად არიან განსაზღვრულნი მეთოდები: draw() და erase().

**ერთნაირი და მსგავსი დამოკიდებულებები.** ისმის საკითხი, მემკვიდრეობითობამ უნდა შეცვალოს მხოლოდ წინაპარი კლასის მეთოდები თუ შეუძლია აგრეთვე დაამატოს მემკვიდრე კლასში ახალი მეთოდები. წინა მაგალითში მხოლოდ შეცვლას ჰქონდა ადგილი. ასეთ შემთხვევაში გვაქვს წმინდა ჩასმა და მას უწოდებენ ხშირად ჩასმის პრინციპს. ჩვენ მას ვუწოდებთ, რომ არის ერთნაირი დამოკიდებულება ძირითად კლასსა და წარმოებულ კლასს შორის. რადგან ჩვენ შეგვიძლია ვთქვათ რომ წრეწირი არის ფორმა. მაგრამ შესაძლებელია ასევე , რომ დავუმატოთ წარმოებულ კლასის ინტერფეისს ახალი ელემენტები და ამით გავაფართოოთ ინტერფეისი და მივიღოთ ახალი ტიპი. ასეთ შემთხვევაში ჩვენ კიდევ შეგვიძლია ძირითადი ტიპი ჩავანაცვლოთ ახალი ტიპით, მაგრამ ეს ჩასმა არ იქნება სრულყოფილი, რადგან ახალი ფუნქციები არ იქნება წვდომადი ძირითადი ტიპიდან. ამ თვალსაზრისით ჩვენ შეგვიძლია ვთქვათ, რომ გვაქვს თითქმის ერთნაირი ტიპები და მას ვუწოდებთ მსგავს დამოკიდებულებებს. განვიხილოთ მაგალითი:




აქ განხილულია სახლის გაგრილების სისტემა. თერმოსტატი არეგულირებს ტემპერატურას. ამისათვის მას აქვს გაცივების სისტემა, რომელიც შედგება ჰაერის კონდიციონერისა და გაცხელების დგუშისაგან. გაციების სისტემასა და მისგან წარმოებულ ტიპებს აქვთ ფუნქცია cool(), რომელიც სხვადასხვანაირადაა განსაზღვრული კლასებში. გაცხელების დგუშს აქვს დამატებით heat() ფუნქცია, რომლის გამოყენება არ შეუძლია გაციების სისტემას. ამიტომ გაციების სისტემასა და გაცხელების დგუშს აქვთ მსგავსი დამოკიდებულება მაშინ, როცა გაციების სისტემასა და ჰაერის კონდიციონერს აქვთ ერთნაირი დამოკიდებულება.

**პოლიმორფიზმით ურთიერთშეცვლადი ობიექტები.** როცა საქმე გვაქვს ტიპების იერარქიასთან ხშირად გვსურს დავამუშაოთ ობიექტი არა როგორც კერძო ტიპის, არამედ როგორც ძირითადი ტიპის. ეს საშუალებას გვაძლევს დავწეროთ კოდი, რომელიც არაა დამოკიდებული კერძო ტიპზე. მაგალითად, ფორმის შემთხვევაში ფუნქციები მოქმედებენ ზოგად ტიპზე, იმის გაუთვალისწინებლად წრეწირია, კვადრატი თუ სამკუთხედი. ყველა ფორმა შეიძლება იყოს დახაზული, წაშლილი ან გადაადგილებული. ეს ფუნქციები უგზავნიან შეტყობინებებს ფორმა ობიექტს და არ ზრუნავენ იმაზე, თუ როგორ გაართმევს თავს ობიექტი ამ შეტყობინებას. ასეთკოდზე არ მოქმედებს ახალი ტიპების დამატება. მაგალითად, ფორმას შეიძლება დავუმატოთ ახალი ტიპი რვაკუთხედი იმ ფუნქციების შეუცვლელად, რომლებიც არიან ფორმაში. ასეთი შესაძლებლობა არის მნიშვნელოვანი, რადგან იგი აუმჯობესებს პროექტირებას და ამცირებს პროგრამული უზრუნველყოფის ღირებულებას. მაგრამ არსებობს პრობლემები, როცა გვინდა დავამუშაოთ წარმოებული ტიპის ობიექტები როგორც ძირითადი ტიპის ობიექტები. როცა ფუნქცია ეუბნება ფორმას დახაზოს იგი კომპილატორმა არ იცის რომელი კოდი(ძირითადი ტიპის თუ წარმოებული ტიპის) გამოიყენოს მის დასახაზავად. როცა გვაქვს არა ობიექტებზე ორიენტირებული დაპროგრამების კომპილატორი, მაშინ იგი აკეთებს სპეციფიკური ფუნქციის სახელის გამოძახების გენერაციას და დამაკავშირებელი პროგრამა(linker) განსაზღვრავს აბსოლიტური მისამართს კოდისა, რომელიც უნდა იქნეს შესრულებული. ამას ჰქვია ადრეული დაკავშირება. ობიექტებზე ორიენტირებული დაპროგრამების კომპილატორის შემთხვევაში, კომპილატორს არ შეუძლია განსაზღვროს შესასრულებელი კოდის მისამართი პროგრამის თვლაზე გაშვებამდე. ამიტომ სხვა სქემა უნდა იყოს გამოყენებული ამ შემთხვევაში. რომ გადაწყდეს ეს პრობლემა. ობიექტებზე ორიენტირებული დაპროგრამების ენები იყენებენ ცნებას, რომელსაც ეწოდება შეყოვნებული დაკავშირება. როცა უგზავნით ობიექტს შეტყობინებას, გამოსაძახებელი კოდი არ განისაზღვრება პროგრამის თვლის პროცესამდე, კომპილატორი ადასტურებს, რომ ასეთი კოდი არსებობს და ასრულებს არგუმენტების ტიპების შემოწმებას და აბრუნებს მნიშვნელობას(ენა, რომლისთვისაც ეს სწორია, ეწოდება სუსტად ტიპირებული ენა), მაგრამ მან არ იცის გამოსათვლელი კოდი. რომ შეასრულოს დაყოვნებული დაკავშირება C++ კომპილატორი აბსოლიტური მისამართის დადგენის ნაცვლად სვამს კოდის სპეციალურ ბიტს. ეს კოდი გამოითვლის ფუნქციის ტანის მისამართს პროგრამის

შესრულების პროცესში ობიექტში შენახული ინფორმაციის საშუალებით. დაწვრილებით ეს იქნება აღწერილი მე-15 ლექციაში. რომ გამოაცხადოთ ფუნქციას ჰქონდეს შეყოვნებული დაკავშირების თვისება, ამისათვის უნდა გამოიყენოთ გასაღები სიტყვა virtual.

**ობიექტების შექმნა და წაშლა.** ძირითადად, OOP-ს არეა აბსტრაქტულ მონაცემთა ტიპირება, მემკვიდრეობითობა და პოლიმორფიზმი, მაგრამ სხვა მხარეებიც შეიძლება იყოს არანაკლებ მნიშვნელოვანი. კერძოდ, მნიშვნელოვანია თუ რა ხერხი გამოიყენება ობიექტების შესაქმნელად და წასაშლელად, რადგან იგი გავლენას ახდენს პროგრამის შესრულების სიჩქარეზე. თუ გამოვიყენებთ ფიქსირებულ ადგილს ობიექტების შესანახად, მაშინ უნდა ვიფიქროთ იმაზე, თუ როგორ გავანთავისუფლოდ ობიექტის მიერ დაკავებული ადგილი, როცა ობიექტი აღარ გვჭირდება და როდის დადგება ასეთი მომენტი. უფრო სწრაფი მეთოდია სტეკების გამოყენება, მაშინ ობიექტებისათვის ადგილის გამოყოფა და მისი გათავისუფლება მოხდება პროგრამის შესრულების დროს.

# მე-7-8 ლექცია

## მონაცემთა აბსტრაქცია

როდესაც თქვენ გადადიხართ ერთი დაპროგრამების ენიდან მეორეზე, ერთ-ერთი მთავარი მიზეზი არის ის, რომ თქვენ დარწმუნებული ხართ, ამით გაზრდით თქვენს პროდუქტიულობას. თუმცა მისი შესწავლის პროცესში თქვენი პროდუქტიულობა დაიკლებს. პროდუქტიულობა ნიშნავს, რომ უფრო ნაკლები რესურსებით უფრო მეტის გაკეთება შეგიძლიათ უფრო ნაკლებ ან ერთიდაიმავე დროში. თუმცა არსებობს სხვა მოტივაციაც, რისთვისაც იცვლით დაპროგრამების ენას, როგორიცაა ეფექტურობა (თქვენი პროგრამა იმუშავებს უფრო სწრაფად უფრო ნაკლები ან იგივე მოცულობის მეხსიერების გამოყენებით, თუმცა ეს არაა ეფექტურობის ზუსტი განმარტება), უსაფრთხოება(თქვენი პროგრამა ყოველთვის გააკეთებს იმას, რაც ჩაფიქრებული იყო ალგორითმით) და პროგრამული მხარდაჭერა(ენას აქვს საშუალებები, რომლებიც გეხმარებათ შექმნათ პროგრამა, რომელიც ადვილად გასაგებია და სწრაფად შეიძლება მისი დაზუსტება ან გაფართოება). პროდუქტიულობის გაზრდის ერთ-ერთი საშუალებაა სხვის მიერ შექმნილი კოდის გამოყენება ბიბლიოთეკების საშუალებით. ბიბლიოთეკა არის ვინმეს მიერ შექმნილი ფუნქციების ერთობლიობა, რომელიც გაერთიანებულია ერთ პაკეტად(ფაილად) და რომლის ფუნქციების ჩადგმა თქვენს პროგრამაში იოლია საჭიროების შემთხვევაში. ხშირად ასეთი ფაილი არის lib გაფართოებით და თან ახლავს header ფაილები, რომლებიც ატყობინებენ კომპილატორს, თუ რა არის მოთავსებული პაკეტში. ჩამტვირთველმა პროგრამამ იცის, თუ როგორ იპოვოს კონკრეტული კოდი ბიბლიოთეკაში და როგორ ჩართოს იგი ტრანსლირებულ კოდში. არსებობს ბიბლიოთეკის შექმნის მეორე გზაც. როცა ბიბლიოთეკაში ჩართულია დაპროგრამების ენაზე დაწერილი ფუნქციები(ამას ჰქვია საწყისი კოდი). ასეთ შემთხვევაში ბიბლიოთეკის ერთი პლატფორმიდან(მაგალითად, windows ან unix) მეორეზე გადატანის შესაძლებლობა არსებობს, ან ჩაისვას თქვენ საწყის კოდში ასეთი ფუნქციები და შემდეგ მოხდეს მიღებული კოდის კომპილაცია მეორე პლატფორმის კონკრეტულ არქიტექტურაზე(რა თქმა უნდა ასეთ შემთხვევაში უნდა არსებობდეს სათანადო კომპილატორი). ამგვარად,

ბიბლიოთეკები არიან პროდუქტიულობის გაზრდის ერთ-ერთი მნიშვნელოვანი ხერხი და C++-ის შექმნის ძირითადი მოტივაციაა, უფრო იოლი გავხადოთ ბიბლიოთეკების გამოყენება.

**C ბიბლიოთეკის მსგავსი პატარა ბიბლიოთეკა.** ბიბლიოთეკა ძირითადად შედგება ფუნქციებისაგან, მაგრამ არსებობს მონაცემები, რომლებიც საჭიროა ჩართულ იქნეს ბიბლიოთეკაში. C ენის შემთხვევაში მიზანშეწონილია გამოყენებულ იქნეს ამისათვის სტრუქტურები. რადგან, ისინი გვაძლევენ საშუალებას გავაერთიანოთ მონაცემები ერთ ობიექტად. შემდეგ შევქმნათ ამ სტრუქტურის ცვლადები. ამგვარად, C ბიბლიოთეკების უმეტესობას აქვთ სტრუქტურების რაიმე სიმრავლე და ფუნქციები, რომლებიც მოქმედებენ ამ სტრუქტურებზე. თუ როგორ გამოიყურება ასეთი სისტემა, განვიხილოთ დაპროგრამების საშუალება, რომელიც მოქმედებს მასივის მსგავსად და ამ მასივის სიგრძე შეიძლება დადგენილ იქნეს პროგრამის შესრულების პროცესში, როცა ეს მასივი იქმნება. ასეთ სისტემას ვუწოდოთ CStash:

```
//: C04:CLib.h
// Header ფაილი c-ს მსგავსი ბიბლიოთეკისათვის
// თვლის დროს შექმნილი მასივის მსგავსი არსება
typedef struct CStashTag {
int size; // თითოეული არეს სიგრძე
int quantity; // მეხსიერების არეების რაოდენობა
int next; // შემდეგი ცარიელი არე
// დინამიკურად განაწილებული ბაიტების მასივი:
unsigned char* storage;
} CStash;
void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
///:~
```

მონიშნული სახელი CStashTag გამოიყენება იმ შემთხვევაში, როცა თქვენ გჭირდებათ სტრუქტურის შიგნიდან მიმართოთ ამავე სტრუქტურას. აქ არის ზოგადად ტიპის განსაზღვრა, რომელიც თქვენ დაგჭირდებათ C ბიბლიოთეკის ყოველი სტრუქტურისათვის. ეს არის გაკეთებული ისე, რომ თქვენ დაამუშაოთ ყოველი struct თითქოს ის არის ახალი ტიპი და შემოიტანოთ ამ ტიპის ცვლადები ასე:

```
CStash A, B, C;
```

მეხსიერების ადგილზე(srorage) მიმთითებელი არის unsigned char*. unsigned char არის storage-ის უმცირესი ელემენტი და დამოკიდებულია რეალიზაციაზე. შეიძლება ვიგულისხმოთ ერთი ბაიტი. კონკრეტული ფაილის საწყისი კოდი გამოიყურება ასე:

```
//: C04:CLib.cpp {O}
```

```
// c-ს მსგავსი ბიბლიოთეკის მაგალითის რეალიზაცია
// სტრუქტურისა ფუნქციების გამოცხადება
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// დასამატებელი ელემენტების რაოდენობა
// როცა ვზრდით მეხსიერებას
const int increment = 100;
void initialize(CStash* s, int sz) {
s->size = sz;
s->quantity = 0;
s->storage = 0;
s->next = 0;
}
int add(CStash* s, const void* element) {
if(s->next >= s->quantity) //დარჩენილია საკმარისი არე?
inflate(s, increment);
// ელემენტის მეხსიერებაში კოპირება,
// დაწყებული შემდეგი ცარიელი ადგილიდან:
int startBytes = s->next * s->size;
unsigned char* e = (unsigned char*)element;
for(int i = 0; i < s->size; i++)
s->storage[startBytes + i] = e[i];
s->next++;
return(s->next - 1); // ინდექსის ნომერი
}
void* fetch(CStash* s, int index) {
// ინდექსის საზღვრების შემოწმება
assert(0 <= index);
if(index >= s->next)
return 0; // უთითებს დამთავრებას
// ღებულობს სასურველი ელემენტის მიმთითებელს:
return &(s->storage[index * s->size]);
}
int count(CStash* s) {
return s->next; // ელემენტები CStash-ში
}
void inflate(CStash* s, int increase) {
assert(increase > 0);
int newQuantity = s->quantity + increase;
int newBytes = newQuantity * s->size;
int oldBytes = s->quantity * s->size;
unsigned char* b = new unsigned char[newBytes];
for(int i = 0; i < oldBytes; i++)
b[i] = s->storage[i]; // ძველის ახალში კოპირება
delete [](s->storage); // ძველი მეხსიერება
s->storage = b; // ახალ მეხსიერებაზე მითითება
s->quantity = newQuantity;
```

```
}
void cleanup(CStash* s) {
if(s->storage != 0) {
cout << "freeing storage" << endl;
delete []s->storage;
}
} ///:~
```

initialize() ასრულებს struct CStash-ის აუცილებელ საწყის დაყენებას – შიგა ცვლადებს აყენებს კონკრეტულ მნიშვნელობებზე. დასაწყისში storage-ზე მიმთითებელი დაყენებულია ნულზე(საწყისი მეხსიერება არაა გამოყოფილი). add() ფუნქცია უმატებს ელემენტს CStash-ში მომდევნო ადგილზე. პირველად იგი ამოწმებს არის თუ არა რაიმე ადგილი დარჩენილი, თუ არაა, მაშინ აფართოებს storage-ს inflate() ფუნქციის გამოყენებით. რადგან კომპილატორმა არ იცის ტიპი შესანახი ცვლადის, რომელიც უნდა იყოს შენახული storage-ში. ამიტომ ყველა ფუნქციის ტიპი არის void*. რადგან არ შეგვიძლია ცვლადზე მინიჭება, ამიტომ ცვლადი მეხსიერებაში(storage) უნდა იყოს კოპირებული თითო-თითო ბაიტით. ამის გაკეთების ყველაზე მოხერხებული გზაა მასივის ინდექსირება. ფაქტიურად next უთითებს მონაცემის ბაიტებს storage-ში. იგი უნდა გაიზარდოს ისე, რომ უჩვენებდეს შემდეგ შესაძლებელ ადგილს storage-ში. შემდეგ ინდექსის ნომერი, სადაც მნიშვნელობა იყო შენახული იცვლება ისე, რომ fetch() ფუნქციამ ამ ინდექსის ნომრით იპოვოს storage-ში შენახული მნიშვნელობა.

fetch() ამოწმებს, ხომ არაა გასული ინდექსი storage-ის საზღვრებს გარეთ და აბრუნებს სასურველი ცვლადის მისამართს, რომელიც გამოითვლება index არგუმენტის გამოყენებით. რადგანაც index უჩვენებს ელემენტების რაოდენობას, რომ გავაკეთოთ ძვრა CStash-ში, ამიტომ იგი უნდა გამრავლდეს ყოველი ნაწილის ბაიტების რაოდენობაზე და მივიღებთ ძვრის სიგრძეს ბაიტებში. მაგრამ ეს არ იძლევა მისამართს storage-ში. რომ მივიღოთ მისამართი უნდა გამოვიყენოთ & ოპერატორი.

**მეხსიერების დინამიკური განაწილება.** რადგან არ ვიცით მეხსიერების მაქსიმალური რაოდენობა, რომელიც შეიძლება დაგვჭირდეს CStesh-ისთვის, ამიტომ CStesh-ისთვის საჭირო მეხსიერება გამოიყოფა heap-ში. heap არის მეხსიერება, რომელსაც იმარაგებს კომპილატორი, რომ გამოჰყოს ადგილი ისეთი სიდიდეებისათვის, რომელთა სიგრძე არაა ცნობილი პროგრამის დაწერის მომენტში. heap-ის მაქსიმალური სიგრძე დაფიქსირებულია კონკრეტული რეალიზაციისათვის. მეხსიერების დინამიკური განაწილებისათვის სტანდარტულ C-ს აქვს ფუნქციები malloc(), calloc(), realloc() და free(). C++ აქვს ამ საკითხის უფრო მახვილგონიერული გადაწყვეტა. ამისათვის იყენებს C++ ენაში ჩართულ საშუალებებს გასაღები სიტყვების new და delete-ს გამოყენებით.

inflate() ფუნქცია იყენებს new-ს, რომ მიიღოს მეხსიერების მონაკვეთი heap-იდან CStash-ისთვის. new-ს დროს ჩვენ შეგვიძლია მეხსიერების მხოლდ გაზრდა და არა შემცირება. ამიტომ assert() ამოწმებს, რომ უარყოფითი რიცხვი არ გადაეცეს inflate() ფუნქციას მეხსიერების გასაზრდელად. newQuantity-ში გვექნება მეხსიერების რაოდენობა და იგი მრავლდება ერთი ელემენტისათვის საჭირო ბაიტების რაოდენობაზე, რომ მივიღოთ newBytes.

ასე, რომ ჩვენ ვიცით, რამდენი ბაიტის კოპირება მოვახდინოთ ძველი ადგილიდან. ასევე oldBytes გამოითვლება ძველი quantity-ის გამოყენებით. აქტუალური მეხსიერების განაწილება იქნება new-expression, რომელიც შეიცავს გასაღებ სიტყვას new:

new unsigned* char[newBytes];

new-ს ზოგადი ფორმაა new Type;
Type აღნიშნავს იმ ცვლადის ტიპს, რომლისთვისაც გვსურს ადგილი გამოვყოთ heap-ში. new-expression აბრუნებს მიმთითებელს ობიექტზე, რომლისთვისაც გვსურს ადგილის გამოყოფა. როდესაც წერთ new Type თქვენ გიბრუნდებათ მიმთითებელი რაიმე ტიპზე. თუ თქვენ წერთ new unsigned char array თქვენ გიბრუნდებათ მიმთითებელი მასივის დასაწყისზე. როცა ახალი მეხსიერება განაწილებულია, ძველი ადგილიდან მონაცემების გადმოწერა ხდება მასივის ინდექსირებით, ციკლის საშუალებით. როცა მონაცემები გადმოიწერება, ძველი ადგილის განთავისუფლება ხდება delete-ს საშუალებით. თუ თქვენ დაგავიწყდათ delete-ს გამოყენება, ამან შეიძლება გამოიწვიოს heape-ს გადავსება. მასივის ამოსაგდებათ გამოიყენება სპეციალური სინტაქსი:

delete [ ] myarray;

ძველი მეხსიერების განთავისუფლების შემდეგ, storage-ს მიმთითებელს მიენიჭება ახალი მიმთითებლის მნიშვნელობა, გამოითვლება მეხსიერების სიგრძე და ამით inflate() ამთავრებს მუშაობას. რადგან მეხსიერების ავტომატური განთავისუფლება არ ხდება, ამისათვის გამოიყენება cleanup() ფუნქცია. შემდეგ მაგალითში, რომ შევამოწმოთ ბიბლიოთეკის მუშაობა ორი CStash არის შექმნილი. პირველი შეიცავს მთელ რიცხვებს და მეორე მასივებს სიგრძით 80 char.

```
//: C04:CLibTest.cpp
//{L} CLib
// ჩ-ს მსგავსი ბიბლიოთეკის ტესტირება
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;
int main() {
// განვსაზღვროთ ცვლადები ბლოკის დასაწყისში
// ისევე როგორც C-შია:
CStash intStash, stringStash;
int i;
char* cp;
ifstream in;
string line;
```

```
const int bufsize = 80;
// მოვახდინოთ ცვლადების ინიციალიზაცია:
initialize(&intStash, sizeof(int));
for(i = 0; i < 100; i++)
add(&intStash, &i);
for(i = 0; i < count(&intStash); i++)
cout << "fetch(&intStash, " << i << ") = "
<< *(int*)fetch(&intStash, i)
<< endl;
// 80 ასოიანი სტრიქონების აღება:
initialize(&stringStash, sizeof(char)*bufsize);
in.open("CLibTest.cpp");
assert(in);
while(getline(in, line))
add(&stringStash, line.c_str());
i = 0;
while((cp = (char*)fetch(&stringStash,i++))!=0)
cout << "fetch(&stringStash, " << i << ") = "
<< cp << endl;
cleanup(&intStash);
cleanup(&stringStash);
} ///:~
```

როგორც ამას C ენა მოითხოვს, ცვლადები შექმნილია main() დასწყისში. რა თქმა უნდა CStash ცვლადების ინიციალიზაცია უნდა მოხდეს შემდეგ initialize() ფუნქციის გამოძახებით. intStash არის შევსებული მთელი რიცხვებით და stringStash char ტიპის მასივებით. ეს მასივები მიიღება საწყისი მონაცემების ფაილების გახსნით და სტრიქონების წაკითხვით line-ში. შემდეგ ვქმნით მიმთითებელს line-ს char წარმოდგენაზე c_str() წევრი ფუნქციის გამოყენებით. CStash-ის ჩატვირთვის შემდეგ იგი გამოიტანება. intStash გამოიტანება for ციკლით, რომელიც იყენებს count()-ს, რომ დაადგინოს მისი საზღვარი. stringStash გამოიტანება while ციკლით, რომელიც წყვეტს მუშაობას, როცა fetch() დააბრუნებს ნულს, რაც იმას ნიშნავს, რომ გამოვედით საზღვრების გარეთ.

**ძირითადი ობიექტი.** შემდეგი წინგადადგმული ნაბიჯია, რომ C++-ის ფუნქციები შეიძლება იყოს ჩართული სტრუქტურაში, როგორც წევრი ფუნქციები. განვიხილოთ მაგალითი, თუ როგორ გარდავქმნათ CStash-ის C ვერსია C++-ის Stash-ში:

```
//: C04:CppLib.h
// C-ს მსგავსი ბიბლიოთეკის გარდაქმნა C++ში
struct Stash {
int size; // თითოეული არის სიგრძე
int quantity; // მეხსიერების არეების რაოდენობა
int next; // შემდეგი ცარიელი არე
// დინამიკურად განაწილებული ბაიტების მასივი:
unsigned char* storage;
```

```
// ფუნქციები!
void initialize(int size);
void cleanup();
int add(const void* element);
void* fetch(int index);
int count();
void inflate(int increase);
}; ///:~
```

პირველ რიგში შევნიშნოთ, რომ არ გვაქვს typedef. C++ კომპილატორი სტრუქტურის სახელს თვითონ გარდაქმნის ახალ ტიპად. ყველა წევრი მონაცემები არიან იგივე, მაგრამ ახლა ფუნქციები არიან struct ტანის შიგნით. ასევე, ბიბლიოთეკის C ვერსიის პირველი არგუმენტი ამოგდებულია.
C++-ში ფუნქციის პირველ არგუმენტად სტრუქტურის მისამართის გადაცემას აკეთებს თვითონ კომპილატორი. ფუნქციის კოდი არის იგივე, რაც იყო ბიბლიოთეკის C ვერსიაში. ყოველი ფუნქციისათვის არსებობს მხოლოდ ფუნქციის ერთი ტანი. ე.ი. როცა გვაქვს

```
Stash A, B, C;
```

ეს არ ნიშნავს, რომ თქვენ მიიღებთ განსხვავებულ add() ფუნქციას თვითეული ფუნქციისათვის, მაგრამ როცა განვსაზღვრავთ ფუნქციას უნდა მივუთითოთ, რომ იგი ეკუთვნის struct Stash-ს და არა ნებისმიერ სხვა struct-ს. ამისათვის ვიყენებთ ფუნქციის საზღვრის განმსაზღვრელ ოპერატორს ::. შემდეგ მაგალითში ნაჩვენებია, თუ როგორ გამოიყენება იგი:

```
//: C04:CppLib.cpp {O}
// C ბიბლიოთეკის C++-ში გადაყვანა
// სტრუქტურისა და ფუნქციების გამოცხადება:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// დასამატებელი ელემენტების რაოდენობა
// როცა ვზრდით მეხსიერებას:
const int increment = 100;
void Stash::initialize(int sz) {
size = sz;
quantity = 0;
storage = 0;
next = 0;
}
int Stash::add(const void* element) {
if(next >= quantity) // არის საკმარისი არე?
inflate(increment);
// ელემენტის მეხსიერებაში კოპირება,
// ახალი ცარიელი ადგილიდან დაწყება:
int startBytes = next * size;
unsigned char* e = (unsigned char*)element;
```

```
for(int i = 0; i < size; i++)
storage[startBytes + i] = e[i];
next++;
return(next - 1); // ინდექსის ნომერი
}
void* Stash::fetch(int index) {
// ამოწმებს ინდექსის საზღვრებს:
assert(0 <= index);
if(index >= next)
return 0; // მიგვანიშნებს ბოლოზე
// სასურველი ელემენტის მიმთითებლის მიღება:
return &(storage[index * size]);
}
int Stash::count() {
return next; // ელემენტების რაოდენობა CStash-ში
}
void Stash::inflate(int increase) {
assert(increase > 0);
int newQuantity = quantity + increase;
int newBytes = newQuantity * size;
int oldBytes = quantity * size;
unsigned char* b = new unsigned char[newBytes];
for(int i = 0; i < oldBytes; i++)
b[i] = storage[i]; // ძველის ახალში კოპირება
delete []storage; // ძველი მეხსიერება
storage = b; // უთითებს ახალ მეხსიერებას
quantity = newQuantity;
}
void Stash::cleanup() {
if(storage != 0) {
cout << "freeing storage" << endl;
delete []storage;
}
} ///:~
```

არსებობს სხვა რამეებიც, რომლებიც განსხვავებულია C-სა და C++-ში. პირველ რიგში გამოცხადება მოითხოვება header ფაილში. ნაცვლად s -> size = sz; ჩვენ ვწერთ size = sz; ვაჩვენოთ კიდევ ერთი განსხვავება მაგალითით:

```
int i=10;
void* vp=&i; // დასაშვებია როგორც C-ში, ასევე C++-ში
int* ip=vp; // დასაშვებია მხოლოდ C-ში
```

თქვენ შეგიძლიათ შეამჩნიოთ ახლი რამეები Stash-ის C++ ვერსიაში შემდეგ სატესტო პროგრამაში:

```
//: C04:CppLibTest.cpp
//{L} CppLib
```

```
// C++ ბიბლიოთეკის ტესტირება
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main() {
Stash intStash;
intStash.initialize(sizeof(int));
for(int i = 0; i < 100; i++)
intStash.add(&i);
for(int j = 0; j < intStash.count(); j++)
cout << "intStash.fetch(" << j << ") = "
<< *(int*)intStash.fetch(j)
<< endl;
// შეიცავს 80 ასოიან სტრიქონს:
Stash stringStash;
const int bufsize = 80;
stringStash.initialize(sizeof(char) * bufsize);
ifstream in("CppLibTest.cpp");
assure(in, "CppLibTest.cpp");
string line;
while(getline(in, line))
stringStash.add(line.c_str());
int k = 0;
char* cp;
while((cp =(char*)stringStash.fetch(k++)) != 0)
cout << "stringStash.fetch(" << k << ") = "
<< cp << endl;
intStash.cleanup();
stringStash.cleanup();
} ///:~
```

პირველ რიგში, ცვლადები არ არიან განსაზღვრულნი პროგრამის თავში, არამედ მათი გამოყენების წინ. წევრ ფუნქციებზე მითითება ხდება .-ის გამოყენებით, ვიყენებთ ახალ header ფაილს require.h.

**რა არის ობიექტი.** ფუნქციების დამატება სტრუქტურებისათვის არის ის ძირითადი მოქმედება, რასაც C++ უმატებს C-ს და ამით შემოაქვს სტრუქტურის ახალი წარმოდგენა. C-ში სტრუქტურა არის მონაცემების ერთობლიობა(პაკეტი), რომელიც საშუალებას გაძლევთ აიღოთ იგი, როგორც ერთეული. ფუნქციები, რომლებიც იყენებენ მას, შეიძლება იყოს მოთავსებული პროგრამაში ნებისმიერ ადგილას. მაგრამ, ფუნქციების მოთავსებით სტრუქტურაში, ვღებულობთ ახალ ქმნილებას, რომელსაც შეუძლია აღწეროს როგორც მახასიათებლები(რასაც C სტრუქტურაც აკეთებს), ასევე ქცევები. ობიექტები არიან ასეთი ქმნილების ეგზემპლარები. C++-ში ობიექტი არის ცვლადი და მისი განმარტება არის – მეხსიერების არე

და მას აქვს უნიკალური სახელი. იგი არის ადგილი, სადაც თქვენ ინახავთ მონაცემებს და არსებობენ აგრეთვე ოპერაციები, რომლებსაც იყენებთ ამ მონაცემებზე. არსებობს შეუთანხმებლობა ობიექტებზე ორიენტირებული ენის ცნების გაგებაში. არსებობენ ობიექტებზე ორიენტირებული ენები,როგორიცაა C++, რომლებიც სტრუქტურებში ათავსებენ მონაცემებსა და ფუნქციებს და არსებობენ ენები, რომლებიც განიხილავენ ფუნქციებს ცალკე და სტრუქტურებში ათავსებენ მხოლოდ მონაცემებს.ასეთ ენებს ვუწოდებთ ობიექტებზე დაფუძნებულ ენებს განსხვავებით ობიექტებზე ორინტირებულ ენებისაგან.

**აბსტრაქტულ მონაცემთა ტიპირება.** მონაცემებისა და ფუნქციების ერთად პაკეტირება საშუალებას გვაძლევს შევქმნათ ახალი მონაცემთა ტიპი. ამას ხშირად უწოდებენ ინკაფსულაციას. რაიმე მონაცემთა ტიპს შეიძლება ჰქონდეს ერთად პაკეტირებული მონაცემთა სხვადასხვა ნაწილები. მაგალითად, როგორიცაა float. მას აქვს ექსპონენტი, მანტისა და ნიშნის ბიტი. ასეთ ტიპს შეგიძლიათ უთხრათ შეასრულოს რაიმე. დაუმატოს მას მეორე float რიცხვი ან მთელი რიცხვი და ა. შ.. ე.ი. მას აქვს მახასიათებლები და ქცევა. Stash-ის განსაზღვრება ჰქმნის ახალ მონაცემთა ტიპს. თქვენ შეგიძლიათ add(), fetch() და inflate() შეასრულოთ მასზე. იგი იქცევა, როგორც ჩვეულებრივი ჩადგმული ტიპი. ჩვენ მას მოვიხსენიებთ, როგორც აბსტრაქტულ მონაცემთა ტიპს, შეიძლება იმიტომ, რომ იგი ჩვენ საშუალებას გვაძლევს მოვახდინოთ რაიმე ცნების აბსტრაქცია პრობლემური არედან პრობლემის გადაწყვეტის არეში. როცა თქვენ იძახებთ Stash-ის რაიმე ფუნქციას კომპილატორი უზრუნველყოფს Stash-ზე ამ ფუნქციის შესრულებას და ამოწმებს ტიპების გამოყენების სისწორეს. ე.ი. კომპილატორი მოქმედებს Stash-ზე, როგორც ჩადგმულ ტიპებზე და ამ მხრივ კომპილატორი არ ანსხვავებს მას ჩადგმული ტიპისაგან. მაგრამ არსებობს სხვა განსხვავებაც. ეს არის ხერხი თუ როგორ ვასრულებთ ოპერაციებს ობიექტებზე. თქვენ წერთ ობიექტი.წევრი-ფუნქცია(არგუმენტების-სია). ეს არის წევრი ფუნქციის გამოძახება ობიექტზე. მეორე მხრივ, ობიექტებზე ორიენტირებული ენის ტერმინებში, ეს ნიშნავს გაუგზავნო შეტყობინება ობიექტს. მაგალითად, Stash.s-სთვის ინსტრუქცია s.add(&i) უგზავნის შეტყობინებას s-ს შეასრულოს add() ფუნქცია თავისთავზე. რეალურად, რასაც თქვენ აკეთებთ – ჰქმნით ობიექტებს და უგზავნით შეტყობინებებს მათ. მთავარი არის ვიცოდეთ, რა ობიექტთან გვაქვს საქმე და რა შეტყობინებას ვუგზავნით მას.

**ობიექტის დეტალები**. ობიექტი იკავებს დაახლოებით იგივე ადგილს, რასაც იკავებს შესატყვისი სტრუქტურა. ამიტომ, ობიექტის მიერ დაკავებული ადგილის სიგრძე შეიძლება გამოვთვალოთ sizeof ოპერატორით. განვიხილოთ მაგალითი:

```
//: C04:Sizeof.cpp
// სტრუქტურის სიგრძე
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;
struct A {
int i[100];
};
```

```
struct B {
void f();
};
void B::f() {}
int main() {
cout << "sizeof struct A = " << sizeof(A)
<< " bytes" << endl;
cout << "sizeof struct B = " << sizeof(B)
<< " bytes" << endl;
cout << "sizeof CStash in C = "
<< sizeof(CStash) << " bytes" << endl;
cout << "sizeof Stash in C++ = "
<< sizeof(Stash) << " bytes" << endl;
} ///:~
```

ამ მაგალითში მოცემულია ორი სტრუქტურა A და B. A შეიცავს მთელი ტიპის 100 ელემენტიან მასივს, სახელით i და B შეიცავს მხოლოდ void f() ფუნქციას, რომლის ტანი ცარიელია. main() ფუნქციას გამოაქვს მათ მიერ დაკავებული ადგილების სიგრძეები ბაიტებში. ასევე გამოაქვს CStash-ისა და Stash-ის მიერ დაკავებული ადგილების სიგრძეები.

**header ფაილის წესები.** როგორც უკვე ვიცით, C-ში სტრუქტურაში ვათავსებთ მონაცემებს და მათზე სამოქმედო ფუნქციები შეიძლება მოთავსებული იყოს ბიბლიოთეკაში ან ობიექტურ მოდულში და ვაცხადებთ ფუნქციებს header ფაილში. მაგრამ, როცა C++-ში შემოგვაქვს აბსტრაქტულ მონაცემთა ტიპი, ვაცხადებთ მონაცემებს სტრუქტურაში და მათზე სამოქმედო ფუნქციების გამოცხადებები არ შეიძლება მოთავსებული იყოს სხვაგან. აგრეთვე, ზოგადი წესი არის, გამოცხადების გარეშე არ შეიძლება ფუნქციის განსაზღვრა და გამოცხადება წინ უნდა უსწრებდეს განსაზღვრას. ბიბლიოთეკის გამოყენება სასურველი იყო C-ში, მაგრამ C++-ში იგი მიღებულია როგორც წესი.

**header ფაილის საჭიროება.** თქვენ შეგიძლიათ არ გამოიყენოთ header ფაილი და გამოაცხადოთ ფუნქცია ხელით თქვენს პროგრამაში. მაგრამ ასეთ შემთხვევაში, თქვენ შეიძლება დაუშვათ შეცდომა მის გამოცხადებაში. კომპილატორი ეყრდნობა თქვენს გამოცხადებას. შეცდომა კი თავს იჩენს პროგრამის შესრულების პროცესში. მაგრამ, თუ თქვენ ფუნქციის გამოცხადებას მოათავსებთ header ფაილში, და მას ჩართავთ ყველგან, სადაც იყენებთ ფუნქციას, მაშინ კომპილატორი ამოწმებს მისი გამოყენების სისწორეს ყველგან მთელ სისტემაში.

**წინასწარდამამუშავებელის(preprocessor) დირექტივები.** #define დირექტივა გამოიყენება, რომ შევქმნათ კომპილაციის დროის ჭდეები. თქვენ გაქვთ ორი არჩევანი: თქვენ შეგიძლიათ უთხრათ წინასწარდამამუშავებელს, რომ ჭდე არის განსაზღვრული მისი მნიშვნელობის განსაზღვრის გარეშე #define Flag დირექტივით ან მას მისცეთ მნიშვნელობა #define Pi 3.14159 დირექტივით. ორივე შემთხვევაში წინასწარდამამუშავებელი შეამოწმებს ჭდე იყო განსაზღვრული თუ არა, როცა #ifdef Flag დირექტივას ჩართავთ. თუ Flag ჭდე არის კოდში, მაშინ ამ დირექტივის შემდეგ მოთავსებული კოდი #endif დირექტივამდე ჩართვება Flag-ის ადგილას, წინააღმდეგ შემთხვევაში არა.

დირექტივები შეიძლება იყვნენ ერთიმეორეში ჩალაგებულნი. #define დირექტივის დამატებაა #undef, რომელიც მოქმედებს #define დირექტივის საპირისპიროდ. #ifdef დირექტივის დამატებაა #ifndef.
**ჰედერ ფაილების სტანდარტი.** ყოველი header ფაილი, რომელიც შეიცავს რაიმე სტრუქტურას, უნდა იყოს შემოწმებული ეს header ფაილი ჩართულია თუ არა კერძო cpp ფაილში. ამას ვაკეთებთ წინასწარდამამუშავებელის მიერ ჭდის შემოწმებით. თუ ჭდე არაა დაყენებული, ეს ნიშნავს, რომ header ფაილი არაა ჩართული. თქვენ უნდა დააყენოთ ჭდე და გამოაცხადოთ სტრუქტურა. თუ ჭდე დაყენებულია, მაშინ უნდა უგულველყოთ კოდი, რომელიც აცხადებს მას. მაგალითი:

```
#ifndef Header_Flag
#define Header_Flag
// აქ არის ტიპის გამოცხადება
#endif // Header_Flag
```

შემდეგი მაგალითი:

```
//: C04:Simple.h
// მარტივი header ფაილი, რომელიც აგვაცილებს ხელმეორედ განსაზღვრებას
#ifndef SIMPLE_H
#define SIMPLE_H
struct Simple {
int i,j,k;
initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

წინასწარდამამუშავებელის ინსტრუქციები უზრუნველყოფენ, რომ არ მოხდეს header ფაილის მრავალჯერ ჩართვა და ამას ჰქვია ჩართვის დაცვა. თუ სახელთა არეს ჩავრთავთ header ფაილში, მაშინ იზღუდება სახელთა არეს გამოყენება. ამიტომ, სახელთა არეს დირექტივა არ უნდა ჩართოთ header ფაილში. როცა ჰქმნით პროექტს C++-ში თქვენ header ფაილში აერთიანებთ სხვადასხვა სტრუქტურებს ერთად. შემდეგ აკეთებთ თვითეული ფუნქციის განსაზღვრას რომელიმე სატრანსლიაციო ერთეულში. როცა იყენებთ ამ ტიპს, თქვენ უნდა ჩართოთ header ფაილი, რომ შეასრულოთ მათი სწორი გამოცხადება.
**ჩადგმული სტრუქტურები.** როცა იყენებთ ჩადგმულ სტრუქტურებს, ამით თქვენ აკავშირებთ ერთიმეორესთან ამ სტრუქტურების ელემენტებს. მაგალითი:

```
//: C04:Stack.h
// ჩადგმული სტრუქტურები დაკავშირებულ სიაში
#ifndef STACK_H
#define STACK_H
struct Stack {
struct Link {
void* data;
```

```
Link* next;
void initialize(void* dat, Link* nxt);
}* head;
void initialize();
void push(void* dat);
void* peek();
void* pop();
void cleanup();
};
#endif // STACK_H ///:~
```

ჩადგმულ სტრუქტურას ჰქვია კავშირი(link) და შეიცავს მიმთითებელს შემდეგ კავშირზე და ასევე მიმთითებელს მონაცემებზე, რომლებიც შენახულია ამ კავშირში. თუ შემდეგი მიმთითებელი არის ნული, ეს ნიშნავს, რომ თქვენ ხართ სიის ბოლოში. შევნიშნოთ, რომ საწყისი მიმთითებელი განისაზღვრება struct Link-ის გამოცხადებით. ჩადგმულ სტრუქტურებს აქვთ საკუთარი initialize() ფუნქცია. Stack-ს აქვს ორივე initialize() და cleanup() ფუნქცია. ასევე აქვს push(), რომელიც იღებს მიმთითებელს Stack-ში მოთავსებულ ელემენტზე და pop(), რომელიც აბრუნებს Stack-ის თავში მოთავსებული ელემენტის მისამართს და ამოაგდებს Stack-იდან ამ ელემენტს. peek() ასევე აბრუნებს Stack-ის თავში მოთავსებული ელემენტის მისამართს, მაგრამ ტოვებს Stack-ში ამ ელემენტს. შემდეგ მაგალითში მოცემულია წევრი ფუნქციების განსაზღვრებები:

```
//: C04:Stack.cpp {O}
// ჩადგმით დაკავშირებული სია
#include "Stack.h"
#include "../require.h"
using namespace std;
void
Stack::Link::initialize(void* dat, Link* nxt) {
data = dat;
next = nxt;
}
void Stack::initialize() { head = 0; }
void Stack::push(void* dat) {
Link* newLink = new Link;
newLink->initialize(dat, head);
head = newLink;
}
void* Stack::peek() {
require(head != 0, "Stack empty");
return head->data;
}
void* Stack::pop() {
if(head == 0) return 0;
void* result = head->data;
Link* oldHead = head;
head = head->next;
```

```
delete oldHead;
return result;
}
void Stack::cleanup() {
require(head == 0, "Stack not empty");
} ///:~
```

პირველი განსაზღვრება საინტერესოა იმით, რომ იგი გვიჩვენებს როგორ განვსაზღვროთ ჩადგმული სტრუქტურის წევრი. Stack : : initialize() აყენებს Stack-ის თავს ნულზე, ე.ი. Stack შეიცავს ცარიელ სიას.
Stack : : push() იღებს არგუმენტს, რომელიც არის მიმთითებელი ცვლადზე, რომელსაც ათავსებს Stack-ში. პირველად იგი იყენებს new-ს, რომ გამოყოს მეხსიერება link-ისთვის, რომელსაც ათავსებს Stack-ის თავში. შემდეგ იძახებს link-ის initialize() ფუნქციას, რომ მიანიჭოს მნიშვნელობები link-ის წევრებს. შევნიშნოთ, რომ next მიმთითებელი უთითებს Stack-ის თავს მოცემულ მომენტში, შემდეგ იგი ენიჭება ახალი link-ის მიმთითებელს. ეს ფაქტიურად ათავსებს ახალ link-ს სიის თავში.
Stack : : pop() იღებს მონაცემის მიმთითებელს Stack-ის თავიდან, გადაადგილებს Stack-ის თავს და ამოაგდებს ძველ Stack-ის თავს და დააბრუნებს ნაპოვნ მიმთითებელს. როცა pop() ამოაგდებს Stack-იდან ბოლო ელემენტს, Stack-ის თავის მიმთითებელი ხდება ნული და ეს ნიშნავს, რომ Stack ცარიელია. Stack : : cleanup() ფაქტიურად არ აკეთებს რაიმე გაწმენდას. კლიენტ პროგრამისტი პასუხისმგებელია ამოყაროს ყველა ელემენტი Stack-იდან. require() გამოიყენება იმისათვის, რომ გვიჩვენოს შეცდომა – Stack არაა ცარიელი. ახლა განვიხილოთ Stack-ის შემოწმების მაგალითი:

```
//: C04:StackTest.cpp
//{L} Stack
//{T} StackTest.cpp
// ჩადგმული დაკავშირებული სიის ტესტირება
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char* argv[]) {
requireArgs(argc, 1); // ფაილის სახელი არის არგუმენტი
ifstream in(argv[1]);
assure(in, argv[1]);
Stack textlines;
textlines.initialize();
string line;
// ფაილის წაკითხვა და სტრიქონების შენახვა სტეკში
while(getline(in, line))
textlines.push(new string(line));
```

```
// სტეკიდან სტრიქონების ამოგდება და მათი დაბეჭდვა:
string* s;
while((s = (string*)textlines.pop()) != 0) {
cout << *s << endl;
delete s;
}
textlines.cleanup();
} ///:~
```

ეს წინა მაგალითის მსგავსია, მაგრამ ათავსებს Stack-ში სტრიქონებს, რომლებსაც იღებს ფაილიდან და შემდეგ ამოგდებს Stack-იდან და ბეჭდავს .რაც ნიშნავს ფაილის სტრიქონების შებრუნებული რიგით ბეჭდვას. requireArgs() აღებულია require.h header ფაილიდან და ამოწმებს ბრძანების სტრიქონში, საიდანაც აღებულია ფაილის სახელი, არის თუ არა არგუმენტების საკმარისი რაოდენობა და თუ არაა იძლევა შეცდომის შეტყობინებას.

**გლობალური ცვლადების საზღვრების დადგენა.** საზღვრების დამდგენი ოპერატორი გვაძლევს საშუალებას ავიღოთ ის ობიექტი, რომელიც ჩვენ გვსურს, როცა გლობალურ და წევრ ობიექტებს აქვთ ერთიდაიგივე სახელი. თუ საზღვარი არაა დაზუსტებული, მაშინ კომპილატორი იღებს გაჩუმებით წევრ ობიექტს. განვიხილოთ მაგალითი:

```
//: C04:Scoperes.cpp
// გლობალური ცვლადის საზღვრის დადგენა
int a;
void f() {}
struct S {
int a;
void f();
};
void S::f() {
::f(); // სხვანაირად იქნებოდა რეკურსიული
::a++; // იღებს გლობალურ a-ს
a--; // იღებს a-ს სტრუქტურიდან
}
int main() { S s; f(); } ///:~
```

**დასკვნა.** ამ ლექციაში ჩვენ განვიხილეთ ახალი სტრუქტურული ტიპი, რომელსაც ჰქვია აბსტრაქტულ მონაცემთა ტიპი, და ცვლადები, რომლებიც იქმნებიან ასეთი ტიპით. მათ ჰქვიათ ამ ტიპის ობიექტები ან მყისიერი მდგომარეობები(instances). ობიექტებისათვის წევრი ფუნქციის გამოძახებას ჰქვია ამ ობიექტისადმი შეტყობინების გაგზავნა. როგორც ვნახეთ, მონაცემებისა და ფუნქციების ერთად მოთავსება სტრუქტურაში იძლევა მოგებას კოდის ორგანიზაციის დროს და უფრო აიოლებს ბიბლიოთეკების გამოყენებას, მაგრამ არსებობენ სხვა უპირატესობები, რომლებსაც შემდეგ ლექციებში გავეცნობით.

# მე-9-10 ლექცია

## რეალიზაციის დაფარვა

**შეზღუდვების დაყენება.** ნებისმიერ დამოკიდებულებაში მნიშვნელოვანია საზღვრების დადგენა დამოკიდებულებაში მონაწილე მხარეებს შორის. როცა, თქვენ ჰქმნით ბიბლიოთეკას, თქვენ ამყარებთ დამოკიდებულებას თქვენსა და კლიენტ-პროგრამისტს შორის. იგი იყენებს თქვენს ბიბლიოთეკას, რომ შექმნას გამოყენებითი პროგრამა ან ახალი ბიბლიოთეკა. C სტრუქტურაში არ არსებობს არავითარი წესები. კლიენტ-პროგრამისტს შეუძლია გააკეთოს ნებისმიერი რამ სტრუქტურაზე, რასაც მოისურვებს. არსებობს ორი მიზეზი, თუ რატომ უნდა დავაწესოთ კონტროლი სტრუქტურის წევრების გამოყენებაზე. პირველი არის ის, რომ კლიენტ-პროგრამისტმა არ უნდა შეეხოს ისეთ წევრებს, რომლებიც მისთვის არაა განკუთვნილი და შექმნილია შიგა საჭიროებისათვის. მეორე მიზეზია, ბიბლიოთეკის შემქმნელს უნდა შეეძლოს შეცვალოს შიგა სამუშაოები ისე, რომ ამან არ მოახდინოს გავლენა კლიენტის პროგრამაზე.

**წვდომის მართვა C++-ში.** C++-ს შემოაქვს სამი გასაღები სიტყვა public, private და protected, რომ დააყენოს საზღვრები სტრუქტურაში. ეს წვდომის დამაზუსტებლები გამოიყენებიან სტრუქტურის გამოცხადებაში და ისინი ცვლიან საზღვრებს ყველა გამოცხადებისათვის, რომლებიც მოსდევენ მათ. ეს გასაღები სიტყვები ბოლოვდებიან ორწერტილით : . public ნიშნავს, რომ ყველა გამოცხადება, რომელიც მოსდევს მას მისაწვდომია ყველასთვის. მაგალითად, სტრუქტურის შემდეგი გამოცხადებები არიან იგივურნი:

```
//: C05:Public.cpp
// Public ჰგავს C-ს struct
struct A {
int i;
char j;
float f;
void func();
```

```
};
void A::func() {}
struct B {
public:
int i;
char j;
float f;
void func();
};
void B::func() {}
int main() {
A a; B b;
a.i = b.i = 1;
a.j = b.j = 'c';
a.f = b.f = 3.14159;
a.func();
b.func();
} ///:~
```

მეორე მხრივ, private გასაღები სიტყვა ნიშნავს, რომ არავის არ შეუძლია გამოიყენოს წევრები ამ სრუქტურის შემქმნელს გარდა. ამგვარად, private არის კედელი თქვენსა და კლიენტ-პროგრამისტს შორის. თუ ვინმე შეეცდება მის გამოყენებას, კომპილატორი გამოიტანს შეცდომის შეტყობინებას. განვიხილოთ მაგალითი:

```
//: C05:Private.cpp
// საზღვრების დაყენება
struct B {
private:
char j;
float f;
public:
```

```
int i;
void func();
};
void B::func() {
i = 0;
j = '0';
f = 0.0;
};
int main() {
B b;
b.i = 1; // OK, public
//! b.j = '1'; // შეცდომაა, private
//! b.f = 1.0; // შეცდომაა, private
} ///:~
```

func() შეიძლება გამოიყენოს B-ს ნებისმიერმა წევრმა, მაგრამ main()-ს არ შეუძლია. protected მოქმედებს private-ს მსგავსად იმ განსხვავებით, რომ მემკვიდრე სტრუქტურებს(განისაზღვრება მოგვიანებით) აქვთ წვდომა მათზე.

**მეგობარი ფუნქციები.** თქვენ შეგიძლიათ ცხადად დართოთ ნება კონკრეტულ ფუნქციას გამოიყენოს სტრუქტურის წევრები, თუ friend განაღები სიტყვა წინ უსწრებს ამ ფუნქციის გამოცხადებას სტრუქტურაში. განვიხილოთ მაგალითი:

```
//: C05:Friend.cpp
// Friend განსაზღვრავს სპეციალურ წვდომას
// სტრუქტურის არასრული გამოცხადება:
struct X;
struct Y {
void f(X*);
};
private:
```

```
int i;
public:
void initialize();
friend void g(X*, int); // გლობალური friend
friend void Y::f(X*); // სტრუქტურის წევრი friend
friend struct Z; // მთელი სტრუქტურა არის friend
friend void h();
};
void X::initialize() {
i = 0;
}
void g(X* x, int i) {
x->i = i;
}
void Y::f(X* x) {
x->i = 47;
}
struct Z {
private:
int j;
public:
void initialize();
void g(X* x);
};
void Z::initialize() {
j = 99;
}
void Z::g(X* x) {
x->i += j;
}
void h() {
X x;
x.i = 100; // მონაცემის პირდაპირი გამოყენება
```

```
}
int main() {
X x;
Z z;
z.g(&x);
} ///:~
```

struct Y აქვს წევრი ფუნქცია f(), რომელიც აზუსტებს X ტიპის ობიექტს. ეს დასაშვებია, რადგანაც X ნაწილობრივ გამოცხადებულია, სანამ გამოვაცხადებდეთ T :: F(X*). ამის შემდეგ f() ფუნქცია შეიძლება გამოცხადდეს როგორც მეგობარი ფუნქცია X-ში.

**ჩადგმული მეგობარი ფუნქციები.** სტრუქტურის ჩადგმულად გაკეთება არ იძლევა ავტომატურად წვდომას private წევრებზე. რომ მივაღწიოთ ამას, პირველ რიგში, უნდა გამოვაცხადოთ ჩადგმული სტრუქტურა, შემდეგ უნდა გამოვაცხადოთ როგორც friend და ბოლოს განვსაზღვროთ ეს სტრუქტურა. ამ სტრუქტურის განსაზღვრება უნდა იყოს გამოყოფილი მისი friend გამოცხადებისაგან. წინააღმდეგ შემთხვევაში მას კომპილატორი გაიგებს როგორც არა წევრს. განვიხილოთ მაგალითი:

```
//: C05:NestFriend.cpp
// ჩადგმული  მეგობრები
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;
struct Holder {
private:
int a[sz];
public:
void initialize();
struct Pointer;
friend struct Pointer;
```

```
struct Pointer {
private:
Holder* h;
int* p;
public:
void initialize(Holder* h);
// გადაადგილება მასივში:
void next();
void previous();
void top();
void end();
// წვდომა მნიშვნელობებზე:
int read();
void set(int i);
};
};
void Holder::initialize() {
memset(a, 0, sz * sizeof(int));
}
void Holder::Pointer::initialize(Holder* rv) {
h = rv;
p = rv->a;
}
void Holder::Pointer::next() {
if(p < &(h->a[sz - 1])) p++;
}
void Holder::Pointer::previous() {
if(p > &(h->a[0])) p--;
}
void Holder::Pointer::top() {
p = &(h->a[0]);
}
void Holder::Pointer::end() {
```

```
p = &(h->a[sz - 1]);
}
int Holder::Pointer::read() {
return *p;
}
void Holder::Pointer::set(int i) {
*p = i;
}
int main() {
Holder h;
Holder::Pointer hp, hp2;
int i;
h.initialize();
hp.initialize(&h);
hp2.initialize(&h);
for(i = 0; i < sz; i++) {
hp.set(i);
hp.next();
}
hp.top();
hp2.end();
for(i = 0; i < sz; i++) {
cout << "hp = " << hp.read()
<< ", hp2 = " << hp2.read() << endl;
hp.next();
hp2.previous();
}
} ///:~
```

რადგანაც Pointer გამოცხადებულია, ამიტომ ნებადართულია წვდომა Holder-ის private წევრებზე წინადადებით:

```
friend struct Pointer;
```

struct Holder შეიცავს მთელი ტიპის მასივს და Pointer ნებას გრთავთ მისწვდეთ მას. რადგან Pointer არის ცალკე კლასი, თქვენ შეგიძლიათ main()-ში შექმნათ ერთზე მეტი მათგანი და აირჩიოთ მასივის სხვადასხვა ნაწილები. სტანდარტული C ბიბლიოთეკის ფუნქცია memset() <String>-დან იყენებს მეხსიერების კონკრეტულ მისამართზე(პირველ არგუმენტს) კონკრეტულ მნიშვნელობაზე(მეორე არგუმენტი) n ბაიტს(მესამე არგუმენტი) დაწყებული სასტარტო მისამართიდან. იგი ცვლის ციკლს, რომელიც თქვენ უნდა გაგეკეთებინათ მთელი მასივის ელემენტების გასავლელად. წვდომის დამაზუსტებლები არიან სტრუქტურის ნაწილები და არავითარ გავლენას არ ახდენენ სტრუქტურის მიერ შექმნილ ობიექტზე. წვდომის ყველა დამაზუსტებელი ინფორმაცია ქრება პროგრამის თვლაზე გაშვების მომენტისათვის. თვლაზე გაშვებულ პროგრამაში, ობიექტები ხდებიან მეხსიერების ნაწილები და მეტი არაფერი. თქვენ შეგიძლიათ დაარღვიოთ წვდომის ყველა წესი და მიმართოთ მეხსიერებას პირდაპირ. მაგრამ ამ შემთხვევაში შესრულებული მოქმედებების სისწორეზე პასუხიმგებლობას იღებთ თქვენს თავზე და C++ არაა დაპროექტებული იმისათვის, რომ ამაში ხელი შეგიშალოს თქვენ.

**კლასი.** წვდომის მართვას ხშირად უწოდებენ რეალიზაციის დაფარვას. სტრუქტურაში ფუნქციების მართვა ჰქმნის მონაცემთა ტიპს თვისებებითა და ქცევებით, სადაც წვდომის მართვა სვამს საზღვრებს მონაცემთა ტიპის შიგნით ორი მნიშვნელოვანი მიზეზის გამო. პირველი მიზეზი არის, რომ დადგინდეს კლიენტ-პროგრამისტს რის გამოყენება შეუძლია და რის არა. ეს იწვევს მეორე მიზეზს – გამოვყოთ ინტერფეისი რეალიზაციისაგან. თუ სტრუქტურა გამოყენებულია სხვადასხვა პროგრამაში და კლიენტის პროგრამას შეუძლია მხოლოდ გაუგზავნოს შეტყობინებები public ინტერფეისს, მაშინ თქვენ შეგიძლიათ შეცვალოთ ნებისმიერი რამ, რომელიც არის public და თქვენ გარანტირებული ხართ, რომ ამით საჭირო არ იქნება კლიენტის პროგრამის გადაკეთება. ინკაფსულაცია და წვდომის მართვა ერთად აღებული უფრო მეტი რამ არის, ვიდრე სტრუქტურა. ახლა ჩვენ ვართ ობიექტებზე ორიენტირებული დაპროგრამების სამყაროში, სადაც რაიმე

სტრუქტურა აღწერს ობიექტების კლასს. კლასს-ის გამოყენება C++-ში ახლოსაა იმასთან, რომ გახდეს არააუცილებელი გასაღები სიტყვა. იგი იგივეა, რაც struct გასაღები სიტყვა გარდა იმისა, რომ კლასი გაჩუმებით გულისხმობს private-ს მაშინ, როცა struct გაჩუმებით გულისხმობს public-ს. განვიხილოთ ორი სტრუქტურა, რომლებიც იძლევიან ერთიდაიგივე შედეგს:

```
//: C05:Class.cpp
// class და struct-ის მსგავსება
struct A {
private:
int i, j, k;
public:
int f();
void g();
};
int A::f() {
return i + j + k;
}
void A::g() {
i = j = k = 0;
}
// იგივე შედეგები მიიღება class-ით:
class B {
int i, j, k;
public:
int f();
void g();
};
int B::f() {
return i + j + k;
}
void B::g() {
i = j = k = 0;
```

```
}
int main() {
A a;
B b;
a.f(); a.g();
b.f(); b.g();
} ///:~
```

თუ ჩვენ სტრუქტურაში გამოვიყენებთ private-ს, მაშინ იგი ხდება კლასი და პირიქით, თუ კლასში გამოვიყენებთ public-ს, მაშინ იგი გახდება სტრუქტურა. ახლა, შევასწოროთ Stash, რომ გამოვიყენოთ წვდომაზე მართვა:

```
//: C05:Stash.h
// გარდაქმნილია, რომ გამოვიტენოთ წვდომაზე მართვა
#ifndef STASH_H
#define STASH_H
class Stash {
int size; // ყოველი არეს სიგრძე
int quantity; // მეხსიერების არეების რაოდენობა
int next; // შემდეგი ცარიელი არე
// დინამიკურად განაწილებული ბაიტების მასივი:
unsigned char* storage;
void inflate(int increase);
public:
void initialize(int size);
void cleanup();
int add(void* element);
void* fetch(int index);
int count();
};
#endif // STASH_H ///:~
```

inflate()-ს აქვს private და იგი უკვე აღარ ეკუთვნის ინტერფეისს. მართლაც, იგი გამოიყენება მხოლოდ შიგა მიზნებისათვის და იგი არაა საჭირო კლიენტ-პროგრამისათვის. ახლა, განვიხილოთ Stack-ის მოდიფიკაცია წვდომაზე კონტროლით, რაც ხდის მას კლასად:

```
//: C05:Stack2.h
// ჩადგმული სტრუქტურები დაკავშირებული სიით
#ifndef STACK2_H
#define STACK2_H
class Stack {
struct Link {
void* data;
Link* next;
void initialize(void* dat, Link* nxt);
}* head;
public:
void initialize();
void push(void* dat);
void* peek();
void* pop();
void cleanup();
};
#endif // STACK2_H ///:~
```

რადგან რეალიზაცია არ იცვლება, მას არ ვიმეორებთ აქ. ასევე test არ იცვლება.