

<http://www.facebook.com/computertechnology2010>

<http://pc-tech-h-tech.blogspot.com/>

Salih Mripa ing. i dipl.

Programimi në C dhe C++

Shtëpia botuese
PHOENIX

Titulli
Programimi në C dhe C++

Autor
Salih Mripa, ing. i dipl.

Recensentë
Ismail Zhilivoda, ing. i dipl.

Korrektura
Gazmend Berljolli

Ballina:
Fisnik Ismaili, dizajner

Tirazhi: 1000 ekzemplarë

*Prindërve të mi që më edukuan
dhe më udhëzuan në rrugë të drejtë*

PËRMBAJTJA

PARATHËNIE	6
ORIGJINA E GJUIHËVE C DHE C++	8
PROGRAMET E THJESHTA DHE TIPET E TË DHËNAVE.....	12
1.1. PROCEDURA E PROGRAMIMIT	12
1.2. KOMPILIMI I PROGRAMIT.....	14
1.2.1 Metodat e kompajlimit të Kodit	17
1.3. STRUKTURA E PROGRAMIT NË GJUHËN PROGRAMUESE C	18
1.4. TIPET ELEMENTARE TË TË DHËNAVE	21
1.4.1 Konstantat.....	23
1.4.2 Variablat.....	25
1.4.3 Operacionet aritmetike	27
1.4.4 Operatori cast (kast).....	29
1.5. AUTO-OPERATORËT.....	30
1.6. FAJLLI HEADER	33
1.6.1 Libraria Standarde për futjen dhe shrypjen e të dhënave	33
1.7. TIPET E DEFINUARA NGA PËRDORUESI.....	37
1.8. KOMANDAT E PRIPROCESORIT	38
1.9. TIPI ARRAY (RADHA)	41
USHTRIME	46
PËRMBLEDHJE	47
VENDIMET DHE VORBULLAT	48
2.1. KOMANDA IF-ELSE.....	48
2.1.1 Operatori ternar	51
2.2. OPERATORËT LOGJIKË.....	52
2.3. KOMANDA SWITCH.....	53
2.4. VORBULLAT	56
2.4.1 Vorbulla for.....	57
2.4.2 Vorbullat while dhe do-while.....	60
2.5. KOMANDA BREAK.....	62
2.6. KOMANDA CONTINUE	64
2.7. KOMANDA GOTO DHE ETIKETAT.....	64
2.8. TIPAT BOOLEAN.....	65
USHTRIME	67
PËRMBLEDHJE	68
FUNKSIONET.....	69
3.1. DIZAJNIMI TOP-DOWN (TOP-DAUN)	69
3.2. VARIABLAT GLOBALE	71
3.3. KOMUNIKIMI NDRËMJET FUNKSIONEVE.....	73
3.3.1 Parametrat.....	73
3.3.2 K:himi i informatave nga funksionet.....	75

3.4 PROTOTIPI I FUNKSIONIT	78
3.5 HAPESIRA E VEPRINTARISE DHE JETA E VARIABLAVE.....	80
USHTRIME	82
PERMBLEDHJE	83
TREGUESIT	84
4.1 TREGUESIT	84
4.1.1 Operacionet për tregues	87
4.2 TREGUESIT DHE VARIABLAT ARRAY.....	89
4.3 VARIABLAT ARRAY TE TIPIT TREGUES.....	92
4.4 TREGUESIT NË TREGUES	94
4.5 REZERVIMI I MEMORIES DINAMIKE.....	95
4.6 TREGUESIT NË FUNKSIONE	102
USHTRIME	109
PERMBLEDHJE	110
5.1 NEVOJA PER PROGRAMIMIN OBJEKT-ORIENTUES	112
5.2 CIKLI I APLIKACIONEVE.....	114
5.3 QELLIMET E INXHINJERINGUT TË SOTFUERIT	115
5.4 LIBRARITË STANDARDE	116
5.4.1 Komentet.....	117
5.4.2 Përdorimi i librarive standarde.....	117
5.5 STRING (VARGU)	124
PERMBLEDHJE	127
KLASËT (CLASS)	128
6.1 TIPI I RI KOHA	128
6.2 DEFINIMI I KLASËS (CLASS)	130
6.3 KONSTRUKTORËT DHE DEKONSTRUKTORËT	133
6.3.1 Definimi i funksioneve të klasëve.....	135
6.3.2 Konstruktorët shndërrues	138
6.3.3 Konstruktorët kopjues.....	142
6.3.4 Konstruktorët shndërrues dhe zëvendësimi i tipeve të ndryshme.....	146
6.3.5 Operatori this	148
6.4 KLIENTËT	151
6.5 FUNKSIONET NGARKUESE.....	151
6.6 VLERAT E VARIABLAVE NGARKUESE (VLERAT BAZË).....	154
6.7 VARIABLAT DHE FUNKSIONET STATIKE	156
6.8 KRUIMI I TIPEVE ABSTRAKTE DHE PSHEHJA E TË DHËNAVE	158
USHTRIME	161
PERMBLEDHJE	162
ORGANIZIMI I KODIT NË C++	163
USHTRIME	170
TRASHËGIMI DHE HIERARKIA NË KLASË	171
8.1 TRASHËGIMI I KLASËVE.....	174

8.1.1 Kontoja e thjeshtë dhe klasa bazë.....	174
8.1.2 Kontoja me interes.....	177
8.1.3 Super Kontoja.....	179
8.1.4 Kontoja me çeqe.....	181
8.1.5 Kontoja me interes dhe me çeqe.....	182
8.1.6 Zëvendësimet standarde.....	185
8.2 LLOJET E TRASHËGIMIT.....	187
USHTRIME.....	188
PËRMBLEDHJE.....	188
POLIMORFIZMI DHE LIDHJA DINAMIKE.....	189
9.1 FUNKSIONET VIRTUAL.....	193
9.2 KLASËT ABSTRAKTE.....	198
USHTRIME.....	206
PËRMBLEDHJE.....	207
MANIPULIMI ME FAJLL.....	208
10.1 MANIPULIME ME FAJLL NË C++.....	213
10.1.1 Hapja e fajllit.....	213
10.1.2 Hapja e fajllit për lexim dhe shkrim.....	216
10.1.3 Përdorimi i fajllit pa renditje.....	220
10.1.4 Manipulime me fajll në metodën binare.....	225
10.2 HIERARKIA E KLASËVE IOS.....	226
USHTRIME.....	229
PËRMBLEDHJE.....	230
SHABLLONET.....	231
11.1 TIPET ABSTRAKTE GJENERIKE.....	233
11.1.1 Tipi Vektori.....	234
11.1.2 Tipi Rreshiti.....	250
11.2 PËRSËRITËSIT.....	259
11.3 UDHËZIMI TYPE NAME.....	263
11.4 TREGUESIT E MENÇUR (SMART POINTERS).....	266
USHTRIME.....	272
PËRMBLEDHJE.....	273
OPERATORËT CAST.....	274
12.1 OPERATORI CAST STATIC_CAST.....	277
12.2 OPERATORI CAST DYNAMIC_CAST.....	282
12.3 OPERATORI CAST CONST_CAST.....	290
12.4 OPERATORI CAST REINTERPRET_CAST.....	293
USHTRIME.....	294
PËRMBLEDHJE.....	295
PËRSËRITJA.....	296
13.1 KONCEPTET E PËRSËRITJES.....	296

13.2 METODA PËRÇAJ-E-SUNDO DHE PËRSËRITJA.....	296
13.3 FUNKSIONET PËRSËRITSE DHE JO-PËRSËRITSE NË C++	297
13.4 PËRSËRITJA DHE PËRCJELLJA E MEMORIES STAK	302
USHTRIME	305
PËRMBLEDHJE	305
GABIMET DHE TRAJTIMI I GABIMEVE.....	306
14.1 TRAJTIMI I GABIMEVE NË GJUHËN C	307
14.2 TRAJTIMI I PËRJASHTIMEVE.....	311
14.2.1 Përrjashtimet e papritura.....	318
14.3 KOMPLETIMI I KLASËS VEKTORI.....	319
USHTRIME	329
PËRMBLEDHJE	329
SHTOJCA A	330
KULLA E HANOIT	330
SHTOJCA B.....	338
B.1 FJALËT E REZERVUARA NË GJUHËN C++	338
SHTOJCA C	339
SIMBOLET SPECIALE	340
SHTOJCA D	341
OPERACIONET NË BIT	341
Operatori AND	341
Operatori OR.....	342
Operatori XOR	343
Operatori Kompliment.....	343
Operatorët për zhvendosje.....	344
SHTOJCA E.....	346
PËRPARËSIA E OPERATORËVE	346
BIBLIOGRAFIA	348
INDEKSI.....	349

Parathënie

Dëshira e flaktë për përparimin arsimor të popullit më të vjetër në Ballkan më shtyu ta shkruaj këtë libër. Studentë të dashur, për shkrimin e këtij libri shpenzova orë të panjehura vetëm e vetëm që ju të mos humbni kohën e ligjëratave me diktimin e një libri të shkruar në gjuhën e huaj.

Për leximin e këtij libri preferohet të keni njohuri për numrat binarë, manipulimin me numrat binarë, matematikën logjike në gjuhët programuese si dhe njohuri të vogël në programim në ndonjë gjuhë programuese.

Në institucionet arsimore në përfundim si gjuhë programuese fillestare preferohet të mësohet gjuha e familjes *Pascal*. Arsyeja është se kjo familje e gjuhëve pretendon t'u shmanget vetive të këqija programuese si dhe është me më shumë tipe kontrolluese se gjuha C dhe disa gjuhë të tjera.

Në disa institucione arsimore si gjuhë fillestare mësohet gjuha e familjes së ashtuquajtur *gjuhë e programimit funksional*. Këto lloj gjuhësh janë gjuhë të gjeneratës së katërt dhe janë përkthyes direkte të problemeve matematikore.

Në këtë libër kam pretenduar t'ju njoftoj shkurtimisht me gjuhën C si paraardhëse kryesore e gjuhës C++, ndonëse njaft autorë të librave për gjuhë programuese kanë shkruar për gjuhën C++, pa i prekur aspak përparësitë a mangësitë e gjuhës programuese C. Mirëpo, këta autorë kanë marrë parasysh se studentët kanë së paku njohuri në ndonjë gjuhë të familjes *Pascal* ose kanë sadopak njohuri të gjuhës C, ose të ndonjë gjuhe tjetër programuese.

Gjuha C nuk preferohet të mësohet si gjuhë fillestare, sepse ju lejon të shkruani programe me gabime (të cilat vërehen vetëm gjatë ekzekutimit të programit), gabime të cilat në gjuhë të tjera programuese nuk do të ishin të mundshme nga që vetë kompajleri do t'i paraqiste gabimet para kompajlimit.

Me rëndësi është se gjuha C është dizajnuar nga vetë programerët me qëllim të rritjes së efikasitetit të programeve dhe zvogëlimit të kodit për një program. Programet në gjuhën C janë kompakte dhe efikase, çka ka bërë që kjo gjuhë të përdoret në programimin e shumë aplikacioneve të softuerit.

Mirëpo, sot teknologjia ka arritur në nivelin ku pjesa teknike mundëson që edhe programet e shkruara në gjuhën e gjeneratës së katërt të jenë pothuajse me të njëjtin efikasitet sikurse programet e shkruara në C. Në teknologjinë më të re të procesimit parallel, disa gjuhë janë më efikase në këtë drejtim.

Në jetën e përditshme efikasiteti i programit (shpejtësia e ekzekutimit) për aplikacionet e softuerit për organizata të ndryshme nuk është aq me rëndësi,

sa është i rëndësishëm përfundimi i programit (kodimi), me kohë dhe zvogëlimi i gabimeve në program.

Mirëpo, në disa organizata, si p.sh. në paisjet për kontrollimin e aeroplanit, programet duhet të jenë me efikasitet maksimal (duhet të ekzekutohen menjëherë), se përndryshe vonimi i ekzekutimit të programit do të shkaktonte ndonjë aksident katastrofal. Këto programe, ku shpejtësia e ekzekutimit është prioritet, quhen real time systems (lexo: rill tajm sistëms). Në këtë libër nuk do të merremi me problemet në sistemet "real time", sepse kjo është lami në vete dhe do të duhej të trajtohej në një libër të veçantë.

Ky libër nuk është i mjaftueshëm për ta mësuar programimin në C apo C++. Programimi në këto gjuhë, ose në çfarëdo gjuhe programuese tjetër, mund të përvetësohet vetëm me praktikimin e gjuhës programuese në probleme të ndryshme dhe ushtrime intensive në programim. Gjuha programuese është sikur ndonjë gjuhë natyrore (si psh anglishtja) dhe nëse nuk përdoret, shumë lehtë harrohet. Mirëpo, nëse përdoret rregullisht, ajo gjuhë vetëmse përvetësohet më shumë. Është e preferueshme që secili problem, ose ndonjë komandë e re e përdorur në libër, të zbatohet praktikisht menjëherë.

Për shpjegimin e çdo mundësie të gjuhëve C dhe C++ do të nevojiteshin disa libra. Studimi dhe përvetësimi i këtyre gjuhëve iu lihet vetë përdoruesve (programerëve të ardhshëm) dhe ju asnjëherë nuk duhet të mbeten vetëm në nivelin e njohurive të përfituara nga ky libër. Programimi përvetësohet duke bërë hulumtime edhe në gjuhët e tjera dhe nga përdorimi i vetive më të mira të gjuhëve të ndryshme.

Në fund kërkoj ndjesë për disa terme të përdorura në gjuhën angleze. Lamia e informatikës është e re dhe po zhvillohet aq shpejt, sa që ne nuk kemi pasur ndonjë standardizim të terminologjisë së saj në gjuhën shqipe. Në lami e informatikës nuk do të kishte qenë aq fatale nëse disa fjalë (psh string, stream etj.) përdoren në gjuhën origjinale (angleze). Në anën tjetër, jam plotësisht i mendimit që çdo term të jetë në gjuhën shqipe, mirëpo derisa ta kemi një terminologji standarde, kalkimi i fjalëpërfjalshëm i terminologjisë së informatikës nuk do të kishte fare kuptim.

Përdorimi i termeve në gjuhë të huaja në shkencë ka edhe ndikimin negativ në ndryshimin e gjuhës së përditshme. Psh. *fjala kompjuter* që rrjedh nga fjala angleze me kuptim *llogaritës* përdoret aq shumë në jetën e përditshme, sa që përdorimi i fjalës *llogaritës* në vend të fjalës *kompjuter* do të shkaktonte huti. Jam i mendimit se fjala *llogaritës* është mjaft e përshtatshme për ta zëvendësuar fjalën *kompjuter*, mirëpo disa terme, si p.sh. *string* (që ka kuptimin e fjalisë, rreshtit, renditjes së germave etj.) nuk është e lehtë të zëvendësohet me ndonjë fjalë shqipe. Për këto fjalë duhen të gjenden dhe të standardizohen fjalë të përshtatshme nga gjuhëtarët në bashkëpunim me shkenctarët e informatikës.

Origjina e gjuhëve C dhe C++

Gjuha programuese C është shkruar gjatë viteve 1970 nga dy programerë, D.M. Ritchie dhe Ken Thompson, punonjës të Laboratorit Bell në Shtetet e Bashkuara të Amerikës. Fillimisht kjo gjuhë u dizajnuar kryekrejt për të shkruar programe për Sistemin Operativ (SO) si dhe programe të ndërlidhura me Sistemin Operativ.

Gjuha programuese C është pasardhëse e gjuhës programuese BCPL (e krijuar nga Martin Richards) dhe e gjuhës programuese B (e krijuar nga Ken Thompson). Ideja për emrin e saj vjen nga pararendësja, gjuha programuese B (kjo vetë duke u emërtuar me inicialin e laboratorit Bell në SHBA). Plotësimet dhe ndryshimet që u bënë në gjuhën programuese B, rezultuan në një gjuhë aq të ndryshme, që si rrjedhojë u emërtua me emrin e ri C, dhe jo, ta zëmë, B versioni 2.0. Ndryshimet dhe plotësimet ishin aq kardinale, sa që për dy dekada e radhën gjuhën programuese C në listën e gjuhëve programuese më të fuqishme. Versioni i parë i gjuhës programuese C është shkruar nga D.M. Ritchie. Mbas tij filluan të dilnin versione të ndryshme të saj, derisa në vitin 1983 Institucioni Amerikan i Standardit Kombëtar (ANSI) krijoi komisionin për të caktuar një version të gjuhës C të pavarur nga Sistemi Operativ. Ky version u quajt ANSI C. Në këtë kohë, vetëm me kompajllimin e programit në sisteme operative të ndryshme, pothuaj çdo program i shkruar në ANSI C mund të ekzekutohej në cilindo sistem operativ. Ky ishte njëri prej shkaqeve kryesore që çoi në popullarizimin e gjuhës C. Që nga nxjerrja e versionit ANSI C ka pasur disa versione, e njëri prej tyre është edhe gjuha programuese C++. Mirëpo edhe në C++ pjesa më e madhe e strukturës dhe e stilit të gjuhës ANSI C ka mbetur e pandryshuar.

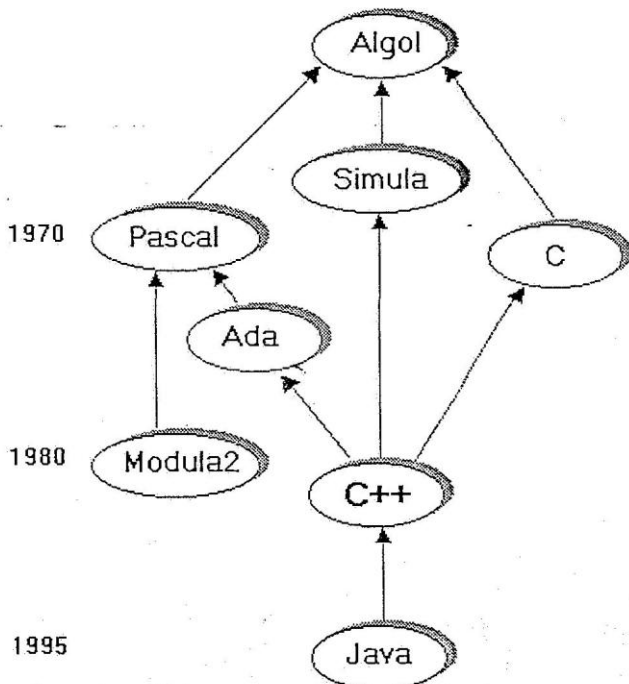
Njëra prej arsyeve kryesore për krijimin e gjuhës programuese C ishte që kjo ta ndihmonte zhvillimin i sistemit operativ UNIX. Versioni i mëhershëm i sistemit operativ UNIX ishte shkruar në gjuhën Assembler, kështu që ky sistem operativ ishte shumë i mvarur nga makina në të cilën ishte instaluar dhe përdorej, dhe me shumë vështirësi transferohej në makina të tjera. Përparësia më e madhe e të shkruarit së sistemit operativ në nivel të lartë të gjuhës programuese ishte se sistemi operativ bëhej kompatibil dhe i përshtatshëm për shumë sisteme kompjuteristike.

Përveç sistemit operativ UNIX, edhe sistemi operativ Microsoft Windows është shkruar në gjuhën C. Gjuha C kombinon përparësitë që ka gjuha programuese e nivelit të lartë me efikasitetin e gjuhës Assembler.

Versioni i ri i kësaj gjuhe programuese (C++) për herë të parë filloi të përdorej në vitin 1983 jashtë institucioneve për hulumtime dhe eksperimente. Kjo gjuhë i ka trashëguar tiparet elementare të gjuhës C, megjithëse disa tipare i janë ndryshuar për ta krijuar gjuhën me kontroll më të fortë mbi sintaksën e

kodit programues si dhe të modelit. Gjuha programuese C++ njihet si gjuhë objekt-orientuese. Ajo është njëra prej anëtarëve më të vonshëm të familjes së gjuhës programuese Algol. Të gjitha këto gjuhë njihen si gjuhë imperative. Diagrami i mëposhtëm paraqet trashëgiminë e disa vetive nga një gjuhë programuese në tjetrën. Ky diagram është thjeshtësuar, duke i lënë pas dore gjuhët më pak të njohura.

1960



1995

Gjuha C++ ka trashëguar të gjitha tipet e të dhënave nga gjuha C duke shtuar tipin me referencë. Kurse nga gjuha Ada dhe Simula trashëgoi konceptin e klasës, kështu që përdoruesit mund t'i definojnë tipet e veta si klasë, së bashku me funksionet që manipulojnë këto tipe. Gjuha C++ ua mundëson përdoruesve definimin e operatorëve të ngarkuar, kështu që veprimi i një operatori të definuar paraprakisht mund të ridefinohet për manipulimin e tipit të sapodefinituar.

Gjuha C++ ka trashëguar konceptin e shablloneve nga Ada (shih kapitullin për shabllonet). Kjo gjuhë programuese u mundëson programerëve t'i shkruajnë programet me përdorimin e metodologjisë programuese objekt-

orientuese. Gjuhë objekt-orientuese janë ato gjuhë programuese të cilat ofrojnë metoda për programim, në të cilat programeri së pari i shqyrton tipet e objekteve që programi do t'i përdorë (disa tipe janë të definuara nga kompajleri si p.sh. int me operacionet e vlefshme si +, -, *, %), si dhe operacionet e mundshme (metodën për manipulim) për këto objekte. Për këtë do të bëhet fjalë më gjerësisht në kapitujt përkatës.

Gjuha C++ mundëson disa stile të programimit, mirëpo për përfitime maksimale nga efikasiteti i programit, programi duhet shkruar në stilin e gjuhës C++, pra në stilin objekt-orientues.

Programimi në C



Programet e thjeshta dhe tipet e të dhënave

1.1. Procedura e programimit

Programet e shkruara në gjuhët C, C++, Pascal etj. njihen si programe të nivelit të lartë. Ato shkruhen në kod të kuptueshëm, që u ngjason gjuhëve natyrore (si anglishtja), dhe shumë të ndryshëm nga kodi i numrave binarë. Mirëpo, kompjuteri e njeh vetëm gjuhën e numrave binarë (të formuar nga shifrat 0 dhe 1). Të gjitha udhëresat kompjuterit i jepen në numra binarë.

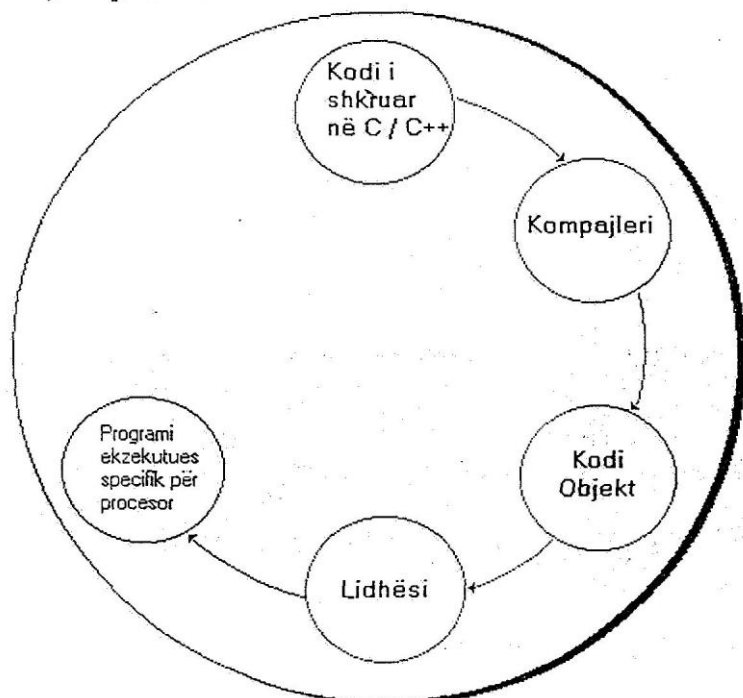
Si është e mundur, atëherë, që kompjuteri t'i kuptojë gjuhët C ose C++ apo ndonjë gjuhë tjetër programuese?

Në filltet e epokës kompjuterike, programet shkruheshin në kartela dhe instrukcionet ekzekutoheshin me shpimin e këtyre kartelave në vende të caktuara. Këto kombinime të vrimave formonin numra binarë që përfaqësonin urdhëresat për kompjuterin. Pastaj filloi të përdorej gjuha Asambler, gjuhë e cila zëvendësonte disa urdhëresa (numra binarë) me komanda në shkronja. Këto komanda ishin më të lehta për t'u mbajtur mend, prandaj më e lehtë ishte të punohej me to, sesa me numrat binarë. Më vonë u paraqitën gjuhë programuese më të kuptueshme sesa Asambleri dhe që njëkohësisht ishin edhe më afër gjuhës njerëzore, si p.sh. gjuhët programuese COBOL, C, Pascal etj. Gjuhët C, C++, Pascal etj., njihen si gjuhë të nivelit të tretë dhe, krahasuar me gjuhën Asambler dhe numrat binarë, këto gjuhë e kanë lehtësuar shumë procesin e programimit. Për t'u kuptuar nga kompjuteri një program i shkruar në gjuhë programuese të nivelit të tretë, ky program duhet të shndërrohet në instrukcione binare, punë kjo të cilën e kryen programi i ashtuquajtur *kompajler*. Kompajleri e shndërron kodin e shkruar në gjuhën C, C++, Pascal etj. në instrukcione të kuptueshme nga kompjuteri (procesori).

Ekzistojnë shumë lloje të procesorëve (disa prej tyre janë Intel, Motorola etj), prandaj nevojitet një kompajler për çdo tip të procesorit. Gjuha Asambler nuk

ishte shumë kompatible me procesorët e llojeve të ndryshme për arsye se ishte shumë afër gjuhës së procesorit. Prandaj, programet e shkruara për një lloj procesori duheshin rishkruar pothuajse nga fillimi për procesorin e llojit tjetër. Gjuha programuese C e zgjidhi këtë problem, sepse programet e shkruara në të punonin pothuajse në çdo lloj të procesorit pa ndryshime të mëdha në program. Këto ndryshime të shpeshtën rezultojnë nga llojet e ndryshme të kompajlerëve dhe nga dallimi i instruksioneve specifike për procesorë të llojeve të ndryshme. Megjithatë, programet e shkruara në gjuhën C ishin më të përdorshme për procesorë të llojeve të ndryshme sesa programet e shkruara në Asambler (të cilat duheshin shkruar rishtazi). Kjo mundësoi suksesin e madh që pati gjuha programuese C.

Gjuhët programuese të nivelit të lartë nuk kuptohen nga procesorët dhe për këtë arsye janë deri diku të pamvarura nga llojet e procesorëve. Për t'u kuptuar nga procesorët, këto gjuhë kalojnë në disa stade. Programet e shkruara në C dhe C++, për t'u shndërruar në kod specifik të procesorit, kalojnë nëpër këto stade:



Transferimi i programit në kod të procesorit specifik

Programet e shkruara në C dhe C++, kalojnë gjatë kompajlimit nëpër një stad më shumë sesa programet e shkruara në Pascal, Mpdula 2, etj. Ky është stadi i paraprosesorit, i cili i zëvendëson makrot (shih më gjerësisht në kapitujt e arshëm) në vlerat e definuara në program si dhe i bashkangjet fajllat (komanda #include). Pasi të kompletohet kodi nga paraprosesori, d.m.th. të bëhet zëvendësimi i makrove me vlerat përkatëse si dhe bashkangjitja e fajllave, kompajleri e shndërron programin në të ashtuquajturin *kod objekt* (kod i përkohshëm i cili është më afër gjuhës së procesorit). Në fazën përfundimtare programi i quajtur Linker (lexo: llinkër), d.m.th. "lidhësi", bën bashkangjitjen e kodit objekt dhe e shndërron atë në program ekzekutues.

Përveç gjuhëve kompajluese, të cilat e kompajlojnë kodin e shkruar në gjuhë të nivelit të lartë, kemi edhe gjuhët e njohura si gjuhë interpretuese (p.sh. gjuha BASIC). Gjuhët interpretuese në vend që të manipulojnë me kodin e shkruar në atë gjuhë dhe ta shndërrojnë në kod specific të procesorit (natyrisht duke bërë edhe optimizimin e kodit), interpretojnë çdo komandë në kod specifik të procesorit.

1.2 Kompajlimi i programit

Kodi i shkruar në ndonjë gjuhë programuese specifike mund të ruhet në çfarëdo edituesi të përgjithshëm, përderisa ruhet si tekst i thjeshtë. Teksti që paraqet kodin e programit kompajlohet me anë të kompajlerit të gjuhës programuese specifike dhe pastaj lidhet (bashkangjitet) për ta formuar programin ekzekutues për procesor specifik.

Versionet e ndryshme të harduerit (procesorëve) të përdorur për kompjuter si dhe versionet e ndryshme të sistemeve operative, kanë paraqitur nevojën për versione të ndryshme të kompajlerëve. Versionet e ndryshme të kompajlerëve kanë ndikuar që procedura e kompajlimit të jetë sadopak e ndryshme ndërmjet kompajlerëve të ndryshëm të gjuhës së njëjtë programuese. Disa kompajlerë kanë opsione të ndryshme për optimizimin e programit ekzekutues, si dhe opsione të tjera që janë specifike për procesorë të ndryshëm. Disa kompajlerë e bashkojnë editorin, kompajlerin dhe lidhësin në një program për t'ia lehtësuar programerit punën e përgjithshme në krijimin e produktit final. Këto programe janë të ndryshme, mirëpo paraqitja dhe shpalimi i tyre nuk përbën ndonjë synim të këtij libri. Këtu do të paraqesim kompajlimin e kodit nëpërmjet komandës së kompajlerit, pasi që kjo metodë është pak a shumë e ngjashme në të gjithë kompajlerët e gjuhës C dhe C++.

Le të fillojmë me kompajlimin e kodit në sistemin operativ UNIX duke marrë si shembull kompajlerin gcc. Le të supozojmë se kodi që vijon është ruajtur në fajllin main.c:

```
#include <stdio.h>

void main (void){
    printf("Tungjatjeta!");
}

fajlli main.c
```

Komanda në vijim e kompajlon fajllin main.c dhe krijon fajllin ekzekutues a.out.

```
gcc main.c
```

Nëse do të dëshironim që fajlli ekzekutues i krijuar ta ketë emrin tung, atëherë do ta përdornim këtë komandë:

```
gcc main.c -o tung
```

Ta shohim rastin kur programi ekzekutues është i përbërë prej më shumë se një fajlli të vetëm. Le të jetë programi test i përbërë prej fajllave main.c, test.h dhe test.c. Atëherë për kompajlimin e këtij programi do ta përdornim këtë komandë:

```
gcc main.c test.c -o test
```

Programet, në të shumtën e rasteve, janë të përbëra prej më shumë se një fajlli secila prej të cilave mund të jetë e mvarur nga kodi i një apo më shumë fajlli tjetër. Shpeshherë, programet e përbëra prej shumë fajllash e ngadalësojnë procesin e kompajlimit të kodit. Prandaj kompajlerët lejojnë kompajlimin e çdo fajlli veçmas, si më poshtë:

```
gcc -c main.c
gcc -c test.c
gcc main.o test.o -o test
```

gjë që mundëson kompajlimin e vetëm atyre fajllave në të cilat është ndryshuar kodi. Kjo shpejton procesin e kompajlimit të projekteve të mëdha, të përbëra prej shumë fajllave. Mirëpo në projekte përmasash të mëdha do të ishte mjaft e vështirë t'i kontrollonim fajllat e ndryshuara, në mënyrë që t'i kompajlonim vetëm ato. Për ta automatizuar procesin e gjetjes së fajllave të ndryshuara dhe të kompajlimit të tyre për ta krijuar produktin final, përdoren fajllat e ashtuquajtura makefile, me anë të programit make. Këta do të shpjegojmë përciptazi se si krijohen fajllat makefile, duke filluar u thelluar në

të gjitha mundësitë e këtyre fajllave, pasi që për këtë do të nevojitej kapitull i veçantë. Për ilustrim, shihni fajllin `makefile` në vijim për programin e përbërë prej fajllave `main.c`, `test.h` dhe `test.c`.

```
test : main.o test.o
    gcc main.o test.o -o test

main.o : main.c test.h test.c
    gcc -c main.c

test.o : test.h test.c
    gcc -c test.c
```

fajlli `makefile`

Në këtë fajll e kemi definuar fajllin ekzekutues `test` si fajll të mvarur nga fajllat `main.o` dhe `test.o` dhe për ta krijuar fajllin ekzekutues `test` nevojitet të ekzekutojmë komandën që vijon:

```
gcc main.o test.o -o test
```

Pastaj kemi definuar se fajlli `main.o` është e mvarur nga fajllat `main.c`, `test.h` dhe `test.c` e kështu me radhë. Në këtë mënyrë, nëse njëri fajll ndryshohet, atëherë programi `make` do t'i kompajlojë vetëm ato fajlla që mvaren nga fajlli i ndryshuar për krijimin e programit ekzekutues.

Ta shohim tani kompajlimin e kodit në gjuhën C dhe C++ për sistemin operativ *Microsoft Windows*. Sikurse në sistemin operativ UNIX dhe në sistemet operative të tjera, edhe në sistemin operativ *Microsoft Windows* kemi disa versione të kompajlerëve të gjuhës C dhe C++. Ndër kompajlerët më të njohur në sistemin operativ *Microsoft Windows* janë Borland C++ nga kompania Inprise dhe Microsoft Visual C++ nga Microsofti. Këtu do ta paraqesim kompajlerin Microsoft Visual C++ pasi që ky është njëri ndër më të përdorshmit.

Komanda e kompajlerit Visual C++ është e njëjtë sikurse komanda e kompajlerit `gcc`, ndonëse në vend të programit ekzekutues `gcc` përdoret programi `cl`. Pra për kompajlimin e programit të përbërë prej fajllave `main.c`, `test.h` dhe `test.c` do ta përdornim këtë komandë:

```
cl main.c test.c -o test
```


Për ta mundur përdorimin e komandës së kompajlerit Visual C++ së pari duhet ekzekutuar fajllin `vcvars32.bat` nga fajlli `Autoexec.bat`, i cili fajll i ndërron variablat e sistemit operativ.

Kompajleri Visual C++ mundëson krijimin e fajllit `makefile` në mënyrë grafike, gjë që e lehtëson punën për krijimin dhe kompajlimin e programeve të mëdha. Mirëpo, për ta kompajluar kodin me anë të kompajlerit Visual C++ nuk është e nevojshme të përdoret fajlli `makefile` apo komanda e kompajlerit, pasi që procesi i kompajlimit të programit mund të kryhet në mënyrë grafike.

1.2.1 Metodat e kompajlimit të Kodit

Zakonisht në gjuhët programuese që kompajlohen (pra në ato gjuhët programuese që nuk interpretohen) ka disa metoda për kompajlimin e kodit. Këto janë mënyrat e kompajlimit për ta krijuar:

- ♦ fajllin ekzekutues më të vogël
- ♦ fajllin ekzekutues për shpejtësi
- ♦ fajllin për përmbajtjen e të dhënave në Unicode (kodi që përfshin edhe ortografitë jolatine si atë arabe, japoneze etj.) ose të dhënave normale (gjuhët me alfabet latin, sikurse është gjuha shqipe),
- ♦ fajllin që përmban të dhëna rreth kodit dhe fajllat e përfshira në krijimin e programit ekzekutues, e njohur si metoda debug (lexo: dibag).

Mirëpo të gjitha këto mënyra klasifikohen në vetëm dy grupe:

- ♦ Metoda Release (lexo: ri'lis), ku bëjnë pjesë tri mënyrat e para, dhe
- ♦ Metoda Debug.

Metoda Release përdoret për krijimin e programit ekzekutues sipas nevojës së klientëve të programit. Ndërkaq, metoda Debug përdoret nga programerët (ndonjëherë edhe nga testuesit e programeve) gjatë programimit të sistemit. Metoda Debug, siç u tha më sipër, përmban më shumë të dhëna sesa metoda Release dhe për këtë arsye fajllat ekzekutues në metodën Debug janë shumë më të mëdha. Këto të dhëna të tepërta janë të pakuptueshme për klientët e thjeshtë dhe nuk përdoren në produktin përfundimtar.

Të dhënat e ruajtura në fajllin ekzekutues të kompajluar me metodën Debug janë shumë të vlefshme për gjetjen e shpejtë të gabimeve në programin ekzekutues, dhe për këtë arsye preferohet të përdoret kjo metodë e kompajlimit në fazën fillestare të projektit. Ndërsa, gjatë testimit të programit preferohet metoda Release e kompajlimit.

1.3 Struktura e programit në gjuhën programuese C

Në gjuhën programuese C nuk ka ndonjë rregull të posaçme për renditjen e programit apo të funksioneve që marrin pjesë në kompletimin e programit. Zakonisht programet (kodi i shkruar) duhen renditur në atë mënyrë që ta bëjnë të mundshme lexueshmërinë dhe inspektimin e kodit të programit.

Programet zakonisht përmbajnë fajlla të krijuara për definimin e variablave dhe të funksioneve. Është e udhës që programet e gjata të ndahen në nënprograme dhe të ruhen në fajlla të veçanta. Pastaj këto fajlla mund të bashkohen gjatë kompajlimit. Kjo është shumë më e preferuar sesa programi të ruhet vetëm në një fajll.

Programi (problem) i ndarë në nënprograme është më i lehtë të kodohet, është më i lexueshëm dhe më i lehtë të testohet për gabime të mundshme. Natyrisht që ka edhe përjashtime: p.sh. shumica e komandave të Sistemit Operativ UNIX janë krijuar nga një fajll i vetëm.

Figura e mëposhtme (Fig. 1.1) paraqet formatin e mundshëm për programe me vetëm një fajll :

```
[ Direktivat e Pri-procesorit ]  
[ definimi i variablave statike ]  
[ definimi i funksioneve të tjera ]  
  
void main(void)  
{  
    [definimi i variablave private]  
  
    /* pjesa për program */  
}
```

Figura 1.1

Deklarimet në mes të kllapave katrore në figurën 1.1 nuk janë të domosdoshme dhe ato nuk janë prezente në të gjitha programet.

Në raste kur kodin për një program e kemi të zbërthyer në shumë fajlla, secili fajll do të ketë format të ngjashëm me atë në figurën 1.1, ndonëse vetëm njëra prej tyre do ta ketë funksionin main (funksioni kryesor).

Ekzekutimi i kodit të programit në C fillon në krye të funksionit main. Duke qenë se programi mund ta ketë vetëm një pikënisje, secili program në C mund të ketë vetëm një funksion main.

Si shembull kemi marrë programin që nxjerr fjalën "Tungjatjeta !" në monitor (figura 1.2). Ky program është shumë i thjeshtë dhe tashmë është bërë traditë në informatikë që ky të merret si shembull për shpjegimin e programit të thjeshtë në cilëndo gjuhë të re programuese.

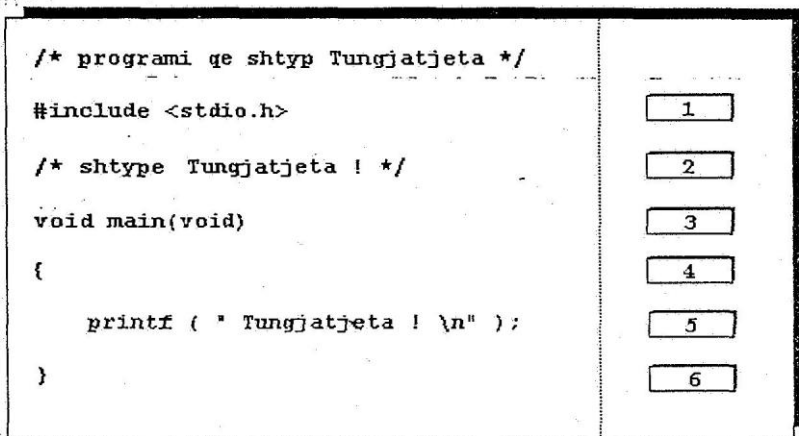


Figura 1.2

Në këtë figurë çdo rresht të kodit të programit e kemi shoqëruar me një numër rendor në kolonën e djathtë, për t'i dhënë më lehtë shpjegimet fillestare të secilit prej tyre:

1. Direktiva e paraprocesorit për të përfshirë përmbajtjen e fajllit të specifikuar në kllapa në këtë pozitë të programit.
2. Komentet në program të cilat shkruhen vetëm për të shpjeguar kodin dhe nuk marrin pjesë në kompajlimin e programit final. Komentet shkruhen në mes të simboleve /* dhe */. Komentet në një rresht mund të paraprihen edhe me simbolet //, por vetëm në gjuhën C++. Nëse komenti është më i gjatë se një rresht atëherë përdorimi i simboleve /* */ është më i udhëzueshëm sesa nisja e çdo rreshti me simbolet //.

Gjatë kodimit të programit preferohet të përdoren komentet për të shpjeguar qëllimin dhe funksionin e kodit si dhe pjesët e kodit në të cilat bëhen kalkulime të komplikuar.

3. Funksioni main (funksioni kryesor), i cili nuk ka parametra dhe nuk kthen asnjë vlerë. Funksioni main i tregon kompajlerit se ku fillon ekzekutimi i programit. Ky funksion mund të shkruhet ndryshe për të pranuar parametra (vlera) të pasuara në program para ekzekutimit.
4. Këtu kllapa gjarpërore e majtë paraqet fillimin e funksionit main, mirëpo gjetiu kllapa gjarpërore do të paraqitste fillimin e ekzekutimit të një blloku të kodit.
5. Funksioni printf i definuar në librarinë standarde të gjuhës programuese C, që nxjerr vargun e simboleve ndërmjet thonjzave në monitor. (Në këtë rast fjalën Tungjatjeta :)
6. Kllapa gjarpërore e djathtë paraqet fundin e funksionit main. Ndërmjet kllapës gjarpërore të majtë dhe asaj të djathtë mund të ketë më shumë se një thirrje të funksioneve të tjera.

Preferohet që në fillim të fajllit ku shkruani kodin e programit, t'i komentoni funksionet që do t'i implementoni në atë fajll. Nëse programi është i ndarë në disa fajlla, atëherë komentoni funksionet përkatëse në ato fajlla ku i përdorni.

Nëse pas këtij shpjegimi, kodi i mësipërm mbetet ende i pakuptimtë, s'ka vend për panik. Me kohë do t'i shpjegojmë hollësisht pjesët e programit (kodit) në gjuhën C si dhe gjuhën në C++. Programin e mësipërm merreni si mostër për programe që kryejnë një funksion të thjeshtë.

Një gjë që duhet mësuar në programim është ecuria e programit në një gjuhë programuese të dhënë (p.sh. në gjuhë C). Në lëndën e informatikës nuk është e udhës të mësohen përmendësh funksionet "përbërëse" të asaj gjuhe programuese. Me rëndësi është të mësohet se si mund ta shprehim problemin konkret në mënyrë sa më efektive nëpërmjet asaj gjuhe programuese.

1.4 Tipet elementare të të dhënave

Çdo gjuhë programuese përmban tipet elementare për ruajtjen e të dhënave. Me anë të këtyre tipeve elementare mund të krijojmë tipe të tjera për t'i ruajtur të dhënat që i përshtaten problemit specifik.

Disa gjuhë programuese kanë përparësi ndaj gjuhëve të tjera në krijimin e tipeve të reja, mirëpo përparësitë e gjuhëve mvaren edhe nga vetitë e tjera.

Gjuha C ofron këto tipe elementare të të dhënave:

```
char /* për simbole e integjerë (numra të plotë) të vegjël */
int /* për integjerë (negativë e pozitivë) */
float /* për numra realë */
double /* për numra realë me precizitet të dyfishtë */
```

Tipi `char` zakonisht është 8 bit. Madhësia e tipeve të tjera mvaret nga lloji i makinës (kompjuterit) dhe i sistemit operativ. Një variabël (ndryshore) e tipit `char` inicohet kështu:

```
char shkronja ;
```

Tani kemi një variabël të tipit `char` të quajtur `shkronja`. Për t'i dhënë një vlerë variablës sonë, veprojmë si më poshtë:

```
shkronja = 'a' ;
```

Variabla `shkronja` e tipit `char` tani ka vlerën `a`. Vëreni përdorimin e pikëpresjes (;) në fund të çdo rreshti të kodit. Pikëpresja shënon fundin e një instruksioni.

Tipi `int` mund t'i ketë këto nëntipe:

```
short
long
unsigned
register
```

Madhësia në bit e vlerës së ruajtur në nëntipin `short` dhe `long` mvaret prej llojit të makinës (kompjuterit). Përdorimi i këtyre nuk është i domosdoshëm. Gjuha C nuk e bën të detyrueshëm përdorimin e këtyre nëntipeve dhe në disa raste `short int`, `int` dhe `long int` zënë sasi të njëjtë të memories. Mirëpo duhet pasur parasysh se `long int` është gjithmonë më i madh ose baras me `int`, dhe `int` është gjithmonë më i madh ose baras me `short int`.

Nëntipi unsigned specifikon se vlera e ruajtur në tipet char, short, int apo long duhet të trajtohet vetëm si vlerë pozitive, psh:

```
unsigned short numri ;
```

Në këtë rast variabla numri mund ta ketë vlerën prej 0 – 65535 (Ky kufi është marrë si shembull në sistemin operativ 32 bitsh).

Nëse variablën numri do ta definorim si:

```
short numri;
```

atëherë variabla numri mund t'i ketë vlerat prej -32728 deri në 32767.

E njëjta vlen edhe për variablat e tipit char. Definimet e variablës shkronja në kodin që vijon mund të kenë vlera maksimale të ndryshme:

```
/* vlerat e mundshme prej 0 - 255 */
unsigned char shkronja;
```

```
/* vlerat e mundshme prej -127 deri në 127 */
char shkronja;
```

Pra, tipi char mund t'i ruajë deri në 255 lloje të simboleve (shih shtojcën C). Për disa gjuhë sikur japonishtja ose kinezishtja, të cilat kanë më shumë se 255 grafema në ortografinë e tyre, tipi char nuk do të na hynte në punë për të manipuluar me to. Për të mundësuar manipulimin me grafema të gjuhëve të cilat kanë më shumë se 255 sosh, duhet përdorë tipin wchar i cili zakonisht është 16 bitsh. Tipi wchar do ta mundësojë ruajtjen e 65535 llojeve të simboleve.

Tipi register int e informon kompajlerin se variabla e deklaruar është kandidate për t'u vendosur në regjistrin e makinës llogaritëse. Kjo e shpejtëson ekzekutimin e programit. Vetëm variablat që përdoren shpesh (si p.sh. numruesit), duhen deklaruar si register. Përdorimi i këtij tipi nuk e detyron kompajlerin që variablën e deklaruar ta ruajë në regjister të makinës. Kompajleri mund të refuzojë që variablën ta ruajë në regjister të makinës nëse të gjithë regjistrat janë në përdorim.

Tipet float dhe double përdoren për ruajtjen e vlerave të numrave realë. Memoria e nevojitur për variablat e tipit double është dyfish më e madhe sesa memoria e nevojitur për variablat e tipit float. Kjo është për arsye se tipi

double manipulon me numrat realë me përpikmëri më të madhe sesa tipi float.

Deklarimi i variablave të tipit float dhe double bëhet kështu:

```
float pesha;
double pi;
```

1.4.1 Konstantat

Konstantat e tipit char zakonisht ngarkohen me vlerë të një simboli:

```
'a' , 'b' , 'c' , '$' , '5'.
```

mirëpo ka raste kur konstantat përmbajnë më shumë se një simbol:

```
'\n' , '\t' etj.
```

Simboli '\' në gjuhën C ka domethënie të veçantë, sepse përdoret në kombinim me simbolet e tjera për t'i shënuar shkronjat e ashtuquajtura «të padukshme» (p.sh. simboli Enter, kryerreshti, zilha e kompjuterit etj.), si dhe për t'i shënuar simbolet e tjera të veçanta që përdoren në program (p.sh. pikëpyetja).

Gjuha C ka këto simbole speciale:

\a	zilha e kompjuterit	\\	simboli '\'
\b	kthyesi mbrapa	\?	pikëpyetja '?'
\f	fundi i faqes	\'	apostofi '
\n	rresht i ri	\"	thonjëzat "
\v	kryerresht vertikal	\t	kryerresht horizontal

Konstantat e tipit char mund të shprehen në disa mënyra, p.sh. shkronja 'a' mund të shkruhet edhe si '\141',

Pra:

```
putchar('\141'); si dhe
putchar('a');
```

do ta japin të njëjtin rezultat. Të dy komandat shtypin shkronjën a.

Numrat në thonjëza janë ose numra oktalë (më së shumti treshifrorë) ose heksadecimalë (x i përcjellë me numër më së shumti dyshifror).

Numrat oktalë duhen të përdoren vetëm për paraqitjen e simboleve të padukshme. Numrat oktalë janë numrat e paraprirë nga 0 (zero), p.sh.:

0123 , 032 , -010

kurse numrat heksadecimalë jepen si më poshtë:

0xffff , 0xa , -0x10

Përdorimi i numrave oktalë dhe heksadecimalë bëhet në aplikacionet ku konstantat kanë domethënie speciale si p.sh. gjatë shkrimit të kodit të programeve për mekanizmat e makinës apo për aplikacione të tjera në të cilat është i domosdoshëm manipulimi me bit.

Konstantat e tipit integjer të përcjella me shkronjën L trajtohen si numra me precizitet të dyfishtë. Për shembull në rastin e mëposhtëm:

384675L

prapashtesa L i tregon kompajlerit se konstantja është e tipit long.

Numrat realë (pra konstantat e tipit float dhe double) paraqiten me pikë (jo me presje dhjetore) p.sh.:

34.21 , -25.12 , 3.4e+3

Konstantat e tipit string janë ato konstanta që përbëhen prej zero apo më shumë simboleve të ngërthyera në thonjëza, p.sh:

"Kjo është një konstantë vargore"

ose

"" /* string null (lexo: null) - varg pa asnjë simbol */

Teknikisht konstanta string është array (shih 1.7), elementet e të cilës janë simbole. Kompajleri automatikisht e përfundon stringun (vargun) me simbolin zero (i quajtur *null*; kështu që programet mund ta gjejnë fundin e stringut (gjatë manipulimit me string).

Psh stringu "Tungjatjeta" në memorie ruhet kështu:

T	u	n	g	j	a	t	j	e	t	a	\0
---	---	---	---	---	---	---	---	---	---	---	----

Kompajleri e cakton se në cilën memorie ruhen konstantat. Mirëpo zakonisht, nëse sistemi ofron memorie të mbrojtur, atëherë kompajleri do t'i ruajë në memorie vetëm si të lexueshme (kjo memorie vetëm mund të kundrohet nga përdoruesi, e jo të ndryshohet prej tij). Nëse tentoni ta ndryshoni vlerën e konstantave, pas ekzekutimit të programit, sistemi ju njofton se keni bërë gabim.

1.4.2 Variablat

Variablat e tipeve të përmendura më sipër definohen duke shkruar së pari tipin e variablës, të përcjellë me emrin e variablës p.sh.:

```
int    numri;          /* definon variablën numri të tipit int */
int    x , y;          /* definon dy variabla të tipit int */
long   int rroga;      /* definon variablën rroga të tipit
                        int me precizitet të madh */
char    shkronja,
        gjinia;        /* definon dy variabla të tipit char */
float   pesha;         /* definon variablën pesha të tipit float */
double  shpejtesia;    /* definon variablën shpejtesia të tipit
                        double - numër real me precizitet
                        të dyfishtë */
```

Në gjuhën C, emrat e variablave përbëhen kryesisht prej shkronjave të alfabetit (a-z), simbolit '_' (nënviza) dhe, së fundi, shifrave (0-9). Simboli i parë në emrin e variablës patjetër duhet të jetë shkronjë (a-z) ose nënvizë. Gjuha C nuk i merr për të njëjta shkronjat e mëdha dhe të voglat, p.sh.:

```
int Numri , numri ; /* deklaron dy variabla të ndryshme */
```

Duhet t'i shmangeni përdorimit të variablave me emra të njëjtë ose shumë të ngjashëm. P.sh. do ta kishit të vështirë në program t'i shquanit variablat numri, numr, Numri, duke i ngatërruar shumë lehtë njërin për tjetrin. Është e udhës që variablës t'i jepni emrin e vlerës që do ta ketë në program (psh. rroga për të mbajtur vlerën e rrogës, pesha për të shënuar peshën e kështu me radhë). Kjo e lehtëson mbikqyrjen dhe më vonë, mirëmbajtjen e programit.

Variabla mund t'u jepen vlerat gjatë definimit të tyre, duke përdorë shenjen e barazimit '=', psh:

```
int mosha = 22;          /* definon variablën mosha
```

dhe ia ngarkon vlerën 22 */

Ngarkimi i vlerës së variablave në këtë mënyrë zakonisht bëhet për konstanta dhe variabla statike. Variablat e tjera deklarohen pa marrë ndonjë vlerë fillestare, dhe vlera u ngarkohet më vonë.

Në kodin që vijon kemi deklaruar variablën numri të tipit int dhe e kemi inicuar me zero në shprehjen for (për të cilën do të flasim në kapitullin e ardhshëm). Vlerën e variablës numri e kemi ndryshuar me anë të autorritësit ++.

```
int    numri;

for ( numri = 0 ; numri <= 10 ; numri++ )
    printf ( " Tani numri e ka vlerën %d \n", numri);
```

Gjuha C është gjuhë me kontroll të dobët mbi tipet e variablave të përdorura në program. Shihni rastin në vijim, ku kemi deklaruar dy variabla të tipeve të ndryshme dhe ia kemi ngarkuar vlerën e tipit tjetër:

```
int    numri;
char   gjinia;

numri = 23.56;          /* ia ngarkon vlerën 23 variablës numri
                        (mbetja 0.56 zhduket) */
gjinia = 5;             /* ia ngarkon vlerën 5 variablës gjinia */
```

Këto ngarkime në gjuhën C janë të lejueshme edhe pse konstantja 23.56 është e tipit float e jo int, dhe konstantja 5 është e tipit int e jo char (vini re: '5' është e ndryshme nga 5. E para e paraqet simbolin '5', kurse e dyta renditjen e simbolit të pestë në alfabetin ASCII të makinës).

Urdhërat si:

```
numri = 23.56 ;    /* ku numri është i tipit int */
```

zakonisht nuk përdoren në programe. Këtu vetëm kemi ilustruar se variabla numri në rast të tillë do ta merrte vlerën 23 e jo 23.56. Shmangituri këtij stili të programimit.

Ndërkaq urdhëri si:

```
char n_vogel = 3; /* n_vogel është variabël e tipit char */
```

përdoret shpesh si variabël e tipit int të vogël (e cila mund t'i ketë vlerat prej 0-255 ose -128 deri 127). Variablat char gjithmonë do ta mbajnë vlerën si int, edhe nëse ia kemi ngarkuar vlerën si shkronjë p.sh.:

```
n_vogel = 'a';
```

Variabla `n_vogel` do ta mbajë vlerën 97 (ku numri 97 paraqet renditjen e shkronjës `a` në kodin ASCII - alfabetin kompjuterik).

Pra:

```
n_vogel = 'a'; është e njëjtë si n_vogel = 97;
```

Vlera e variablës `char` mund të trajtohet si `int` apo `char`, mvarësisht nga kodi që përdorni në program. Stilet e mira për programim sugjerojnë që variabla `char` të ngarkohet me vlerën `symbol`, kur trajtohet si variabël që mban simbol, pra që vlera t'i ngarkohet si e tillë `'a'`, `'7'`, `'@'` etj. Nëse trajtohet si `int` (numër i plotë), atëherë duhet t'i ngarkohen vlerat si 4, 16, 127 etj.

Variablat e tipit `long int` (ose vetëm `long`) mund të definohen dhe të inicohen kështu:

```
long int lNumriIMadh;
```

ose

```
long lNumriIMadh;
```

dhe inicohen me konstanta `int` me mbreshtesën `L`, psh:

```
lNumriIMadh = 1512151212L;
```

1.4.3 Operacionet aritmetike

Operacionet aritmetike për tipet e thjeshta të ofruara nga kompajleri janë këto:

`+`, `-`, `*`, `/` operacionet elementare.

`%` operacion që si vlerë kthen mbetjen gjatë pjesëtimit të dy numrave, p.sh.:

`5 % 2 = 1`

`7 % 4 = 3`

`3 % 8 = 3` etj.

Të gjitha këto operacione janë operacione binare. Vlerat e variablave për këto operacione mund të jenë të tipit `int`, `char`, `float` ose `double`. Kurse për `%`, vlerat e variablave duhet të jenë të tipit `int` ose `char`.

Në gjuhën C, nëse gjatë një operacioni, vlerat e variablave janë të tipeve të ndryshme, atëherë variabla e tipit më të "ulët" shndërrohet në atë të tipit më të "lartë", p.sh.:

```
float x ;
x = 5 * 2.5 ;
```

numri 5 shndërrohet në 5.0 (numër real) dhe pastaj kryhet operacioni (shumëzimi):

```
x = 5.0 * 2.5 ;
```

kurse në rastin:

```
x = 5 / 2 ;
```

variabla x do ta ketë vlerën 2 (e jo 2.5) për arsye se të dy operatorët janë të tipit më të "ulët" (numra të plotë), prandaj edhe rezultati do të jetë i tipit të "ulët".

Nëse dëshirojmë që rezultati ta marrë vlerën reale 2.5 atëherë udhëzimin $x = 5/2$; duhet ta shkruajmë në njërën prej këtyre formave:

```
x = 5.0 / 2 ;      ose
x = 5 / 2.0 ;      ose
x = 5.0 / 2.0 ;
```

Operatorët kanë renditje hierarkike të përparësisë së llogaritjes ndaj njëritjetrit dhe këto përparësi për disa operatorë llogariten duke filluar prej të djathtës në të majtë e për disa operatorë të tjerë prej të majtës në të djathtë (për të gjithë operatorët shih Shtojcën D).

Për ta ilustruar këtë, vëreni kodin në vijim:

```
int x;
x = 5 + 2 * 3;
```

Rezultati do të jetë 11, pra $5 + (2*3)$, e jo 21, ose $(5+2) * 3$, që d.m.th. se operatori * ka përparësi ndaj operatorit +. Gjithmonë duhet pasur kujdes gjatë formulimit të shprehjeve matematikore, si dhe duhet pasur parasysh përparësitë e operatorëve. Për t'u ikur problemeve të mundshme (nga moskujdesi ose duke menduar gabimisht se një operator ka përparësi ndaj ndonjë operatori tjetër), preferohet t'i përdorni kllapat e vogla, të cilat i tejkalojnë përparësitë e operatorit superior dhe kështu vlerat në kllapa

kalkulohen së pari. P.sh. nëse keni menduar ta bëni mbledhjen para shumëzimit, atëherë kodi në vijim do të jetë i shkruar në rregull:

```
x = (5 + 2) * 3;
```

Edhe nëse keni menduar që shumëzimin ta bëni para mbledhjes, nuk do të ishte gabim t'i përdornit kllapat:

```
x = 5 + (2 * 3);
```

Ndonëse në këtë rast jemi të sigurtë se shumëzimi do të kryhej para mbledhjes edhe pa përdorimin e kllapave, shumë lehtë do të mund të gabonim me operatorët e tjerë duke menduar gabimisht se një operator ka përparësi ndaj operatorit tjetër. Përdorimi i kllapave përpos që mundëson t'u shmangeni gabimeve të mundshme, bën që kodi të jetë më i lexueshëm dhe më i kuptueshëm.

1.4.4 Operatori cast (kast)

Ky operator është shumë i përdorshëm në gjuhën C. Operatori cast përkohsisht e ndërron tipin e variablës që përdoret në ekuacion. Nëse i kemi deklaruar variablat e tipeve të ndryshme dhe tentojmë të operojmë me këto variabla në një shprehje:

```
int    numruesi, emeruesi;
float  rezultati;

numruesi    = 5;
emeruesi    = 2;

rezultati = numruesi / emeruesi ;
```

dihet tashmë, nga ajo që thamë më parë, se variabla rezultati do ta ketë vlerën të plotë (pa pjesën pas presjes decimale). Nëse duam që rezultati ta ketë vlerën reale, atëherë ose numruesi ose emeruesi duhet ta kenë vlerën reale. Kjo mund të arrihet duke e përdorë operatorin cast, pa ndërrimin e definimit të tipit të variablës numruesi apo emeruesi.

```
rezultati = (float) numruesi / emeruesi ;    /* ose */
rezultati = numruesi / (float) emeruesi ;    /* ose */
```

```
rezultati = (float) numruesi / (float) emeruesi ;
```

Në rastin e parë tipi i variablës numruesi ndërrohet përkohësisht në tip realë (float), dhe pastaj operacioni aritmetik kryhet sipas teorisë së përgjithshme mbi operacionet aritmetike. Në rastin e dytë tipi i variablës emeruesi ndryshohet përkohësisht, kurse në rastin e tretë të dy tipet e variablave (numruesi dhe emeruesi) ndryshohen përkohësisht.

Operatori cast zakonisht përdoret me tregues (pointer) të variablave, për të ndryshuar tipin tregues të variablës. Ky operator përdoret për ta ndryshuar përkohësisht tipin e një objekti në ndonjë tip tjetër që pritet në parametra të funksioneve apo në shprehje të tjera. Duhet pasur kujdes gjatë përdorimit të këtij operatori, sepse mund të ketë efekte të padëshirueshme gjatë ekzekutimit të programit.

Operatori cast përdoret edhe në gjuhën C++, mirëpo gjuha C++ mundëson mënyra të tjera, më të sigurta, për t'iu shmangur përdorimit të operatorit cast (shih kapitullin 12).

1.5 Auto-operatorët

Në gjuhën C kemi auto-operatorët e mëposhtëm:

```
++      /* auto-rritësi */
dhe --  /* auto-zbritësi */
```

Të dy këta operatorë mund të përdoren me variabla të tipit short, int, long dhe char. Ta zëmë:

```
int viti = 1997;

viti++;      /* rrit vlerën e variablës viti për një */
++viti;      /* po ashtu rrit vlerën e variablës viti për një */
viti--;      /* zbret vlerën e variablës viti për një */
--viti;      /* zbret vlerën e variablës viti për një */
```

Mvarësisht nga pozita e auto-operatorit në raport me variablën (pra vendosja e auto-operatorit para ose prapa variablës) edhe rezultatet e kalkulimeve do të jenë të ndryshme. Në rastin e mëposhtëm:

```
x = (++y) - 5; /* rasti A */
```

së pari rritet vlera e variablës *y* dhe vlera e saj e re pastaj përdoret për operacionin e zbritjes, kurse me auto-operatorin prapa variablës:

```
x = (y++) - 5; /* rasti B */
```

së pari përdoret vlera e vjetër e variablës *y* (pa u rritur) në operacionin e zbritjes dhe pastaj rritet për një. Nëse vlera fillestare e variablës *y* do të ishte 10, atëherë në rastin A variabla *x* do të jetë e barabartë me 6, kurse në rastin B me 5.

Auto-operatorët e tjerë janë:

```
+= , -= , *= , /=
```

Ti ilustrojmë funksionet e tyre me një shembull:

```
int viti = 1950;
int numri = 3;
```

```
viti += 56;      /* e rrit vlerën e variablës viti për 56 */
numri *= 4;     /* vlera e variablës numri shumëzohet me 4 */
```

Udhëzimi `viti+=56;` është i barasvlefshëm me udhëzimin `viti = viti + 56;` e po ashtu edhe udhëzimi `numri *= 4;` është i barasvlefshëm me udhëzimin `numri = numri * 4;`. Keni parasysh se përdorimi i auto-operatorëve ++ dhe -- mund të japë rezultate të paplanifikuara. Funksioni i këtyre operatorëve shpesh është i mvarur nga lloji i kompajlerit dhe nga përparësitë e operatorëve të tjerë. Për ta provuar problemin e auto-operatorëve, programin në vazhdim e kemi kompajluar nëpër kompajlerë të ndryshëm dhe ekzekutimi i tij si rrjedhojë ka dhënë rezultate të ndryshme:

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

```
main (*)
```

```
{
    int x = 1;
    printf("\n1) ++x * ++x = %d\n", ++x * ++x); /*a*/
    x = 1;
    printf("\n2) x++ * x++ = %d\n", x++ * x++); /*b*/
    x = 1;
    printf("\n3) ++x * x++ = %d\n", ++x * x++); /*c*/
    x = 1;
    printf("\n4) x++ * --x = %d\n", x++ * --x); /*d*/
}
```

Në kompajlerin Microsoft Visual C++ 1.5 programi ka dhënë këto rezultate:

- a) $++x * ++x = 6$
- b) $x++ * x++ = 2$
- c) $++x * x++ = 4$
- d) $x++ * ++x = 3$

Në kompajlerin Microsoft Visual C++ 6.0 programi ka dhënë këto rezultate:

- a) $++x * ++x = 9$
- b) $x++ * x++ = 1$
- c) $++x * x++ = 4$
- d) $x++ * ++x = 4$

Në kompajlerin CC të Sun UNIX-it programi ka dhënë këto rezultate:

- a) $++x * ++x = 9$
- b) $x++ * x++ = 2$
- c) $++x * x++ = 6$
- d) $x++ * ++x = 3$

Dhe në kompajlerin gcc të sistemit operativ UNIX programi ka dhënë rezultatet:

- a) $++x * ++x = 6$
- b) $x++ * x++ = 1$
- c) $++x * x++ = 4$
- d) $x++ * ++x = 2$

Pra keni kujdes gjatë përdorimit të auto-operatorëve. Nëse këta operatorë përdoren pa u kombinuar me të tjerët:

```
x++;      ose
++x;
```

atëherë rezultati do të jetë i pamvarur nga tipi i kompajlerit. Përdorimi pa kujdes i auto-operatorëve shkakton gabime që vështirë vërehen, si dhe vështirëson leximin e kodit dhe ndjekjen e ecuresë së programit.

1.6 Fajlli Header

Sipas konventës në gjuhën C, simbolet, tipet e të dhënave dhe emërtimi i jashtëm i funksioneve të librarisë, ruhen në fajllin e quajtur "header" (lexo hedër); kjo është fajlli me prapashitesën *.h. Në fillim të fajllave do të vëreni direktivat e paraprocessorit si:

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

```
/* dhe */
```

```
#include "lista.h"
```

Emri i fajllit header shkruhet në thonjëza, nëse fajlli gjendet në direktoriumin privat ku programi është duke u kompajluar. Kurse në mes të shenjave < > shkruhet emri i fajllit header, nëse fajlli gjendet në direktoriumin standard të fajllave header (fajllat që i ofron kompajleri).

Në programin e paraqitur në figurën 1.2 kemi përdorë fajllin <stdio.h> e cila definon prototipet e funksioneve të librarisë standarde për futjen dhe shtypjen e të dhënave. Fajllat e tjera shërbejnë për definimin e prototipeve të funksioneve, variablaive globale ose makrove të ndryshme. P.sh. fajlli ctype.h ofron definimin e prototipeve të funksioneve që përdoren për caktimin e tipit të simboleve, ta zëmë, funksioni isdigit() i cili përcakton nëse simboli është shifër apo gërmë. Ngjashëm është edhe me fajllin string.h e cila ofron funksione për vargje simbolesh (frazë ose fjali), ta zëmë funksioni strlen() që përdoret për llogaritjen e gjatësisë së vargut të simboleve.

1.6.1 Libraria Standarde për futjen dhe shtypjen e të dhënave

Për përdorimin e librarisë standarde ose të ndonjë librerie tjetër, rekomandohet përdorimi i direktivës include të priprocessorit (paraprocessorit). Në këtë mënyrë përfshihet fajlli i librarisë, e cila përmban definimet e prototipeve të funksioneve të implementuara në librari. P.sh. për përdorimin e librarisë standarde për futjen dhe shtypjen e të dhënave përdoret udhëzimi:

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

Disa prej funksioneve të kësaj librerie janë:

a) Funkzioni printf

Funksioni printf përdoret për shtypjen e të dhënave në formate të ndryshme. Ky funksion merr si parametër formal vargje simbolesh (frazja ose fjali) që shpjegojnë formatin për shtypje, si dhe bllokun e variablave për shtypje, p.sh:

```
char  gjinia = 'm';
int   mosha = 22 ;
long  int rroga = 42500L ;
float pesha = 84.55;

printf("gjinia = %c   vjet = %d", gjinia, mosha);
printf("   rroga = %ld   pesha = %f \n", rroga, pesha);
```

rezultati që do ta shihni në monitor do të jetë:

```
gjinia = m   mosha = 22   rroga = 32500   pesha = 84.55
```

Shenja e përqindjes % në funksionin printf përdoret për të treguar se shkronja mbas shenjës '%' është shkronjë formati. Disa nga shkronjat e formatit janë:

c	për gurmë
d	për numër decimal - integjer
o	për numër oktal
x	për numër heksadecimal
u	për numër natyral
hd	për numër decimal të vogël
ld	për numër decimal të madh
f	për numër real
lf	për numër real me precizitet të dyfishtë
e	për numër real në formë të shkurtuar (psh: 3E+3)
le	njëlloj si e, por me precizitet të dyfishtë
s	për varg simbolesh (frazja ose fjali)

Shkronjat e formatit mund të paraprihen me numër që cakton hapësirën që do ta zënë në fjali, kurse numri mbas pikës dhjetore (para shkronjës së formatit) përcakton numrin e shifrave që do të shfaqen mbas pikës dhjetore për numra realë:

```
char  gjinia = 'm';
int   mosha = 22 ;
float pesha = 84.55;

printf("gjinia = ***%3c***   mosha = ***%5d***",
      gjinia, mosha);
printf("pesha = ***%5.3f***, pesha);
```

Rezultati që do ta shihni në monitor do të jetë:

```
gjinia = *** m*** mosha = *** 22*** pesha = *** 84.550***
```

b) Funkzioni putchar

Ky funksion përdoret për të shtypur një simbol dhe merr si parametër konstanta të tipit `char` apo variabla të tipit `char`, p.sh.:

```
putchar('m'); /* shtyp shkronjën m */
putchar(gjinia); /* shtyp shkronjën që mban si vlerë variabla
                  gjinia e tipit char */
putchar(65); /* shtyp shkronjën A, e 65-ta në kodin ASCII */
putchar('\\155'); /* shtyp shkronjën e dhënë në numër oktal -
                  shkronjën m */
```

Dy komandat e fundit (ku si parametër kemi numrat) i kemi përdorë vetëm për ilustrim, dhe ju rekomandojmë që t'i shmangeni përdorimit të numrit si parametër. Përdorimi i tyre mund të rezultojë në gabime, dhe njëherësh, e bën kodin e programit të pallexueshëm. Përdorni parametra të kuptueshëm sikundër janë dy të parët.

c) Funkzioni scanf

Përdoret për futjen e formatuar të të dhënave. Ky funksion ka sintaksë të ngjashme me funksionin `printf`, mirëpo në të variablat duhen të paraprihen me shenjë '&', për arsye se funksioni `scanf` merr si parametër adresën e variablës. Kjo do të bëhet më e qartë kur t'i shpjegojmë treguesit (shih kapitullin 4). P.sh.

```
printf ("Shtype numrin e vjetëve :");
scanf ("%d", &vjet);
```

Variablat e përdorura me funksionin `scanf`, patjetër duhet të paraprihen me shkronjën `&`, përndryshe programi nuk do të funksionojë në rregull. Vini re se asnjëri simbol i shkruar në fjalinë e formatuar përveç shkronjës së formatit, nuk paraqitet në monitor, kështu që:

```
scanf ("Shtype numrin e vjetëve : %d", &vjet);
```

nuk është e ngjashme me komandat e mëparshme.

1.7 Tipet e definuara nga përdoruesi

Në gjuhën C tipet e reja mund të definohen me komandën `typedef` dhe `enum`. Komanda `typedef` përdoret zakonisht me komandën `struct` për definimin e tipeve të reja, kurse `enum` përdoret për të definuar tipe me vlerat e mundshme të renditura njëra pas tjetrës, p.sh.:

```
enum DITA_E_JAVES
{
    E_HENE,
    E_MARTE,
    E_MERKURE,
    E_EJTE,
    E_PREMTE
};
```

Kodi i mësipërm definon tipin `DITA_E_JAVES` me pesë vlera të mundshme. Këto vlera ruhen si integjerë. Në këtë rast, variabla e tipit `DITA_E_JAVES` mund t'i ketë vlerat prej 0 deri 4 (ku `E_HENE` e ka vlerën 0, `E_MARTE` e ka vlerën 1 e kështu me radhë). Mirëpo këtë renditje të vlerave mund ta ndërroni nëse i përcaktoni vlerat kështu:

```
enum SHKRONJA_SPECIALE
{
    RRESHTI_RI = '\n' ,
    TAB = '\t'
};
```

Në këtë rast vlerat e brendshme të tipit `SHKRONJA_SPECIALE` do të jenë: `RRESHTI_RI` i cili do ta ketë vlerën 10 (pra jo zero) që përfaqëson simbolin "Enter" (rreshti i ri) në kodin ASCII, kurse `TAB` do ta ketë vlerën 9 (jo 1 a 11, nëse e marim parasysh vlerën e anëtarit të mëparshëm). Këto tipe të reja mund të përdoren si konstanta apo të definojnë variabël, p.sh.:

```
enum SHKRONJA_SPECIALE shk;
```

apo si konstanta

```
putchar (RRESHTI_RI);
```

Gjuha C nuk ka variabël të tipit boolean, sikundër gjuhët Pascal e Modula 2. Mirëpo këtë tip mund ta definoni vetë:

```
enum boolean { FALSE , TRUE };
```

Në kapitujt e ardhshëm do të shihni se si gjuha C++ është më e pasur në shprehje dhe metoda për krijimin e tipeve të reja, të cilat janë të ngjashme me tipet e furnizuara nga kompajleri.

1.8 Komandat e priprocesorit

Priprocesori (paraprocessori) i gjuhëve C dhe C++ është pjesë e pandashme e kompajlerit, i cili ekzekutohet para procesit të kompajlimit të programit. Komandat e priprocesorit fillojnë me simbolin '#' në kolonën e parë të rreshtit. Disa nga komandat janë:

```
#include < emri i fajllit > /*përfshin përbërjen e fajllit në këtë pozitë.
                             Fajlli është i sistemit, i ruajtur zakonisht në
                             direktoriun /include */

#include "emri i fajllit" /*përfshin përbërjen e fajllit në këtë pozitë.
                           Fajlli mund të jetë në cilindo direktorium,
                           dhe të jetë çfarëdo fajlli (mirëpo zakonisht
                           është fajll header). */

#define SIMBOLI VLERA /*Definon simbolin e ri SIMBOLI me vlerën
                       VLERA. Kjo lehtëson lexueshmërinë e
                       programit dhe ndërrimin e vlerave të
                       simboleve */

#undef SIMBOLI /*Eliminon definimin e SIMBOLI-t */

#include <stdio.h>

#define MAXNO 30 /* definimi i maksimumit të numrit */
#define katrori(x) ((x) * (x)) /* definimi i makros katrori
                                e cila llogarit katrorin
                                e me brinjën x */

void main (void)
{
    int brinja

    do{
        printf("Shtype një numër më të vogël se %d \n", MAXNO);
        printf("Për përfundimin e programit shtyp numrin 0 " );

        scanf("%d",&brinja);
        printf("Katrori me brinjën %d ka syprinën %d \n",
               brinja, katrori(brinja));
```

```

    }while(brinja != 0);
}

```

Programi 1.8.1

Para kompajlimit të programit, të gjitha udhëzimet në #define ndërrohen nga priprocesori në vlera si në programin 1.8.2.

[Deklarimet nga fajlli stdio.h]

```

void main (void)
{
    int brinja
    do{
        printf("Shtype një numër më të vogël se %d \n",30);
        printf("Për përfundimin e programit shtyp numrin 0 " );
        scanf("%d",&brinja);
        printf("Katrori me brinjën %d ka syprinën %d \n",
                brinja,((brinja)*(brinja)));
    }while(brinja != 0);
}

```

Programi 1.8.2

Në këtë shembull komanda #define përdoret për dy qëllime. #define MAXNO definon konstantën MAXNO, kurse #define katrori ((x) * (x)) definon makron për ta llogaritur syprinën e katrorit të brinjës së dhënë. Vëreni se makroja MAXNO përdoret jashtë thonjzave në funksionin printf, sepse makrot nuk ndërrohen nga priprocesori nëse janë brenda në thonjzë. Priprocesori e zëvendëson makron dhe parametrat e makrove në kod. Makrot, të cilat definojnë funksionet, janë të ngjashme me funksionet normale. Këto makro duhet të definohen vetëm nëse funksioni i cili do të ekzekutohet është i vogël (zakonisht funksionet jo më të gjata se një rresht), përndryshe duhet të përdoren funksionet normale.

Duhet të keni kujdes gjatë definimit të makrove, p.sh. nëse makroja katrori definohet si:

```
#define katrori(no)    no * no
```

dhe nëse thirret më vonë si:

```
s = katrori(i-1);
```

atëherë makroja do të ndërrohet nga priprocesori në:

```
s = i - i + 1;
```

Kodi i mësipërm do të jetë i njëjtë me $(i+1) * (i+1)$, kështu që rezultati përfundimtar nuk do të jetë i saktë:

$i+1 * i+1 = 2i + 1$, kurse $(i+1)*(i+1) = (i+1)^2 = i^2 + 2i + 1$.

Po ashtu duhet të keni parasysh se makrot nuk do të duhej të marrin si parametër variablat me auto-operatorë, p.sh.:

```
s = katrori(++i);
```

Kodi i mësipërm interpretohet si $s = ((++i) * (++i))$. Nëse $i = 2$ atëherë rezultati do të ishte $3 * 4$ e jo $3 * 3$.

Duhet pasur parasysh se makrot mund të kenë efekt të padëshirueshëm në program. Ta zëmë p.sh. se kemi një funksion Numri i cili kthen tipin int:

```
int Numri ( );
```

Për kompajlerin, integrimi i këtij funksioni në makron katrori do të ishte krejtësisht i pranueshëm:

```
katrori(Numri());
```

Edhe njëherë po përsërisim se kodi i mësipërm shndërrohet në:

```
Numri() * Numri();
```

Në rastin e mësipërm kjo do të ishte krejtësisht e pranueshme, mirëpo ka raste kur funksioni nuk duhet të thirret më shumë se njëherë (normalisht pa dijen e programerit), si p.sh. funksioni i cili tërheq paratë prej kontos dhe kthen shumën e tërhequr. Në raste të këtilla shumë lehtë mund të bëjmë gabime, të cilat vështirë dallohen.

1.9 Tipi Array (radha)

Tipi *array* (lexo: ërej) nuk është tip i ri. "Array" në anglisht do të thotë radhitje, renditje, rreshtim. Me *array* pra kuptojmë radhitje të njëpasnjëshme të tipeve të paradefinuara në memorie. Deri më tani kemi përdorë variabla të cilat kanë mbajtur vetëm një vlerë, siç janë:

```
int    i;
char   c;
float  rroga;
```

Të gjitha këto variabla mund të mbajnë vetëm një vlerë në çfarëdo kohe të ekzekutimit të programit, pra nuk mund të mbajnë më shumë se një vlerë në të njëjtën kohë.

Le të supozojmë se në program na duhet të numrojmë sa herë paraqitet çdo germë e alfabetit në ndonjë fjali (string). Një prej zgjidhjeve do të ishte t'i deklaranim 27 variabla¹ të tipit int (int a_var, b_var, c_var... etj) ku secila variabël do ta mbante numrin e paraqitjes së një germe të dhënë në fjali. Kodi për këtë program do të ishte i gjatë dhe më pak i lexueshëm:

```
char c;
unsigned int a_var = 0,
             b_var = 0,
             c_var = 0... , /* dhe kështu deri te: */
             z_var = 0;

while ((c = getchar( )) != EOF)
{
    switch (c)
    {
        case 'a':  a_var++;
                   break;
        case 'b':  b_var++;
                   break;
        case 'c':  c_var++;
                   break;
        :
        :
        case 'z' :  z_var++;
                   break;
    }
}
```

¹ duhet pasur parasysh se gerrat dh. gj, ll, nj, rr, sh, th, xh, zh në programim njihen si dy germa.

E meta kryesore e këtij programi është se duhet të deklarojmë numër të madh të variablave. Do të ishte më e përshtatshme sikur të kishim vetëm një variabël e cila do të kishte 27 lokacione të ndryshme për të ruajtur numruesin e çdo germe. Kjo metodë bëhet e mundur me përdorimin e variablës së veçantë për ruajtjen e një vargu të dhënave të quajtur *array*. Një *array* mund të ketë një apo më shumë anëtarë të tipit të definuar paraprakisht, të cilët anëtarë janë të renditur, të grupuar dhe të njohur me një emër të vetëm:

```
int   alfabeti[27];    /* definon një array me 27 integjerë */
char  emri[10];        /* definon një array me 10 germa */
float koordinatat[6];  /* definon një array me 6 numra realë */
```

Të gjithë anëtarët e një variabël *array* duhen të jenë të të njëjtit tip. Sintaksa e definimit të variablës *array* është e ngjashme me atë të definimit të variablave të thjeshta, përveçse emri i variablës *array* është i përcjellë me një numër brenda kllapave katrore. Në gjuhën C numrimi i anëtarëve të radhitur fillon nga zeroja, përkundër disa gjuhëve të tjera (si Pascal, Modula2 etj) ku programeri mund të specifikojë vetë nga cili numër të fillojë numrimi i anëtarëve. P.sh. për të përdorë vlerat e anëtarëve të variablës *alfabeti*, duhet pasur parasysh se vlera e parë e saj është *alfabeti[0]* dhe e fundit *alfabeti[26]*:

```
alfabeti[0]    /* anëtari i parë i variablës alfabeti */
alfabeti[1]    /* anëtari i dytë i variablës alfabeti */
:
:
alfabeti[26]   /* anëtari i 27-të i variablës alfabeti */
```

Në kodin që vijon kemi paraqitur njërën prej mënyrave se si përdoret tipi *array*. Në këtë shembull kemi përdorë disa udhëzime që ende nuk i kemi shpjeguar, mirëpo, keni durim, sepse do t'i shpjegojmë në kapitujt e ardhshëm.

```
#include <stdio.h>

#define ANETAR 26      /* defino numrin e anëtarëve */

void numro_germat(const char *fjalja, int *alf)
{
    int i;              /* indeksi për kontroll të anëtarëve në array */
    char c;

    for ( i = 0; fjalja[i] != NULL ; i++)
    {
        c = fjalja[i] ;
        /* nëse është germë e madhe e shtypit */
```

```

        if ((c < 91) && (c > 64)
            alf[c-65]++;
        else if ((c > 96) && (c < 123)) /* germë e vogël */
            alf[c-97] ++ ;
    }

}

void main (void)
{
    int alfabeti[ANETAR];
    int i ;
    char fjalia[] = "Mësim, mësim dhe vetëm mësim" ;

    for ( i = 0; i < ANETAR; i++)
        alfabeti[i] = 0;

    numro_germat(fjalia,alfabeti) ;

    for ( i = 0; i < ANETAR; i++)
        printf("Germa %c është paraqitur %d herë\n",i+97,
            alfabeti[i]);
}

```

Numri i anëtarëve të ndonjë variable të tipit *array* nuk mund të jetë numër negativ. Në gjuhën C tipi më i përdorur me *array* është tipi *char* për paraqitjen e një blloku simbolesh (*string*). Për ta ilustruar përdorimin e variablës *array* të tipit *char*, shih programin i cili lexon disa rreshta dhe shtyp rreshtin më të gjatë.

Koncepti i thjeshtë i programin në pseudokod² do të ishte

```

përderisa ( ende ka rreshta për t'u lexuar)
    nëse (rreshti i lexuar është më i madh
        sesa rreshti i mëparshëm)
        ruaje këtë rresht dhe gjatësinë e tij
    shtype rreshtin më të gjatë

```

Fjala *përderisa* në C do të ishte *while* dhe *nëse* do të ishte *if*. Ky koncept bën që programi të ndahet në pjesë, kështu që një pjesë do ta lexonte rreshtin, tjetra do ta testonte, tjetra do ta ruante rreshtin dhe pjesa e mbetur do ta kontrollonte procesin.

Në bazë të konceptit në pseudokod, na duhet funksioni *lexo_rreshtin* që ta lexojë rreshtin. Do të ishte e udhës që ky funksion të kthejë vlerën 0 për të sinjalizuar përfundimin e fajllit, ose një numër "jo-zero" për ta sinjalizuar

² Pseudokod quajmë shpjegimin e kodit të programit në shqip (jo në gjuhë programuese).

gjatësinë e rreshtit të lexuar. Kur ta gjejmë rreshtin më të gjatë sesa rreshti i mëparshëm, ky i fundit do të ruhet (kopjohet). Për këtë na duhet funksion kopjo. Më në fund na duhet programi kryesor që t'i kontrollojë funksionet lexo_rreshtin dhe kopjo.

```

/* programi për të lexuar disa rreshta dhe për të shtypur
   rreshtin më të gjatë */

/* maksimumi i gjatësisë së rreshtit */
#define MAXRRESHTI 500

/* deklarimi i prototipit të funksioneve */
int lexo_rreshtin( char *, int);
void kopjo(char*, char *);

void main (void )
{
    int gjatesia; /* gjatësia e rreshtit të lexuar */
    int max; /* gjatësia e rreshtit më të gjatë
               të lexuar */
    char rreshti[MAXRRESHTI] ; /* rreshti i lexuar */

    /* rreshti më i gjatë i ruajtur */
    char i_ruajtur[MAXRRESHTI] ;

    max = 0 ;

    while ((gjatesia = getline(rreshti, MAXRRESHTI)) > 0)
        if ( gjatesia > max )
        {
            max = gjatesia;
            kopjo(rreshti, i_ruajtur);
        }

    if ( max > 0 ) /* nëse është lexuar diçka */
        printf("%s", i_ruajtur);
}

/* funksioni që lexon rreshtin dhe kthen gjatësinë e tij: */
int lexo_rreshtin(char s[ ], int m_g )
{
    int c , i ;

    /* lexoji germet derisa të arrijmë në numrin maksimal
       të germave (m_g), ose germa e shtypur është fundi i
       fajllit, ose rresht i ri */

    for ( i = 0 ; (i < m_g - 1)
          && (c = getchar( )) != EOF

```

```

        && (c != '\n') ; ++i)
    {
        s[i] = c ;
    }

    if ( c == '\n')
    {
        s[i] = c ;
        ++i ;
    }

    s[i] = '\0' ;
    return i ;
}

/* funksioni që kopjon nga një string (varg) në tjetrin */
void kopjo(char s1[ ], char s2[ ])
{
    int i ;

    for ( i = 0 ; (s1[i] = s2[i]) != '\0' ; ++i )
        ;
}

```

Programi 1.9.1

Parametrat e funksionit `lexo_rreshtin` janë *array* për tipin `char` dhe `int`. Funksioni `lexo_rreshtin` përdor komandën `return` për t'ia kthyer vlerën mbresa thirrësit të këtij funksioni. Kjo vlerë paraqet gjatësinë e rreshtit të lexuar. Para se ta kthejë gjatësinë, ky funksion e përfundon rreshtin (*stringun*) me shkronjën `'\0'` sikurse kompajleri i C-së. Funksioni `kopjo` përdoret për kopjimin e të dhënave prej objektit *array* `s2` në objektin *array* `s1`.

Ushtrime

1. Shkruani një program që shtyp numrin 10.
2. Analizojeni programin në figurën 1.1 dhe shkruani një program të ngjashëm që shtyp frazën "Lahuta e Malcís!".
3. Shkruani programin që deklaron tri variabla të tipit `int` (`x`, `y`, `z`). Inicoi dy variablat e para me vlerat 5 dhe 12 (`x`, `y`), pastaj inicojeni variablën e tretë (`z`) me shumën e variablave të para (`z = x + y;`). Në fund shtypeni vlerën e variablës `z`.
4. Shkruani një program për gjetjen e mbetjes së numrave 20 dhe 6 të pjesëtuar me numrin 3 (përdorni operatorin `%`).
5. Provoni ta shkruani programin që shtyp numrat prej 0 deri në 50. Përdorni operatorin auto-rritës (`++`).
6. Deklaroni një variabël të tipit `array` që përmban 5 fjali dhe pastaj shtypini fjalitë me radhë me anë të funksionit `printf`.
7. Shkruani programin që lexon fjalinë e shtypur nga përdoruesit e programit dhe i numëron zanoret në atë fjalë. Në fund lajmërojeni shfrytëzuesin e programit për numrin e zanoreve të gjetura në fjalë.
8. Tentoni të ndryshoni funksionin `kopjo` që në vend të tipit `array` të përdorë tregues të tipit `char` (`char*`). Ndryshoni tipin e parametrave të funksionit `kopjo` prej `char s1[]` dhe `char s2[]` në `char* s1` dhe `char* s2` dhe verëni ndryshimet pas kompajlimit të programit.
9. Shkruani programin që definon dy makro të ndryshme dhe pastaj provoni të shtypni vlerën e këtyre dy makrove me anë të funksionit `printf`.
10. Definoni tipin `enum` `FUNDJAVA` që përmban konstanta për dy ditët e fundit të javës.
11. Shkruani programin që shtyp vlerat e konstantave të tipit `enum` `FUNDJAVA` të definuar në ushtrimin 10.
12. Deklaroni një variabël të tipit `array` për të mbajtur 10 numra integjerë. Provoni ta shkruani kodin që ta inicojë çdo indeks në `array` me ndonjë vlerë dhe pastaj të shtypë secilën vlerë.

Përmbledhje

Gjuha C është një ndër gjuhët më të përdorura në programim. Kjo gjuhë kombinon përparësitë e gjuhës së nivelit të lartë me efikasitetin e gjuhës Assembler.

Tipet elementare për ruajtjen e të dhënave që gjuha C i ofron janë:

`char, int, float dhe double`

Më anë të këtyre tipeve mund të krijojmë tipe më të komplikuar për ruajtjen e të dhënave në përshtatje me problemet specifike.

Gjuha C ofron librari standarde që ngërthejnë funksione për shtypjen dhe leximin e të dhënave. Edhe pse kemi mundësi t'i shkruajmë vetë këto funksione të implementuara në libraritë standarde, rekomandohet që ato të përdoren për arsye se janë të testuara, prandaj mundësia për gabime është më e vogël. Implementimi i këtyre librarive standarde është i fshehur zakonisht në fajllat binarë *.lib, mirëpo për përdorimin e këtyre librarive, gjuha C furnizon fajllat header që definojnë prototipet e funksioneve, variablat globale si dhe makrot e ndryshme.

Vendimet dhe vorbullat

Vendimet në C merren me anë të komandave `if` dhe `switch`. Këto dy komanda do të jenë ndër kryefjalët e këtij kapitulli. Shprehjet boolean përcaktojnë vendimin e komandës `if`, kurse vlerat int përcaktojnë vendimin e komandës `switch`.

2.1 Komanda If-else

Gjatë të shkruarit të kodit të programit, shpesh duhet të vendosim për kahjet (opcionet) e ekzekutimit të programit. Kjo gjë bëhet përmes paraqitjes së kushteve dhe dhënies së udhëzimeve se cilat kahje duhen ndjekur në faza të ndryshme të ekzekutimit të programit. Gjuha C ofron këto mundësi për t'i shkruar kushtet e tilla:

- ♦ `if-else` komanda që përdoret për të zgjedhur ndërmjet dy kushteve, dhe
- ♦ `switch` komanda për zgjedhje në rast të më shumë se dy kushteve.

Sintaksa e komandës përzgjedhëse `if-else` është kështu:

```
if ( kushti )  
    udhëzimi 1;  
else  
    udhëzimi 2;
```

Udhëzimi 1 ekzekutohet nëse ekuacioni kushtor (kushti) jep rezultatin e saktë (jo-zero), në të kundërtën ekzekutohet udhëzimi 2, pra nëse ekuacioni kushtor jep rezultat jo të saktë (ekuacioni është baraz me zero).

Në programin më poshtë lexojmë numrin e shtypur nga përdoruesi për moshën e tij, dhe nëse numri i shtypur është baras me ose më i madh se 18, atëherë shtypim fjalinë "Personi mund të votojë". Në të kundërtën, shtypim fjalinë "Personi është i ri për votim".:

```
#include <stdio.h>

#define MOSHA_PER_VOTIM 18

void main (void)
{
    int moshë;

    printf("Shtypni moshën e personit: ");

    scanf("%d", &mosha); /* lexo numrin e shtypur */

    if ( mosha >= MOSHA_PER_VOTIM )
        printf("Personi mund të votojë.");
    else
        printf("Personi është i ri për të votuar.");
}
```

Mund të ndodhë që udhëzimi 2-të mungojë fare. Atëherë komanda do të dukej kështu:

```
if ( kushti )
    udhëzimi
```

Në këtë rast, duke qenë se kushti if-else nuk është i plotë, programi nuk do ta ekzekutojë asnjë udhëzim nëse kushti nuk është i saktë. Kjo është plotësisht e arsyeshme, për shkak se shpesh kemi raste kur në program duam ta ekzekutojmë një bllok të udhëzimeve vetëm nëse plotësohet një kusht i dhënë, përndryshe nuk duam të ekzekutojmë asgjë.

Po ashtu është e mundshme që udhëzimi 1 ose udhëzimi 2 të jenë të përbërë prej komandave seleksionuese apo prej vorbullave (togje përsëritëse të udhëzimeve) p.sh.:

```
if (kushti 1)
{
    if (kushti 2)
        udhëzimi 2a
    else
        udhëzimi 2b
}
else
```

```

{
    udhëzimi i kushtit 1
}

```

Udhëzimet në komandat seleksionuese edhe vetë mund të jenë komanda seleksionuese në disa nivele më poshtë sesa kushti i parë, psh:

```

if (kushti 1)
{
    if (kushti 2)
        if (kushti 3)
            if (kushti 4)
                if (kushti 5)
                    udhëzimi A
            else
                if (kushti 6)
                    udhëzimi B
        }
    else
    {
        udhëzimi i kushtit 1
    }
}

```

Mirëpo kjo gjë nuk preferohet, sepse shkakton vështirësi në leximin e programit, si dhe është ndër pjesët ku programerët më së shpeshti gabojnë, duke menduar se një bllok udhëzimesh do të ekzekutohej pas plotësimit apo mosplotësimit të një kushti.

P.sh. në kodin e mësipërm, pjesa e kodit:

```

if (kushti 6)
    udhëzimi B

```

duket se do të ekzekutohej nëse kushti 2 nuk plotësohet. Kështu me siguri do të mendonte ai që do ta shkruante këtë program, për të vënë re gjatë ekzekutimit të programit, se programi nuk e kryen punën si duhet. Gjatë leximit të kodit, shumica do të ishin të bindur se nuk është bërë asnjë gabim në kod dhe se kompajleri apo sistemi operativ janë shkaktarë që programi nuk punon si duhet. Kjo nuk është e drejtë sepse blloku i udhëzimeve:

```

if (kushti 6)
    udhëzimi B

```

do të ekzekutohej vetëm nëse kushti i pestë nuk plotësohet. Për t'i ikur këtij problemi dhe problemeve të tjera me komandat përsëritëse, duhet t'i përdorni

klapat e mëdha {} që ta definoni mirë bllokun e kodit dhe t'i veçoni mirë udhëzimet. P.sh. nëse keni pasur ndër mend që kodi i mësipërm të ekzekutohet nëse kushti 2 nuk plotësohet, atëherë ky kod do të ishte në rregull:

```
if (kushti 1)
{
    if (kushti 2)
    {
        if (kushti 3)
            if (kushti 4)
                if (kushti 5)
                    udhëzimi A
            }
        else
        {
            if (kushti 6)
                udhëzimi B
            }
        }
    else
    {
        udhëzimi i kushtit 1
    }
}
```

Kini kujdes, sepse krahasimi i dy shprehjeve apo variablave bëhet me operatorin e relacioneve == (jo vetëm me një =). Nëse e harroni një operator = kompajleri nuk do të ankohet, mirëpo programi nuk do të jetë i saktë. Disa kompajlerë të gjuhës C++ lajmërojnë për këtë gabim (duke e paraqitur si vërejtje - warning e jo si gabim -error).

2.1.1 Operatori ternar

Ekuacionet kushtëzore me trajtën e mëposhtme:

```
if ( numri == MAX)
    numri = 1;
else
    numri++;
```

janë shumë të shpeshta dhe mund të zëvendësohen me operatorin ternar që përbëhet prej tri pjesëve (prandaj edhe emërtimi *ternar*): kushti, pjesa e saktë dhe pjesa e pasaktë.

Psh:

```
kushti ? pjesa_esaktë : pjesa _epasaktë;
```

Kodi i mësipërm i cili përdor shprehjen `if-else` do të paraqitet kështu me operatorin ternar:

```
numri = (numri == MAX) ? 1 : numri++;
```

Në operacionin kushtëzor së pari shqyrtohet kushti dhe nëse është e saktë vlera e ekuacionit, programi ekzekuton pjesën e saktë (`pjesa_esaktë`), përndryshe programi ekzekuton pjesën e pasaktë (`pjesa_epasaktë`).

Operatori ternar zakonisht përdoret kur udhëzimet për ekzekutim janë të shkurtëra (siç është shembulli i mësipërm). Nëse udhëzimet janë të gjata ose kemi një sërë udhëzimesh për ekzekutim, që mvaren nga (mos)plotësimi i kushtit, atëherë preferohet të përdoret komanda `if-else`, si në shembullin e mëposhtëm:

```
if (numri == MAX)
{
    numri = 1;
    printf(" Numri është i barabartë me %d\n", numri);
}
else
{
    numri++;
}
```

2.2 Operatorët logjikë

Në gjuhën C kemi këta operatorë logjikë:

Në matematikë

\wedge
 \vee
 $!$

Në gjuhën C dhe C++

`&&`
`||`
`!`

DHE
OSE
JO

Programin në vazhdim mund ta plotësojmë duke i përjashtuar personat shumë të vjetër apo shumë të rinj për votim:

```
#include <stdio.h>

#define MOSHA_PER_VOTIM 18
#define SHUME_I_VJETER 65
```

```

void main (void)
{
    int  mosha;

    printf("Shtypni moshën e personit: ");
    scanf("%d", &mosha);

    if (mosha >= MOSHA_PER_VOTIM && mosha < SHUME_I_VJETER)
        printf("Personi mund të votojë.");
    else
        printf("Personi nuk mund të votojë.");
}

```

Është me rëndësi ta keni parasysh se gjuha C në ekuacionet e relacioneve garanton renditjen e llogaritjes së ekuacionit prej të majtës në të djathtë dhe llogaritja ndalet menjëherë nëse rezultati i tërë ekuacionit dihet. Në shembullin e mëparshëm, nëse kushti `mosha >= MOSHA_PER_VOTIM` nuk plotësohet (vlera e kthyer është 0), llogaritja e kushtit tjetër `mosha < SHUME_I_VJETER` nuk bëhet fare, për arsye se rezultati i saj nuk ndikon në rezultatin e tërë ekuacionit. Arsyja është se, në rregullat e logjikës: `pasaktë && (çfarëdo rezultati: saktë apo pasaktë)`

do të japë rezultat të pasaktë. D.m.th. menjëherë pas llogaritjes së rezultatit të udhëzimit të parë, duke e pasur parasysh se kemi të bejmë me operatorin `&&` ("edhe") kompajleri do ta dijë vlerën rezultuese, kështu që nuk do t'i llogarisë udhëzimet e tjera. Në qoftë se në vend të operatorit `&&` do ta kishim operatorin `||` ("ose"), atëherë kompajleri do ta llogariste edhe udhëzimin e dytë, sepse rezultati i plotë do të mvarej edhe nga udhëzimi i dytë.

2.3 Komanda switch

Gjuha C përdor komandën `switch` për të vendosur në bazë të vlerës (`int`) së një udhëzimi a variable.

Sintaksa e komandës `switch` është:

```

switch ( <udhëzimi> )
{
    case < vlera 1 >: < deklarimi 1 >; break;
    case < vlera 2 >: < deklarimi 2 >; break;
    ...
    case < vlera i >: < deklarimi i >; break;
    default: <deklarimi (i -1)>
}

```

Vlera e <udhëzimit> përcakton në cilin <deklarim> fillon ekzekutimi. Pasi të fillojë ekzekutimi, programi do të ekzekutohet përderisa nuk arrihet fundi i komandës switch apo komanda break.

P.sh. në rast të mungesës së komandës break pas deklarimit 1:

```
switch ( <shprehja> )
{
    case < vlera 1 >: < deklarimi 1 >;
    case < vlera 2 >: < deklarimi 2 >; break;
    default: <deklarimi (i +1)>
}
```

nëse shprehja është e barabartë me vlerën 1 atëherë të dy deklarimet 1 dhe 2 do të ekzekutoheshin. Pra keni kujdes gjatë përdorimit të komandës switch pa komandën break.

Shihni në shembullin në vazhdim se si e nxjerrim frazën që na duhet, në bazë të vlerës që ka variabla d:

```
enum ditaejaves {E_hene, E_marte, E_merkure,
                  E_enjte, E_premte,
                  E_shtune, E_diel};
```

```
ditaejaves d;
```

```
...
switch (d) {
    case E_hene:
        printf( "E hene" );
        break ;
    case E_marte:
        printf("E marte");
        break ;
    case E_merkure:
        printf("E merkure");
        break ;
    case E_enjte:
        printf("E enjte" );
        break ;
    case E_premte:
        printf("E premtë" ) ;
        break ;
    case E_shtune:
        printf("E shtune" ) ;
        break ;
    case E_diel:
        printf("E diel" ) ;
        break ;
}
```

}

Vlera kontrolluese e komandës `switch` duhet të jetë int, mirëpo mund të përdorim edhe tipin `enum`, sepse vlera e brendshme e këtij tipi është int.

Në shembullin që vijon kemi përdorë disa vlera për ta ekzekutuar të njëjtin deklaram. P.sh. nëse vlera e variablës `d` është `E_hene`, `E_marte` ose `E_merkure`, programi do ta nxjerrë të njëjtën frazë (e njëjta gjë vlen edhe për vlerat e tjera):

```
switch (d )
{
    case E_hene:
    case E_marte:
    case E_merkure:
        printf("Sot ia filloj mësimin në ora 8:00\n");
        printf("Dhe mbaroj në ora 16:00\n");
        break;
    case E_enjte:
    case E_premte:
        printf("Sot ia filloj mësimin në ora 9:00\n");
        printf("Dhe mbaroj në ora 15:00\n");
        break;
    case E_shtune:
    case E_diel:
        printf("Sot jam në pushim");
}
```

Si shembull tjetër për komandën `switch` do të marrim përcaktimin e ditëve të çdo muaji të vitit. Në kodin që vijon kemi përcaktuar vlerën e variablës `numriiditeve` në bazë të muajit të vitit:

```
enum muaji {janar, shkurt, mars, prill, maj,
qershor, korrik, gusht, shtator,
tetor, nentor, dhjetor}

int    numriiditeve;
muaji m;

/* ndonjë kod i cili e ndryshon vlerën e variablës m */
...

switch (m)
{
    case shkurt:
        if ( ( (viti % 4) == 0 ) &&
            ( (viti % 100) != 0 ) ) ||
            ( (viti % 400) == 0 )
            numriiditeve = 29;
        else
```

```

        numriiditeve = 28;
        break;
case prill:
case qershor:
case shtator:
case nentor:
        numriiditeve = 30;
        break;
case janar:
case mars:
case maj:
case korrik:
case gusht:
case tetor:
case dhjetor:
        numriiditeve = 31;
        break;
}

```

Komanda `default` përdoret në `switch` për t'i ekzekutuar shprehjet nëse vlerat e mëparshme nuk e plotësojnë ndonjërin prej kushteve në shprehjen `switch`. Vëreni shembullin që vijon:

```

int i = 0;

switch (i)
{
    case 1:
        printf("Variabla (i) e ka vlerën 1\n");
        break;
    case 2:
        printf("Variabla (i) e ka vlerën 2\n");
        break;
    default:
        printf("Variabla (i) nuk është baras me 1 apo 2\n");
        break;
}

```

Ky është një shembull artificial sa për të ilustruar se si shprehjet në bllokun e kodit pas komandës `default` ekzekutohen nëse asnjë shprehje në `case` nuk e ka plotësuar kushtin për ekzekutim.

2.4 Vorbullat

I kemi quajtur kështu blloqet e udhëzimeve që përsëriten disa herë para se ekzekutimi i programit të vazhdojë poshtë tyre. Gjuha C ofron vorbulla të

cilat ndihmojnë në zvogëlimin e përsëritjes së udhëzimeve të njëjta (duplikimin e kodit) si dhe leximin më të lehtë të programit.

Gjuha C ofron tri vorbullat e mëposhtme:

- ♦ for
- ♦ while
- ♦ do-while.

Prej këtyre të trijave, vorbulla for është më e përdorshme.

2.4.1 Vorbulla for

Sintaksa e kësaj vorbulle është:

```
for (fillim-vlera; kushti ; udhëzimi-ndërrues)
    pjesa e programit që përsëritet;
```

Struktura for përmban tri udhëzime brenda kllapave (të ndara me pikëpresje ;) dhe pjesën e programit që përsëritet sa herë që plotësohet kushti për përsëritje. Udhëzimi fillim-vlera është udhëzimi ku caktohet vlera fillestare e variablës apo ekzekutohet ndonjë funksion në fillim të përsëritjes. Udhëzimi fillim-vlera ekzekutohet vetëm një herë gjatë hyrjes në vorbullën for. Udhëzimi udhëzimi-ndërrues ekzekutohet pas çdo përfundimi të ekzekutimit të kodit brenda vorbullës. Ndërkaq, udhëzimi kushti është udhëzim që llogaritet në hyrje të vorbullës for dhe pas llogaritjes të udhëzimit udhëzimi-ndërrues.

Duke qenë se kushti llogaritet në fillim të vorbullës for, blloku i udhëzimeve në kodin brenda vorbullës ndodh të mos ekzekutohet fare, nëse kushti nuk plotësohet. kushti mund ta ketë vlerën të saktë apo të pasaktë. Kjo vlerë zakonisht llogaritet duke përdorë operatorët e relacioneve:

Në matematikë	Në C dhe C++	
>	>	më e madhe
<	<	më e vogël
≥	>=	më e madhe ose barraz
≤	<=	më e vogël ose barraz
=	==	barraz
≠	!=	jo barraz

P.sh. kodi që vijon:

```
for (i=0 ; i < 5 ; i++)
    putchar('*');
```

nxjerr një varg prej pesë simboleve '*'.

Të tri udhëzimet në vorbullën for nuk janë të domosdoshme, pra mund të mos jenë prezente, mirëpo dy pikëpresjet ; duhen të jenë patjetër prezente.

Vorbulla for mund të përdoret për leximin e simboleve një nga një nga tastatura. P.sh. kodi që vijon e lexon një rresht të shkronjave nga tastatura, përderisa tastiera 'rreshti i ri' (Enter) nuk është shtypur.

```
int c ;
for ( c= getchar( ) ; c != '\n' ; c = getchar( ) )
    putchar(c);
```

Një mënyrë tjetër më kompakte e kësaj përsëritjeje, do të ishte kështu:

```
int c ;
for ( ; (c=getchar( )) != '\n' ; )
    putchar(c);
```

Në këtë rast funksioni getchar thirret së pari dhe simboli i lexuar ruhet në variablën c. Pastaj variabla c krahasohet me simbolin 'rreshti i ri'. Nëse variabla c nuk është e barabartë me simbolin 'rreshti i ri', kodi në trupin e vorbullës (në këtë rast putchar(c)) përsëritet prapë.

Kllapat rreth udhëzimit c = getchar() janë të domosdoshme. Kjo për shkak të përparësisë më të vogël të operatorit = në krahasim me operatorin e relacionit !=.

Në mungesë të kllapave, simboli i lexuar krahasohet me simbolin "\n" ('rreshtin e ri'). Nëse simboli i dhënë nuk është 'rreshti i ri', variabla c e merr vlerën e saktë; në rast të kundërt merr vlerë të pasaktë. Ky është gabimi, sepse variabla c është dashur ta merrte vlerën e simbolit të lexuar. Pra në mungesë të kllapave, llogaritja nga kompajleri bëhet kështu:

```
c = ( getchar ( ) != '\n' )
```

e barabartë me udhëzimin c = getchar () != '\n' por jo me udhëzimin

```
(c = getchar( )) != '\n'.
```

Në mungesë të kllapave rreth udhëzimit `c = getchar()`, kompajleri nuk do të ankohet për ndonjë gabim, mirëpo vorbulla `for` nuk do ta bëjë atë që mendoni se do ta bëjë.

Nëse vëreni shembullin për `array` në kapitullin 1 (programi 1.7.1), vorbullën `for` e kemi përdorë në këtë mënyrë:

```
metoda 1
for ( i = 0 ; (s1[i] = s2[i]) != '\0'; ++i )
    ;
/* në këtë rast mungon kodi në trup të vorbullës */
```

kjo mund të paraqitet edhe si:

```
for ( i = 0 ; (s1[i] = s2[i]) != '\0'; ++i )
{
}
```

Vëreni me kujdes, dhe do të shihni se anëtarët e variablës `array` `s1` e kopjojnë vlerën e anëtarëve të variablës `array` `s2` dhe në të njëjtën kohë testohen se mos anëtari me indeks (`i`) është i barabartë me `'\0'` për përfundimin e përsëritjes.

Vorbullën `for` në rastin e mësipërm mund ta paraqesim edhe kështu:

```
metoda 2
for ( i = 0 ; s2[i] != '\0'; ++i )
    s1[i] = s2[i];
```

edhe pse kjo metodë është më e lexueshme, metoda e mësipërme është më kompakte dhe me më pak kalkulime për të njëjtin problem (ndaj është më efikase). Kjo për arsye se në metodën 2, brenda vorbullës `for`, për të manipuluar me çdo anëtar të variablës `array` `s2`, adresa e memories llogaritet dy herë (`s2[i] != '\0'` dhe në `s1[i] = s2[i]`), përderisa në metodën 1 llogaritet vetëm një herë (`(s1[i]=s2[i]) != '\0'`).

Kini parasysh se në probleme të tjera metoda 1 nuk është e mundshme të përdoret, ose është shumë e vështirë të lexohet.

Në vorbullën `for` mund të përdoret operatori presje (`,`), i cili bën ndarjen e udhëzimeve në nënudhëzime. Secili udhëzim llogaritet prej të majtës në të djathtë. P.sh. në kodin në vijim në vorbullën `for`, në pjesën e udhëzimit `fillin-vlera` kemi inicuar dy variabla (pra dy nënudhëzime të ndara me presje) `x = 0`, `i = 0`:

```

int x , y , i ;

for ( x = 0 , i = 0 ; i < 25 ; i++ )
{
    scanf("%d",&y);
    x += y ;
}

```

2.4.2 Vorbullat while dhe do-while

Sintaksa e kësaj vorbulle është:

```

while ( kushti )
    <udhëzimi>

```

Nëse <udhëzimi> është i përbërë prej më shumë se një udhëzimi ekzekutues, atëherë <udhëzimi> duhet të jetë i rrethuar me kllapa gjarpërore {}. Në strukturën while, <udhëzimi> ekzekutohet sa kohë që kushti plotësohet. Nëse nuk jeni të kujdesshëm gjatë kodimit të programit dhe kushti nuk bëhet asnjëherë i plotësueshëm, atëherë ekzekutimi i kësaj vorbulle nuk përfundon (kodi do të ekzekutohet pakufi herë). E njëjta gjë ndodh edhe me vorbullat e tjera (for dhe do-while).

Pa përdorimin e vorbullave, kodi është më i gjatë dhe më pak i lexueshëm, edhe pse kryen të njëjtën punë sikurse kodi që përdor vorbullat. P.sh. programi për të nxjerrë numrat prej 1 deri në 10, si dhe fuqinë katrore e kubike të këtyre numrave pa përdorimin e strukturave përsëritëse do të ishte:

```

printf ( "   numri 1   katrori   %d   kubi   %d",
        1*1, 1*1*1);
printf ( "   numri 2   katrori   %d   kubi   %d",
        2*2, 2*2*2);
:
:

printf ( "   numri 10   katrori   %d   kubi   %d",
        10*10, 10*10*10);

```

Me përdorimin e vorbullave, shumica e këtij kodi zhduket, gjë që e bën kodin të jetë më kompakt dhe më lehtë për t'u shkruar dhe për t'u lexuar. P.sh:

```
#include <stdio.h>

void main (void)
{
    int numri;

    printf("NUMRI      KATRORI      KUBI\n");
    printf("-----      -")
    printf("-----\n");

    numri = 1;

    while (numri < 11)
    {
        printf ("%5d %7d %4d\n", numri ,
                numri*numri, numri*numri*numri);
        numri++ ;
    }
}
```

Testimi i kushtit $\text{numri} < 11$ bëhet për arsye se numri i fundit për t'u nxjerrë është numri 10. Nëse do të testonim $\text{numri} < 10$ atëherë numri i fundit i dalë do të ishte 9, sepse kur variabla numri e merr vlerën 10 (vlerë e cila nuk do të ishte më e vogël se konstantja 10) kodi brenda vobullës while nuk do të ekzekutohej më. Po të harronim të shkruanim udhëzimin numri-- brenda strukturës while, atëherë while do të përsëritej pa fund, për arsye se vlera e variablës numri nuk do të ndryshohej fare, duke mbetur gjithmonë 1 dhe kushti $\text{numri} < 11$ (pra $1 < 11$) do të plotësohej gjithmonë.

Vorbulla do-while është e ngjashme me vorbullën while, përveçse në vorbullën do-while së pari ekzekutohet udhëzimi dhe pastaj ekzekutohet kushti. Në vorbullën do-while, udhëzimi ekzekutohet së paku njëherë, kurse në vorbullën while mund të mos ekzekutohet fare nëse kushti nuk plotësohet që në fillim.

Sintaksa e vorbullës do-while është:

```
do
    <udhëzimi>
while( <kushti> );
```

2.5 Komanda Break

Ndonjëherë nevojitet që ta kontrollojmë përsëritjen e vorbullave të përmendura më sipër, jo vetëm me testimin e kushtit në fillim të vorbullës `for` dhe `while`, ose në mbarim të vorbullës `do-while`. Komanda `break` bën daljen e menjëhershme nga këto vorbulla pa testuar kushtin për përsëritje.

Programi i mëposhtëm i "fshin" simbolet për 'hapësirë' (tastiera e gjatë) dhe simbolet për "kryerresht" (tab) në fund të çdo fjalie. Me përdorimin e komandës `break`, ekzekutimi i programit "del" nga trupi i vorbullës `while` nëse simboli në fund të fjalisë (vargut të simboleve, *string*) nuk është hapësirë (' ') apo kryerresht ('\t').

```
#define    MAXFJALIA    500

/* ky program largon hapësirat dhe shkronjat
   tab në fund të fjalive */

void main ( )
{
    int n;
    char fjalia[MAXFJALIA];

    while ( (n = lexofjaline (fjalia, MAXFJALIA)) > 0 )
    {
        while ( --n >= 0 )    /* vorbulla e brendshme */
        {
            if ( fjalia[n] != ' ' &&
                 fjalia[n] != '\t' &&
                 fjalia[n] != '\n' )
                break ;
        }

        fjalia[n+1] = '\0' ;
        printf("%s\n", fjalia);
    }
}
```

Funksioni `lexofjaline` kthen gjatësinë e fjalisë së lexuar dhe fjalinë e lexuar e ruan në variablën e pasuar si parametër. Vorbulla e brendshme `while` fillon në shkronjën e fundit të fjalisë së lexuar, dhe kontrollon çdo simbol (prej të djathtës në të majtë) përderisa nuk gjendet simboli i parë që nuk është as hapësirë as kryerresht. Përsëritja ndërpritet kur ky simbol të jetë gjetur.

Mënyrë tjetër për ta shkruar kodin, pa përdorimin e komandës break, do të ishte:

```
while ( (n = lexofjaline (.fjalia, MAXFJALIA) ) > 0)
{
    while ( --n >= 0 && (fjalia[n] == ' ' || /*vorb. e brend.*/
                        fjalia[n] == '\t' ||
                        fjalia[n] == '\n')
    )

        fjalia[n+1] = '\0' ;
    printf("%s\n", fjalia);
}
```

Në vorbullën e brendshme while, përsëritja bëhet përderisa kemi më shumë se një simbol në fjali dhe anëtarin *n* i fjalisë është njëri prej simboleve ' ' apo '\t' apo '\n'. Komanda break duhet të përdoret me kujdes, sepse bën që programi i shkruar të jetë më pak i lexueshëm dhe të përcillet më vështirë ecuria e ekzekutimit gjatë leximit të programit.

2.6 Komanda Continue

Komanda `continue` është e ngjashme me komandën `break`. Me përdorimin e komandës `continue` nuk dilet nga struktura përsëritëse, mirëpo suspendohet ekzekutimi i pjesës së kodit që pason komandën `continue` për atë radhë të përsëritjes së vorbullës. P.sh.:

```
for ( i = 0 ; i < n; i++)
{
    /* tejkaloji numrat më të vegjël se 10 */
    if (numrat[i] < 10)
        continue ;
    :
    /* kodi që manipulon me numra më të mëdhenj se 10 */
}
```

Komanda `continue` përdoret zakonisht kur pjesa e vorbullës që vjen pas kësaj komande është shumë e komplikuar, dhe përdorimi i kësaj komande mundëson që programi të jetë më kompakt.

2.7 Komanda goto dhe etiketat

Zakonisht, kjo komandë nuk përdoret, për arsye se programi me këtë komandë është shumë i vështirë të lexohet dhe të testohet.

Kjo komandë do të përdorej në situata si:

```
for ( . . . )
    for ( . . . )
    {
        :
        :
        if ( gabim )
            goto trajto_gabimin ;
    }
    :
    :

trajto_gabimin:
    analizo vlerën e gabueshme ;
```


Në situata të këtilla, komanda `break` nuk mund të përdoret, për arsye se dilet vetëm nga vorbulla në të cilën komanda `break` ekzekutohet. Me komandën `goto` tejkalohej ekzekutimi i programit deri te etiketa e përdorur pas komandës `goto`. Zakonisht kjo komandë përdoret për përballimin e gabimeve të mundshme në program.

Kodi i shkruar me komandën `goto` gjithmonë mund të shkruhet pa të. P.sh. kodi që vijon:

```
inico vlerat;
fillimi:
    if kushti nuk plotësohet goto fund
    vorbulla që përsëritet
    udhëzimi ndërrues
    goto fillimi
fund:
```

mund të shkruhet pa `goto`:

```
for (inico vlerat; kushti ; udhëzimi ndërrues)
    pjesa e programit që përsëritet;
```

Në gjuhën C si dhe në gjuhët tjera programuese ka disa mënyra për shprehjen e një problemi, mirëpo duhet zgjedhur metodën e cila është më e lexueshme dhe i përshtatet më shumë problemit të paraqitur.

Komanda `goto` zakonisht përdoret nga programet për gjenerimin e kodit në mënyrë "automatike", për shembull programet që përdoren për gjenerimin e kompajlerëve (të quajtur `yacc` dhe `lex`).

2.8 Tipat boolean

Gjuha C nuk ofron tip *boolean* (lexo: buliën). Variablat boolean janë ato variabla që kanë si rezultat vetëm vlerën e saktë dhe të pasaktë. Në gjuhën C variablat boolean mund të zëvendësohen me variablat e tipit `int`, ku vlera e saktë është çdo numër i ndryshëm nga zero, kurse vlera e pasaktë është numri zero.

P.sh. kodi:

```
for (numri = 9 ; numri ; --numri)
    printf("%d\n", numri);
```

është i njëjtë me:

```
for (numri = 9 ; numri > 0 ; --numri)
    printf("%d\n", numri);
```

për arsye se në rastin e parë, nëse numri është i barabartë me zero, vlerë e llogaritet se kushti nuk plotësohet (është i pasaktë) ndaj përfundon përsëritja e vorbullës for.

Kini parasysh se kështu mund të shkaktohet përsëritja e pakufishme e vorbullës, nëse udhëzimi i kushtit nuk e merr asnjëherë vlerën zero.

Një problem tjetër potencial është përdorimi i operatorit `*` në vend të operatorit `==`, p.sh. në rast të programit që duhet t'i injorojë simbolet hapësirë (' ') si:

```
for ( c= getchar( ) ; c == ' ' ; c= getchar( ) );
```

është shumë lehtë të gaboni në program dhe të shkruani:

```
for ( c = getchar( ) ; c = ' ' ; c = getchar( ) );
```

ku udhëzimi `c = ' '` nuk krahason variablën `c` me simbolin për hapësirë, por ia cakton vlerën `' '` variablës `c`, e cila vlerë në kodin ASCII nuk është zero (pra udhëzimi brenda në kusht plotësohet). Kjo shkakton që vorbulla for të përsëritet pa mbarim.

Gjuha C përdor zeron dhe jo-zeron, për të paraqitur vlerën e saktë dhe të pasaktë në bazë të instrukcioneve të makinës, të përdorura në Asambler. Përdorimi i zeros dhe jo-zeros në gjuhën C lejon kompajlerin që drejtpërdrejt t'i përdorë instrukcionet e nivelit të makinës.

Ushtrime

1. Shkruajeni një program që lexon të dhënat e shtypura nga shfrytëzuesi. Nëse shfrytëzuesi shtyp germën 'a' programi le të nxjerrë në monitor fjalinë "Shkronja e parë e gjuhës shqipe!", përndryshe le të nxjerrë fjalinë "Provo prapë!". Programi të përfundojë me shtypjen e germës x nga shfrytëzuesi.
2. Shkruajeni programin që lexon numrat e shtypur nga përdoruesi. Nëse përdoruesi shtyp numrin 28 nxirrni në monitor fjalinë "28 Nëntori - Dita e Flamurit!". Nëse përdoruesi shtyp numër më të vogël se 28, nxirrni në monitor fjalinë "Provoni numër më të madh!". Nëse përdoruesi shtyp numër më të madh se 28 atëherë nxirrni fjalinë "Provoni numër më të vogël!". Përdorëni komandën `if else if`.
3. Shkruajeni programin sikurse nën pikën 2, mirëpo asisoj që programi të mos përfundojë përderisa përdoruesi nuk shtyp numrin 28. Përdorëni komandën `while`.
4. Shkruajeni programin sikurse në pikën 3, duke përdorë komandën `for`.
5. Shkruajeni programin që shtyp në monitor të gjithë numrat tek, prej 1 deri në 99. Pastaj shtypini numrat në renditje prej numrit 99 deri në 1. Përdorëni komandën `if brenda` vorbullës përsëritëse për të testuar nëse numri nuk është i plotpjesëtueshëm me 2 (pra për t'i përzgjedhur vetëm numrat tek).
6. Shkruajeni programin që bën shumëzimin e numrit 2 me secilin numër prej 1 deri në 500. Mirëpo, nëse prodhimi është më i madh sesa 256, atëherë përfundojeni përsëritjen. Përdorëni komandën `break` për përfundimin e përsëritjes (daljen nga vorbulla).
7. Shkruajeni programin që shtyp numrat çift prej 2 deri në 64. Përdorëni komandën `continue`.
8. Shkruajeni programin që lexon numrin e shtypur nga shfrytëzuesi. Nëse numri i shtypur është i plotpjesëtueshëm me 2, atëherë nxirrni në monitor tre numra çift të njëpasnjëshëm më të mëdhenj sesa numri i shtypur nga shfrytëzuesi. Nëse numri i shtypur nga përdoruesi nuk është i plotpjesëtueshëm me 2, atëherë shtypini tre numra tek të njëpasnjëshëm, më të mëdhenj sesa numri i shtypur nga shfrytëzuesi. Përdorëni komandën `do - while` për përsëritjen e jashtme, kurse komandën `for` për përsëritjen e brendshme.
9. Shkruajeni programin për leximin e fjalive (vargjeve të simboleve). Nëse shfrytëzuesi shtyp fjalën "fund", përfundojeni programin, përndryshe shtypeni fjalinë me renditje të anasjelltë të simboleve. P.sh. nëse përdoruesi shtyp fjalinë "Nëna nuk e pa Anën", ju do ta shtypni fjalinë kështu: "nëna ap e kun anën".

Përmbledhje

Gjuha C ofron dy komanda seleksionuese `if-else` dhe `switch` për të vendosur për kahjet (opcionet) e ekzekutimit të programit në bazë të disa kushteve të paraqitura gjatë ekzekutimit të programit.

Vorbullat janë struktura që mundësojnë ekzekutimin e një pjese të kodit shumë herë, përderisa plotësohet kushti për ekzekutim.

Gjuha C ofron këto tri vorbulla:

- ♦ `for`
- ♦ `while`
- ♦ `do-while`

Vorbullat bëjnë që kodi të jetë më kompakt dhe më i lexueshëm.

Komandat seleksionuese dhe vorbullat përsëritëse janë shumë të përdorshme dhe mundësojnë zgjidhjen e shumë problemeve në programim.

Nëse kemi parasysh se paraqitja e një simboli '*' me funksionin putchar (të furnizuar nga libraritë e gjuhës C) është nënproblemi më primitiv, atëherë dizajnimi *top-down* sugjeron që ky problem të zgjidhet me përsëritjen e nënproblemit për paraqitjen e një rreshti. Kurse nënproblemi për paraqitjen e një rreshti mund të zgjidhet me përsëritjen e funksionit (primitiv) putchar.

Një prej zgjidhjeve të këtij problemi e kemi paraqitur në figurën 3.2. Tipi void para funksionit tregon se funksioni nuk kthen asnjë vlerë, ndaj është "procedurë". Gjuha C nuk bën ndonjë dallim ndërmjet *procedurave* dhe *funksioneve*, sikundër bëjnë disa gjuhë të tjera. Gjuha C, ofron vetëm funksione, të cilat kthejnë ose nuk kthejnë vlerë. Nëse tipi specifikues nuk shkruhet para funksionit, atëherë gjuha C supozon se funksioni kthen vlerë të tipit int. Funksionet në gjuhën C janë të gjitha të të njëjtit nivel, që d.m.th. se nuk mund ta deklarojnë ndonjë funksion brenda funksionit tjetër. Nga kjo rrjedh se të gjitha funksionet janë të dukshme për njëra-tjetrën. Mirëpo, kjo ka efekt të kundërt në fshehjen e informatave. Në gjuhën C, fshehja e informatave arrihet me kompajlimin e fajllave të veçuarë.

```
#include <stdio.h>

void rreshti(void)    // shtyp vargun me katër yje
{
    putchar('*');
    putchar('*');
    putchar('*');
    putchar('*');
    putchar('\n');
}

void drejtkëndëshi (void) // vizaton drejtkëndëshin
{
    rreshti( );
    rreshti( );
    rreshti( );
}

void main ( void )    // programi kryesor
{
    drejtkëndëshi ( );
}
```

Figura 3.2

Në gjuhën C, një funksion me një emër të caktuar dhe një numër të dhënë parametrash mund ta definojmë vetëm një herë të vetme. Kurse në gjuhën

C++ mund të definojmë disa funksione me emër të njëjtë, përdorisa tipi dhe numri i parametrave të funksioneve është i ndryshëm.

3.2 Variablat globale

Disa programe nuk mund të shkruhen pa përdorimin e variablave globale, mirëpo duhet të keni kujdes gjatë përdorimit të tyre. Preferohet që t'i shmangeni sa më shumë përdorimit të variablave globale, sepse këto variabla shkaktojnë paqartësi në leximin e programit, si dhe gjatë ekzekutimit programi mund të japë rezultat të papritur. P.sh. programi i mëposhtëm jep rezultat të papritur:

```
#include <stdio.h>

#define RR      '\n'
#define GJATESIA 5

// variabla globale e cila nuk duhet të deklarohet këtu:
int i ;

// funksion i cili shtyp vargun prej pesë yjesh :
void vargu ( void )
{
    for ( i = 0; i < GJATESIA; ++i )
        putchar('*');

    putchar(RR);
}

void drejtkendeshi(void)
{
    for (i=0; i < GJATESIA ; ++i )
        vargu( );
}

void main (void)
{
    drejtkendeshi( );
}
```

Rezultati i këtij programi është një varg i vetëm prej 5 yjesh. Pra rezultati nuk është siç kemi menduar se do të jetë - drejtkëndësh. Arsyeja është përdorimi

pa kujdes i variablës globale. Vëreni se vlera e variablës i pas thirrjes së parë të funksionit vargu është 5, kështu që në funksionin drejtkendeshi (në vorbullën for), funksioni vargu thirret vetëm një herë. Në strukturën for të funksionit drejtkendeshi, variabla i inicohet me vlerën 0, dhe pas thirrjes së funksionit vargu, vlera e kësaj variable ndryshohet në 5. Kështu që kushti për përsëritjen e funksionit vargu nuk plotësohet. Problemi i mësipërm do të zgjidhej me përdorimin e variablës lokale në funksion:

```
#include <stdio.h>

#define RR      '\n'
#define GJATESIA 5

// funksion i cili shtyp vargun prej pesë yjesh
void vargu ( void )
{
    int i;

    for ( i = 0; i < GJATESIA; ++i )
        putchar('*');

    putchar(RR);
}

void drejtkendeshi(void)
{
    int i;

    for (i=0; i < GJATESIA ; ++i )
        vargu( );
}

void main (void)
{
    drejtkendeshi( );
}
```

Variablat globale zakonisht përdoren për ta paraqitur kodin e gabimit (në implementimin e librarive standarde). Mirëpo në të shumtën e rasteve është shumë lehtë të gjejmë zgjidhje të tjera, pa i përdorë variablat globale.

3.3 Komunikimi ndërmjet funksioneve

3.3.1 Parametrat

Funksionet, të cilat marrin pjesë në zgjidhjen e problemit, ndonjëherë kanë nevojë të komunikojnë ndërmjet vete për zgjidhjen e problemit. Funksioni duhet të jetë në gjendje të pranojë informata nga funksioni thirrës dhe t'i kthejë informata funksionit thirrës.

Mënyra në të cilën funksioni mund të komunikojë me ndonjë funksion tjetër, definon nënshkrimin e funksionit. Nënshkrimi i funksionit përmban emrin e funksionit dhe definimin e parametrave. Këta parametra quhen *parametrat formalë*, kurse gjatë thirrjes së funksionit, vlerat që i pasohen këtij funksioni quhen *parametrat aktualë*.

Është e domosdoshme që numri, renditja dhe tipi i parametrave aktualë të jenë të njëjtë si parametrat formalë. P.sh. të supozojmë se funksioni duhet ta shkruajë një rresht prej katër simbolesh, atëherë një zgjidhje e mundshme e këtij problemi do të ishte:

```
#include <stdio.h>

void rreshti ( char g )
{
    putchar (g);    putchar(g);
    putchar (g);    putchar(g);
}

void main (void)
{
    rreshti('*');
}
```

Nëse në vend të yllit '*' na duhet simboli '+', atëherë funksionin do ta thirrjmë në këtë mënyrë: `rreshti('++')`. Në kodin e mësipërm, variabla `g` është parametër formal dhe kur thirret funksioni `rreshti`, atëherë parametri formal `g` merr vlerën e parametrat aktual nga udhëzimi që e thirr këtë funksion.

Numri i parametrave formalë dhe i parametrave aktualë në shprehjen thirrëse të funksionit, duhet të jetë i njëjtë. Meqë gjuha C është kontrolluese e dobët e tipeve, kompajleri i gjuhës C nuk analizon nëse përputhen tipet e parametrave formalë dhe aktualë. Ju është lënë përsipër juve si programerë, që ta kontrolloni përputhjen e tipeve të parametrave formalë me ata aktualë. Nëse

nuk përputhen tipet e parametrave, do të vëreni se kenë gabime gjatë ekzekutimit të programit.

Në programin që vijon kemi paraqitur një shembull të thjeshtë, i cili në vend që të shtypë vlerën 2.3, shtyp vlerën 2.

```
#include <stdio.h>

void shtyp(int no)
{
    printf(" numri %d", no);
}

void main (void)
{
    float numri = 2.3 ;
    shtyp(numri);
}
```

Funksioni mund të ketë numër të madh të parametrave, mirëpo rekomandohet që më së shumti t'i ketë 3 deri në 4 parametra. Po ashtu rekomandohet që funksioni ta kryejë një funksion të vetëm.

Gjuha C++ (për dallim prej gjuhës C) ofron dy lloje të pasimit të parametrave: pasimi i parametrave me vlerë dhe pasimi i parametrave me referencë. Pasimi i parametrave me vlerë bën pasimin e kopjes së vlerës së variablës, kurse pasimi me referencë bën pasimin e adresës së variablës. Gjatë pasimit të parametrave me vlerë, funksioni krijon një variabël lokale me vlerë të njëjtë me atë të parametrat aktual. Gjatë përdorimit të parametrat në funksion, manipulimet bëhen me variablën lokale të krijuar.

P.sh. nëse e shkruajmë një funksion për t'i shkëmbyer vlerat e dy variablave:

```
void shkembe3 ( int a , int b )
{
    int temp ;

    temp = a;
    a    = b;
    b    = temp;
}
```

Le ta provojmë këtë funksion duke ia çuar dy variabla: variablën x me vlerë 5, si dhe variablën y me vlerë 8. Me thirrjen e funksionit:

³ Disa kompajlerë nuk mundësojnë përdorimin e germës 'e' për emërtimin e variablave ose funksioneve. Prandaj funksioni është quajtur shkembe në vend se shkëmbe.

```
shkembe ( x , y );
```

vlerat e variablae x dhe y nuk do të shkëmbehen për arsye se manipulimi në funksion është bërë me variabla lokale. Këtë problem mund ta zgjidhim me përdorimin e pasimit të parametrave me referencë:

```
void shkembe ( int& a , int& b )
{
    int temp ;
    temp = a;
    a     = b ;
    b     = temp ;
}
```

Duhet ta keni parasysh se kjo metodë vlen vetëm për gjuhën C++. Nëse doni që parametrat të pasohen me referencë për gjuhën C, atëherë shihni kapitullin për treguesit (kapitulli 4). Me pasimin e adresave të variablae x dhe y, funksioni manipulon me këto variabla dhe pas thirrjes së funksionit, vlerat e variablae x dhe y do të shkëmbehen, pra do të kemi x = 8 dhe y = 5. Konstantat ose udhëzimet nuk mund të përdoren si parametra aktualë nëse pasojmë parametrat me referencë.

3.3.2 Kthimi i informatave nga funksionet

Përveç pranimit të informatave nga funksioni thirrës, funksionet mund t'u kthejnë vlera funksioneve thirrëse. Shembujt më të mirë, të njohur për kthimin e vlerave, janë funksionet që kryejnë kalkulime matematikore. Zakonisht këto funksione ekzekutojnë disa kalkulime dhe pastaj ia kthejnë rezultatin funksionit thirrës. Si shembull, shihni kodin në vijim, ku funksioni kthen katrorin e numrit që pranon:

```
#include <stdio.h>

int katrori(int n)
{
    int k;

    k = n*n;

    return k;
}
```

```

void main (void)
{
    int numri , rezultati;

    printf("Shtyp një numër: ");
    scanf("%d",&numri);
    rezultati = katrori(numri);
    printf("Katrori i numrit %d është %d \n",
        numri, rezultati);
}

```

Shembulli 3.3.1

Shihni shembullin e mëposhtëm, në të cilin funksioni `nderro_germat` i ndërron germat nga të mëdha në të vogla:

```

#include <stdio.h>

char nderro_germat(char germa)
{
    return germa + ( 'a' - 'A' );
}

void main ( void )
{
    int g;

    printf("Shtype ndonjë gërmë të madhe:");

    g = getchar( );

    printf("Germa e vogël %c e shndërruar në gërmë të madhe %c\n",
        g, nderro_germat(g));
}

```

Shembulli 3.3.2

Në programin në shembullin 3.3.1, vlera e kthyer nga funksioni ruhet në variabël, kurse në programin në shembullin 3.3.2, vlera e kthyer nga funksioni është pjesë e udhëzimit.

Keni parasysh se funksioni mund të pranojë disa vlera të variablave të ndryshme , mirëpo mund ta kthejë vetëm një vlerë. Preferohet që secili funksion ta ketë vetëm një përdorim të komandës `return`, sepse kështu lehtësohet leximi i programit. Kështu edhe sigurohet që funksioni të ketë një pikë së hyrjes së informatave dhe një pikë të daljes së informatave. Mirëpo, në raste të caktuara në funksion duhet të përdoren më shumë se një komandë e vetme `return`, psh:

```
tipi funksioni ( parametrat )  
{  
    if ( parametrat e gabueshëm )  
        return tregues_i_gabimit  
  
    Bëj disa kalkulime  
    return rezultati_i_saktë  
}
```

Në këtë rast, nëse parametrat janë të gabueshëm, funksioni kthen treguesin e gabimit. Përndryshe bëhen kalkulimet dhe kthehet rezultati i tyre. Ju duhet ta testoni në funksionin thirrës, se cila vlerë është kthyer (pra: treguesi i gabimit apo rezultati i saktë).

3.4 Prototipi i funksionit

Secili funksion i përdorur në program, duhet të jetë i definuar para përdorimit, ose përndryshe, duhet definuar të paktën prototipi i funksionit para përdorimit të tij. Prototipi i funksionit përbëhet prej emrit të funksionit, tipit kthyes si dhe tipit dhe numrit të parametrave të funksionit. P.sh. në kodin e mëposhtëm kemi definuar prototipin e funksionit `nderro_germat`, para përdorimit të këtij funksioni në funksionin `main`:

```
#include <stdio.h>

/* prototipi i funksionit */
char nderro_germat(char germa);

/* ose deklarimi i funksionit */
void main ( void )
{
    int g;

    printf("Shtype ndonjë germë të madhe:");
    g = getchar( );
    printf("Germa e vogël %c e shndërruar në germë të madhe %c\n",
           g, nderro_germat(g));
}

/* definimi i funksionit */
char nderro_germat(char germa)
{
    return germa + ( 'a' - 'A' );
}
```

Prototipi përdoret nëse definimi i funksionit paraqitet më poshtë në kod sesa thirrja e këtij funksioni, ose definimi i funksionit është pjesë e librarisë apo ndonjë fajlli tjetër. Prototipi i funksionit përdoret edhe nga kompajleri, për të garantuar përputhjen e thirrjeve të funksioneve me definimet e tyre.

Nëse në kodin e mësipërm nuk bëhet deklarimi i prototipit, atëherë kompajleri do ta lajmërojë gabimin

Prototipi i funksionit nderro_germat nuk është deklaruar.

Nuk është e thënë që prototipi i funksionit të paraqitet në fillim të programit. Kjo vjen më shumë nga konvencioni i përdorur nga programerët e gjuhës C.

Alternativë tjetër do të ishte përdorimi i prototipit vetëm në funksionet që e thirrin atë funksion:

```
#include <stdio.h>

void main ( void )
{
    /* prototipit i funksionit */
    char nderro_germat(char germa);

    int g;

    printf("Shtype ndonjë garmë të madhe:");
    g = getchar( );
    printf("Germa e vogël %c e shndërruar në garmë të madhe %c\n",
        g, nderro_germat(g));
}

/* definimi i funksionit */
char nderro_germat(char germa)
{
    return germa + ( 'a' - 'A' );
}
```

Ky variant ka përparësi gjatë përmirësimit të programit, ose bartjes së funksionit në fajll tjetër. Mirëpo nuk e preferojnë programerët (zakonisht përtacë) sepse kërkon më shumë kohë për ta shkruar prototipin në secilin funksion që përdor funksionin e përmendur në prototip.

3.5 Hapësira e veprimtarisë dhe jeta e variablave

Në programet e përdorura në shembujt e mëparshëm, variablat janë deklaruar brenda funksioneve. Këto variabla janë variabla lokale të funksionit dhe ekzistojnë vetëm brenda funksionit të dhënë. Me fjalë të tjera, këto variabla nuk mund të përdoren në funksionet e tjera.

Variablat globale janë ato variabla që deklarohen në fillim të fajllit (kudit) dhe jashtë funksioneve të këtij fajlli. Këto variabla nuk preferohet të përdoren, për arsye se paraqesin vështirësi për gjetjen e gabimeve logjike në program.

Të gjitha variablat, përveç hapësirës së tyre të veprimtarisë, kanë edhe kohëzgjatjen e jetës. Në rastin e variablave lokale, ato krijohen automatikisht në hyrje të funksionit dhe humben në dalje të funksionit. Për këtë arsye këto variabla quhen edhe variabla automatike. Pra, kohëzgjatja e jetës së variablave automatike është sa edhe kohëzgjatja e ekzekutimit të funksionit në të cilin këto variabla janë deklaruar. Në anën tjetër, kohëzgjatja e jetës së variablave globale është baras me kohëzgjatjen e ekzekutimit të programit. Këni parasysh se vlera fillestare e variablave automatike është e padefinuar, ndërsa vlera fillestare e variablave globale është zero. Vlerat fillestare të variablave globale ose variablave automatike mund të jepen me përdorimin e shenjës së barazimit:

```
int numri = 15;
```

Është e mundur që kohëzgjatja e jetës së variablave automatike të jetë e barabartë me kohëzgjatjen e ekzekutimit të programit, nëse përdorni komandën `static` para variablës së definuar. Sikurse në variablat globale, ku vlera fillestare është e barabartë me zero, edhe në variablat statike vlera fillestare është e barabartë me zero (nëse ju nuk i inicioni ndonjë vlerë me shenjën '=').

```
#include <stdio.h>

void vizit_e_funksionit (void)
{
    static int viz = 1;

    printf("Ky funksion është vizituar %d herë\n", viz);
    viz++;
}

void main (void)
{
    int i;
```


Ushtrime

1. Shkruajeni një funksion, i cili i shumëzon numrat kompleksë. Parametrat e këtij funksioni duhet ta paraqesin pjesën reale dhe pjesën imagjinare të dy numrave kompleksë, p.sh. `preale1`, `pimagj1`, `preale2`, `pimagj2`. Formula për shumëzimin e numrave kompleksë është:

$$(a + bi) \times (c + di) = (ac - bd) + (bc + ad)i$$

2. Shkruani një funksion të ngjashëm me atë më sipër, mirëpo që ky funksion ta bëjë mbledhjen e dy numrave kompleksë. Formula për mbledhjen e numrave kompleksë është:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

3. Shkruani funksionin (maksimumi) i cili pranon dy parametra të tipit `int` dhe kthen vlerën më të madhe të këtyre dy parametrave.

4. Implementoni funksionin e ushtrimit 3 pas funksionit `main` (në të cilin është përdorë funksioni `maksimumi`) dhe vështroni problemet e paraqitura nga kompajleri. Pastaj definoni prototipin e funksionit `maksimumi` para implementimit të funksionit `main`.

5. Shkruani funksionin për të gjetur rrënjën katrore të një numri të tipit `int`. Pra ky funksion duhet të marrë si parametër tipin `int` dhe ta kthejë vlerën e tipit `float`.

6. Proveni të shkruani funksionin i cili kthen vlerën faktoriale të një numri `int`. Formula për vlerën faktoriale të një numri `n` është:

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

P.sh. për numrin 5, vlera faktoriale do të ishte:

$$5 * 4 * 3 * 2 * 1 = 120$$

Përmbledhje

Funksionet mundësojnë ndarjen e problemeve në nënprobleme, duke e lehtësuar kështu zgjidhjen e problemeve të ndryshme në programim. Në gjuhën C çdo funksion duhet definuar para përdorimit të tij, ose përndryshe duhet definuar të paktën prototipin e funksionit. Prototipet janë nënshkrimet e funksioneve të cilat përshkruajnë tipin e parametrave dhe të vlerës kthyesë.

Gjuha C lejon definimin vetëm të një funksioni me emër të dhënë gjatë tërë programit. Të gjitha funksionet e definuara në gjuhën C janë të të njëjtit nivel, pra nuk është e mundur ta implementojmë një funksion brenda funksionit tjetër.

Në këtë kapitull përmendëm edhe hapësirën dhe jetën e variablave globale dhe variablave të definuara brenda funksionit. Kohëzgjatja e jetës së variablave globale është baras me kohëzgjatjen e ekzekutimit të programit, kurse kohëzgjatja e jetës së variablave të definuara brenda në funksion (përveç variablave statike brenda funksionit) është baras me kohëzgjatjen e ekzekutimit të atij funksioni.

Treguesit

Deri tani, numri i mundshëm i të dhënave në përdorim ka qenë i kufizuar në numrin e variablave të deklaruara. Kjo gjë nuk është shumë e levërdishme për programet e mëdha në të cilat numri i të dhënave nuk dihet paraprakisht. Ky problem tejkalohet me përdorimin e treguesve.

Shumica e programeve ruajnë sasi të mëdha të të dhënave gjatë ekzekutimit të tyre. Shpesh sasia e të dhënave nuk dihet, p.sh. editor i tekstit nuk e di sa i gjatë do të jetë teksti derisa ta përfundoni programin apo ta ruani fajllin. Nëse programin për editor e shkruani me përdorimin e variablave, atëherë sa variabla ju nevojiten për ruajtjen e tekstit të shkruar në editor, 100 variabla, 10000 variabla?

Në këtë kapitull ju njohim me tipin e ri - treguesin, i cili ndihmon për zgjidhjen e këtij problemi.

4.1 Treguesit

Treguesi është variabël e cila përmban adresën (adresa fizike në memorie) e një variable tjetër, dhe mund të përdoret për ta shfrytëzuar variablën në mënyrë indirekte.

Treguesit përdoren për t'i fshehur detajet (e nivelit më të ulët) e mënyrës se si ruhen variablat në memorie, mirëpo na mundësojnë t'i përdorim vlerat (përmbytjet) e këtyre variablave. Përdorimi pa kujdes dhe i tepërt i treguesve shpesh ndikon në përfundimin e padëshirueshëm të programit gjatë ekzekutimit ose në humbjen e memories (përcaktimin e memories dhe moslirimin e kësaj memorie). Prandaj shumë gjuhë (siç është Java) nuk parapëlqejnë përdorimin e treguesve, sepse është vështirë të mirëmbahen dhe të përdoren krahasuar me variablat normale.

Sidoqoftë, përparësia e treguesve qëndron në shpejtimin e ekzekutimit të programit, si dhe në shkurtimin e kodit të programit.

Në gjuhën C, treguesit përdoren edhe për ta tejkaluar problemin e pasimit të parametrave formalë në funksion. (Problem ky që e kemi përmendur edhe në kapitullin për funksionet). Në kodin më poshtë kemi paraqitur funksionin për ndërrimin e vlerave të dy variablaeve:

```
#include <stdio.h>

/* funksion me gabim */
void shkembe (int a , int b)
{
    int t;          /* variabël e përkohshme */

    t = a ;  /* ruaje vlerën e variablës a në t */
    a = b ;  /* variabla a tani mban vlerën e variablës b */
    b = t ;  /* variabla b tani mban vlerën e variablës a */
}

void main (void)
{
    int x, y;

    /* cakto vlerat e variablaeve x dhe y */
    x = 5;
    y = 7;

    shkembe(x, y) ;

    printf("x = %d dhe y = %d\n", x, y);
}
```

Ky program nuk bën atë që do të donim, edhe pse kodi duket në rregull. Rezultati i këtij programi është $x = 5$ dhe $y = 7$ në vend se të ishte $x = 7$ dhe $y = 5$. Arsyeja është se ndërrimi i vlerave të variablaeve bëhet vetëm brenda funksionit shkembe. Në funksionin shkembe kemi deklaruar parametrat a dhe b, të cilëve u caktohet adresë tjetër në memorie nga adresa e variablaeve x dhe y. Variabla a e kopjon vlerën e variablës x, si dhe variabla b e kopjon vlerën e variablës y, kështu që edhe pas shkëmbimit të vlerave të variablaeve a dhe b, vlerat origjinale të variablaeve x dhe y mbeten të pandryshuara.

Për ta zgjidhur këtë problem, adresat (në memorie) të variablaeve x dhe y i përdorim brenda në funksionin shkembe, në mënyrë që të ndërrohet përmbajtja e memories së caktuar për variablat që pasohen në funksion (psh x dhe y). Njëra prej mundësive do të ishte deklarimi i variablaeve x dhe y si variabla globale (kjo nuk është e preferueshme). Mundësia tjetër është përdorimi i treguesve.

Në gjuhën C++ pasimi i parametrave aktualë mund të bëhet pa përdorimin e treguesve. Zgjidhja e problemit në gjuhën C do të ishte (normalisht që kjo zgjidhje vlen edhe për gjuhën C++):

```
void shkembe (int *a , int *b)
{
    int t;

    t = *a ;
    *a = *b ;
    *b = t ;
}
```

Arsyeja pse me përdorimin e treguesve zgjidhet problemi i mësipërm, është se treguesi a dhe treguesi b përmbajnë adresat e memories së variablave që pasohen në funksionin shkembe. Dhe çdo manipulim me këta tregues bën që të manipuloni me vlerën e ruajtur në adresën e variablave origjinale (variablave të pasuara në funksionin shkembe - në shembullin tonë variablat x dhe y).



Treguesit a dhe b tregojnë në memorien e variablave x dhe y.

Kjo metodë mund të përdoret edhe në C++, mirëpo gjuha C++ e ka metodën e vet (pasimin me anë të referencës së variablave), metodë e cila është më e lexueshme sesa kjo e përdorimit të treguesve.

Treguesit në gjuhët C dhe C++ definohen sikurse variablat e tjera:

```
/* defino tregues të tipit integjer */
int *pi ;
```

Në gjuhën C++ mund ta përdorim edhe këtë metodë (ku shenja * ndodhet pas tipit të variablës, jo para emrit):

```
int* pi ;
```

Treguesi i deklaruar nuk tregon në asnjë adresë. Për ta përdorë treguesin, ne duhet t'i adresojmë ndonjë objekti (të tipit të njëjtë me atë të treguesit). Kjo bëhet me vënien e shenjës & para objektit të deklaruar, përveç nëse objekti është array apo tregues. Vlera e variablës së treguar nga treguesit lexohet ose ndërrohet me anë të operatorit referencues *. Në shembullin që vijon kemi

deklaruar variablën `x` dhe treguesin `pi` të tipit `int`. Në këtë shembull treguesi `pi` tregon në adresën e variablës `x` dhe pastaj e ndërron vlerën e kësaj variable.

```
int x = 5 ;
int *pi;

// shenja & para variablës paraqet adresën
// në memorie të variablës x

pi = &x ;

*pi = 11 ;

printf( "%d", x);
```

Rezultati i këtij programi do të jetë shtypja e vlerës 11. Siç keni vënë re, përdorimi i përmbajtjes së ndonjë objekti të treguar me tregues bëhet përmes operatorit `diferencues *`. Në shembullin e mësipërm `*pi` tregon vlerën e variablës `x`, përndryshe vetëm `pi` (pa shenjën `*`) tregon adresën në memorie të variablës `x`.

4.1.1 Operacionet për tregues

Autorritja dhe autozbritja janë operacionet që përdoren më shpesh me tregues, sidomos kur treguesit tregojnë në objekt *array* të tipit `char`⁴. Kur treguesi autorritet, atëherë rritet për aq sa objekti shfrytëzon memorie për një vlerë të atij tipi. Kjo vlen edhe për autozbritjen, pra treguesi zbritet për aq sa objekti shfrytëzon memorie për një vlerë. P.sh. nëse kemi deklaruar një variabël të tipit `int` (`int x`) dhe supozojmë se objekti `x` gjendet në adresën 500 në memorie, si dhe ky objekt (`int x`) shfrytëzon 4 bajt për çfarëdo vlere, atëherë:

```
int x = 3;
int *p;
p = &x;          // p do ta ketë vlerën 500
                  // kurse *p do ta ketë vlerën 3
p++;
```

Pas autorritjes (`p++`), `p` do ta tregojë adresën e memories 504 (e jo 501, siç do të ndodhte me autorritjen e vlerave të variablave normale). Pra në autorritje dhe autozbritje, treguesit rriten apo zbriten mvarësisht nga sasia e memories që kërkojnë tipet e treguesit.

⁴ Objekti *array* i tipit `char` njihet si *string*

Tipi `char` zakonisht kërkon një bajt për mbajtjen e vlerës së një variable, kështu që autorritja dhe autozbritja për treguesit e tipit `char` është e ngjashme me atë të variablave të zakonshme. Sasia e memories për tipin `int` mvaret nga sistemi operativ. P.sh. në MS-DOS (16 bit) është 2 bajt, kurse në UNIX 4 bajt, për këtë arsye autorritja dhe autozbritja për treguesit e tipit `int` mvaret nga sistemi operativ.

4.2 Treguesit dhe variablat array

Në gjuhën C treguesit dhe tipet *array* përdoren së bashku për arsye të relacionit të fortë ndërmjet këtyre tipeve. Treguesit zakonisht përdoren për manipulimin e variablave *array*. Variablat *array* janë sikurse treguesit, duke qenë se variablat *array* tregojnë elementin e parë në radhën e elementeve të njëjta. P.sh. nëse kemi deklaruar një variabël *array* të tipit *char* dhe një tregues të tipit *char*:

```
char fjalial[20] ;      /* variabla array e tipit char */
char *p ;              /* tregues për char */
```

Pasi *fjalial* është adresa e fillimit të variablës *array*, atëherë udhëzimet 1 dhe 2 kanë të njëjtin kuptim.

```
p = fjalial ;          /* 1 */
```

apo

```
p = &fjalial[0] ;     /* 2 */
```

Në këtë rast treguesi *p* tregon anëtarin e parë të variablës *fjalial* të tipit *array*. Në rastet kur treguesi tregon në *array*, nuk kemi nevojë ta përdorim shenjën & para emrit të variablës, sepse vetë emri është adresa e anëtarit të parë.

Udhëzimi 1 ekzekutohet më shpejtë sepse udhëzimi *fjalial[0]* përkthehet nga kompajleri në:

```
fjalial + 0
```

po ashtu:

```
fjalial[2] ;
```

përkthehet në *fjalial + 2*, etj. P.sh. funksioni për ta shtypur një *string* (varg), mund të shkruhet në disa mënyra, disa prej të cilave i kemi definuar në kodin e mëposhtëm:

```
/* definimi 1 */
void shtype_vargun(const char str[ ])
{
    unsigned int i ;

    for ( i = 0 ; str [i] != '\0' ; ++ i )
        putchar ( str [i] ) ;
}
```

```

/* definimi 2 */
void shtype_vargun(const char str[ ])
{
    for ( ; *str != '\0' ; ++str )
        putchar ( *str ) ;
}

```

Ekzekutimi i funksionit në definimin 2 është më i shpejtë sesa ai i definimit 1, sepse në funksionin e parë variabla `str[i]` e tipit `array` përkthehet në çdo përsëritje në `str + i`. Kalkulimi i adresës bëhet edhe në `str[i] != '\0'`, e edhe në `putchar(str[i])`, kurse në funksionin e definimit 2, kalkulimi bëhet vetëm në shprehjen `++str`.

Duhet të keni kujdes gjatë përdorimit të treguesve të tipeve të ndryshme (p.sh. treguesit të tipit `int` me variablën `array` të tipit `char`, apo anasjelltas). Vëreni me kujdes kodin e mëposhtëm:

```

char a[10] = { 3, 1, 12, 33, 9,
               66, 31, 65, 88, 0 };
char *tc;
int *ti;

```

atëherë, nëse kemi nevojë t'i përdorim treguesit, do të bënim si më poshtë:

```

tc = a ;           /* tc tregon variablën array a */
ti = ( int * ) tc; /* ti po ashtu tregon variablën array a */

```

Të dy treguesit `tc` dhe `ti` tregojnë në të njëjtën adresë, mirëpo nëse preferencohen, atëherë japin vlera të ndryshme. Arsyeja është se tipi `int` zë vend në memorie më shumë se tipi `char`, dhe gjatë referencimit të treguesit `ti` lexohen dy vlera të tipit `char` (apo më shumë, mvarësisht nga hapësira në bajt që zë tipi `int` në sistemin e dhënë operativ) të cilat bashkangjiten dhe japin vlerë tjetër. Me bashkangjitje këtu nuk kemi parasysh mbledhjen e dy vlerave, por bashkangjitjen e vlerave në numra binarë, p.sh.:

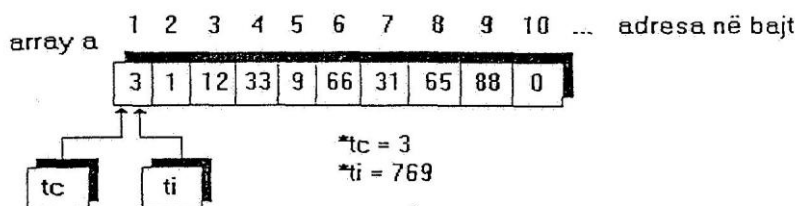
```

vlera 1: 00000011 (numri decimal 3)
dhe     vlera 2: 00000001 (numri decimal 1)

```

Nëse këto vlera janë të ruajtura në memorie njëra pas tjetrës (siç është rasti me anëtarët e variablës `array`), dhe nëse treguesi `ti` tregon në anëtarin i cili e ka vlerën 00000011, atëherë:

```
*ti do ta ketë vlerën 00000011000000012 = 76910
```



Në shumë makina (p.sh. në makinat të llojit 68000) madhësia e tregueseve të tipeve të ndryshme është e njëjtë, kështu që krahasimi ose barazimi i njërës tregues me tjetrin nuk do të shkaktonte ndonjë problem.

Në makina të tjera (si 80386) treguesit e tipeve të ndryshme janë të madhësive të ndryshme, kështu që duhet të kenë kujdes gjatë krahasimit ose barazimit të treguesve të tyre. Preferohet që treguesit t'i iniconi me të njëjtin tip variable si ajo e tipit *array*, kështu që kodi do të jetë kompatibil për lloje të ndryshme të makinave.

4.3 Variablat array të tipit tregues

Treguesi i tipit `char` mund t'i tregojë konstantat *string* të gjatësive të ndryshme. P.sh. një variabël *array* e tipit `char` tregues mund të ketë numër të ndryshëm të anëtarëve që tregojnë në *string*.

```
char *emrat[5] ;
```

Në kodin e mësipërm kemi deklaruar një variabël të tipit *array* që përmban 5 anëtarë të tipit `char` tregues. Si zakonisht pas deklarimit të variablës *array*, anëtarët kanë vlerë të padefinuar, kështu që duhet t'u jepet ndonjë vlerë:

```
emrat[0] = "Naim Frasheri" ;
emrat[1] = "Migjeni" ;
emrat[2] = "Fan Noli" ;
emrat[3] = "Ismail Qemajli" ;
```

Kjo mund të arrihet edhe me deklarimin e variablës *array* si dydimensionale:

```
char Aemrat[4][20] =
{
    "Naim Frasheri" ,
    "Migjeni" ,
    "Fan Noli" ,
    "Ismail Qemajli"
} ;
```

mirëpo në këtë rast, 20 bajt rezervohen për çdo anëtar të variablës *array*, pa marrë parasysh gjatësinë e anëtarëve në *string* (p.sh. "Migjeni" zë 8 bajt mirëpo për të rezervohen 20 bajt). Kurse në rastin e parë, anëtarët e variablës *string* zënë vend aq sa iu nevojitet, plus vend për treguesin e çdo anëtari. Në figurën 4.3.1 kemi paraqitur në mënyrë figurative se si ruhen konstantat në të dy rastet e sipërpërmendura të definimit të tipit *array*.

Variablën *emrat* mund ta iniciojmë edhe kështu:

```
char *emrat [ ] =
{
    "Naim Frasheri" ,
    "Migjeni" ,
    "Fan Noli" ,
    "Ismail Qemajli"
} ;
```

Kodi për shtypjen e këtyre emrave do të ishte:


```
for ( i = 0 ; emrat[i] != 0 ; ++i )
    printf ("%s == %s\n", emrat[i]) ;
```

Kjo metodë është më e preferueshme, sepse kështu, kodi për shtypjen e emrave është i pamvarur nga numri i anëtarëve të variablës *array*.

Në përgjithësi rekomandohet që kodi i programit të jetë i pamvarur nga të dhënat, kështu që me shtimin apo heqjen e të dhënave, kodi i programit nuk ka nevojë të ndërrohet në shumë vende (ose nuk ka nevojë të ndërrohet fare).

4.4 Treguesit në tregues

Me definimin e variablës *array* të tipit `char` tregues, p.sh.:

```
char *emrat[4] ;
```

udhëzimi `*emrat` është adresa e anëtarit të parë (treguesit `char`) kurse `emrat` është adresa e variablës *array*, anëtarët e të cilës janë tregues të tipit `char`. Më sipër kemi përmendur se mund të manipulohet me anëtarët e variablës *array* me përdorimin e treguesit të tipit të njëjtë si të anëtarëve të variablës *array*. Në këtë rast anëtarët në *array* janë vetë tregues, kështu që për manipulim na duhet treguesi në tregues të tipit `char`, p.sh.:

```
char **p ; /* definim plotësisht i vlefshëm */
```

Në këtë rast `p` është treguesi në tregues të tipit `char`, kurse `*p` është vetëm tregues `char`, si dhe `**p` është variabël e tipit `char`. Atëherë nëse e inicojmë treguesin `p` me variablën *array* `emrat`:

```
p = emrat ;
```

vlerave të ruajtura në *array* do të mund t'u qasemi me anë të treguesit `p` ose variablës `emrat` të tipit *array*. Në kodin e mëposhtëm kemi paraqitur mundësitë e ndryshme për t'u qasur vlerave të variablës `emrat` të tipit *array*:

<code>emrat</code>	<code>= p</code>	<code>= p</code>
<code>emrat[i]</code>	<code>= *(p+i)</code>	<code>= p[i]</code>
<code>*emrat[i]</code>	<code>= (*(p+i))</code>	<code>= *p[i]</code>
<code>&emrat</code>	<code>= p</code>	<code>= p</code>
<code>&emrat[i]</code>	<code>= p+i</code>	<code>= &p[i]</code>

Kodi për shtypjen e emrave do të mund të shkruhet si më poshtë:

```
void shtypi_emrat (const char *n[ ])
```

```

{
    unsigned int i ;

    for ( i = 0 ; n[i] != 0 ; i++)
        printf("%s\n",n[i]);
}

```

ose

```

void shtypi_emrat (const char **n )
{
    for ( ; *n != 0 ; ++n)
        printf("%s\n",*n);
}

```

4.5 Rezervimi i memories dinamike

Siç përmendëm më sipër, programet si editorit ose programet e tjera në të cilat sasia e të dhënave nuk dihet para ekzekutimit, është e vështirë të ruhen në program me numër të variabla të limituar. Këto të dhëna shpesh ruhen në memorie të qëndrueshme (si p.sh. fajll, apo detabejs), mirëpo këto të dhëna duhen lexuar nga programi i cili do të bëjë manipulimin me to. Pasi që sasia e të dhënave nuk dihet paraprakisht, atëherë duhet disa të deklarojmë (të rezervojmë në mënyrë dinamike) memorie për ruajtjen e këtyre të dhënave, si dhe ta mbajmë treguesin në memorie, në mënyrë që ta lirojmë memorien e rezervuar pas mbarimit të përdorimit. Natyrisht se ka mundësi të shumta për zgjidhjen e problemeve të këtilla pa përdorimin e treguesve, e sidomos në gjuhën C++ ose në gjuhët e tjera të nivelit të lartë. Sidoqoftë, në gjuhën C duhet krijuar struktura të reja për ruajtjen e të dhënave (të njohura si *list*, *b-tree* etj.) të cilat e rezervojnë memorien në mënyrë dinamike. Këto struktura përdorin treguesit për të treguar në memorien e rezervuar në mënyrë dinamike. Zakonisht një sasi e memories mban disa të dhëna si dhe treguesin e memories për të dhëna të tjera të lidhura mes vete. P.sh. në programin Editor, një sasi e memories përmban tekstin e rreshtit të parë të shkruar, treguesin në memorien që përmban rreshtin që vijon, si dhe treguesin në memorien që përmban tekstin në rreshtin tjetër, e kështu me radhë. Pasi që gjatësia e secilit rresht të tekstit të shkruar do të jetë e ndryshme, do ta kemi një tregues në tipin *char* si dhe tregues në strukturën e memories së përdorur për ruajtjen e të dhënave në çdo rresht të tekstit. Strukturën për ruajtjen e të dhënave do ta definojmë kështu:

```

typedef struct tagData
{

```

```

char *pRreshti;
struct tagData* pData;
} DATA;

```

Në program do të duhej ta rezervonim memorien për ruajtjen e fjalive, gjë që në gjuhën C mund të bëhet me përdorimin e funksionit:

```
void * malloc (size_t size);
```

Funksioni malloc është i definuar në librarinë standarde stdlib. Ky funksion kthen tregues në sasinë e memories së rezervuar. Nëse për shkaqe të ndryshme funksioni malloc nuk mund ta kthejë memorien e kërkuar (p.sh. kemi kërkuar më shumë memorie sesa përmban kompjuteri, apo memoria është rezervuar nga programet e tjera, etj.), atëherë ky funksion kthen vlerën NULL. T'ju përkujtojmë se tipi size_t i përdorur për parametrin në funksionin malloc është typesit i tipit int:

```
typedef unsigned int size_t;
```

Memoria e rezervuar me funksionin malloc duhet të lirohet me anë të funksionit free:

```
void free(void *p);
```

Funksioni për rezervimin e memories malloc zakonisht përdoret bashkë me funksionin sizeof, i cili tregon sasinë e memories së nevojitur të tipit të caktuar apo treguesit të atij tipi.

Të supozojmë se kemi shkruar këta rreshta në editor:

*O ju malet e Shqipërisë,
E ju lisat e gjatë që ju kam ndërmend ditë e natë*

Për ruajtjen e këtyre rreshtave do të shkruanim kodin si më poshtë:

```

DATA *pFillimi      = NULL;
/* rezervu memorien për rreshtin e parë */
DATA *pFjalia       = (DATA *) malloc(sizeof(DATA));
/* rezervu memorien për fjalinë */
pFjalia->pRreshti     = (char*)
malloc(strlen("O ju malet e Shqipërisë,")-1);
strcpy(pFjalia->pRreshti, "O ju malet e Shqipërisë,");

pFillimi = pFjalia;

```



```

/* rezervo memorien për rreshtin tjetër */
pFjalia = (DATA *) malloc(sizeof(DATA));

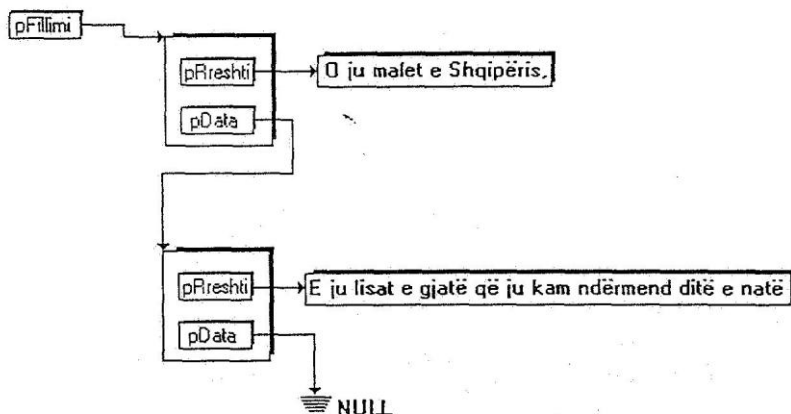
pFjalia->pRreshti = (char*) malloc(strlen("E ju lisat e gjatë
që ju kam ndërmend ditë e natë")+1);
strcpy(pFjalia->pRreshti, "E ju lisat e gjatë
që ju kam ndërmend ditë e natë");

pFillimi->pData = pFjalia;
pFjalia->pData = NULL; /* fundi i listës lidhëse */

```

Në kodin e mësipërm kemi përdorë konstantat (fjalitë konstanta) për të ilustruar si lidhen në *listë* lidhëse. Mirëpo në programe të vërteta nuk është e preferueshme t'i përdorni konstantat në kod (pasi që fjalitë pritet të furnizohen nga përdoruesi i programit).

Në figurën e mëposhtme kemi ilustruar si keni krijuar listën lidhëse të fjalive.



Siç mund ta vëreni nga kodi i mësipërm, kemi rezervuar memorie për strukturën për ruajtjen e rreshtit si dhe për vetë fjalinë në rresht. Këto fjali duhet të lidhen ndërmjet vete, në mënyrë që në rastet kur duam ta shtypim gjithë tekstin në editor, të fillojmë nga fjalia e parë duke vazhduar t'u qasemi fjalive që vijnë. Lidhshmëria e të dhënave njihet si listë lidhëse dhe përdoret shumë për ruajtjen e të dhënave, në rastet kur numri i elementeve nuk dihet paraprakisht. Në gjuhën C++ tipi *list lidhëse* është i definuar në libraritë standarde, të njohura si libraritë STL.

Siç mund ta vëreni, treguesi `pData` i objektit të fundit është inicuar me `NULL` për ta definuar fundin e listës, përndryshe gjatë leximit të listës nuk do të mund ta gjenim fundin e listës, dhe me siguri do të tentonim ta lexonim memorien e rezervuar për procese të tjera, çka do ta shkaktonte përfundimin e padëshirueshëm të programit. Që një shembull i kodit për shtypjen e listës së fjalive:

```
DATA *p = pFillimi;

/* përderisa nuk është fundi i listës lidhëse */
while (p->pData != NULL)
{
    printf(p->pRreshti); /* shtype rreshtin */
    printf("\n");

    p = p->pData; /*vizito rreshtin tjetër që vijon*/
}
```

Natyrisht, para përfundimit të programit duhet ta lironi memorien e rezervuar për çdo objekt në listë, p.sh.:

```
DATA *pTemp;
DATA *p = pFillimi;

while (p->pData != NULL)
{
    /* liro memorien e rezervuar për fjalinë */
    free(p->pRreshti);

    pTemp = p;
    p = p->pData;

    /* liro memorien e rezervuar për objektin në listë */
    free(pTemp);
}
```

Siç po e vëreni, në vorbullën `while` përdorim një tregues të përkohshëm për ta liruuar memorien e rezervuar për objekt. Këtë e bëjmë me arsye, sepse sikur ta përdornim kodin e mëposhtëm:

```
while (p->pData != NULL)
{
    free(p->pRreshti);
    p = p->pData;
    free(p);          /* gabim */
}
```

atëherë do ta lironim memorien që tregohet nga treguesi p, kështu që udhëzimi për të parë se mos jemi në fund të listës lidhëse:

```
p->pData != NULL
```

nuk do të ishte i vlefshëm. (Pas lirimit të memories që tregohet nga treguesi p, e njëjta memorie mund të rezervohet nga proceset e tjera, gjë që natyrisht se nuk do të ishte e vlefshme për procesin ekzekutues).

Në programin për editor do të ishte e nevojshme të kemi një funksion, i cili pranon si parametër fjalinë e shkruar dhe ia bashkangjet fundit të listës lidhëse, p.sh.:

```
void Shtofjaline(const char* fjalia)
{
    DATA *pTemp;

    /* nëse lista lidhëse është e zbrazët */
    if (pFillimi == NULL)
    {
        pFillimi = (DATA *) malloc(sizeof(DATA));
        pFillimi->pRreshti = (char*)
            malloc(strlen(fjalia)+1);
        strcpy(pFillimi->pRreshti, fjalia);

        /* inico fundin e listës */
        pFillimi->pData = NULL;
    }
    else
    {
        /* gjeje fundin e listës */
        for (pTemp = pFillimi; pTemp->pData != NULL;
            pTemp = pTemp->pData)
        {
            pTemp->pData = (DATA *) malloc(sizeof(DATA));
            pTemp->pData->pRreshti = (char*)
                malloc(strlen(fjalia)+1);
            strcpy(pTemp->pData->pRreshti, fjalia);

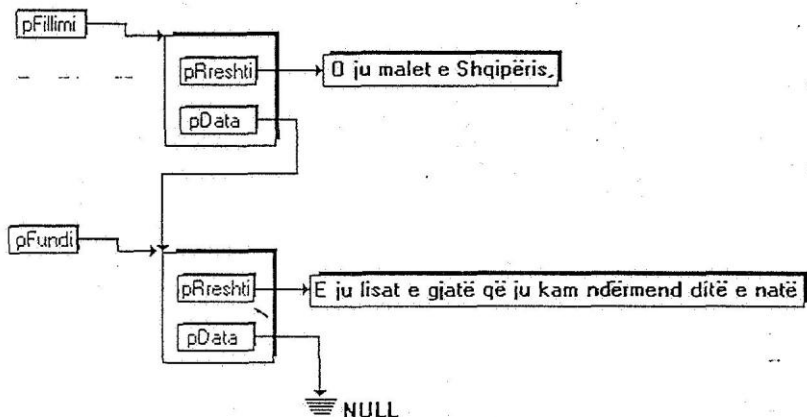
            /* inico fundin e listës */
            pTemp->pData->pData = NULL;
        }
    }
}
```

Në funksionin Shtofjaline mund të vëreni se kur lista nuk është e zbrazët, atëherë duhet ta gjejmë fundin e listës duke e shikuar çdo objekt në listë.

Mirëpo nëse lista është e gjatë, ky proces do ta ngadalësonte programin. Një zgjidhje do të ishte që përpos treguesit global `pFillimi` ta kemi edhe një tregues tjetër global të tipit `DATA` `pFundi` që tregon në fundin e listës.

Treguesi `pFundi` do të duhej të iniciohej me objektin e fundit, sa herë që ta shtojmë ndonjë objekt në listë ose ta heqim ndonjë objekt nga lista.

Nëse e marrim si shembull figurën e mëparshme, atëherë treguesi `pFundi` do të tregonte në objektin që përmban fjalinë: "E ju lisat e gjatë që ju kam ndërmend ditë e natë".



Me përdorimin e treguesit për fundin e listës, funksioni `Shtofjaline` do të shkruhej kështu:

```

void Shtofjaline (const char* fjalia)
{
    DATA *pTemp;

    /* nëse lista lidhëse është e zbrazët */
    if (pFillimi == NULL)
    {
        pFillimi = (DATA *) malloc(sizeof(DATA));
        pFillimi->pRreshti = (char*)
            malloc(strlen(fjalia)+1);
        strcpy(pFillimi->pRreshti, fjalia);

        /* inico fundin e listës */
        pFillimi->pData = NULL;

        /* meqë kemi vetëm një objekt në listë
            fundi i listës është i njëjti si fillimi */
        pFundi = pFillimi;
    }
}
  
```

```
else
{
    pTemp = pFundi;

    pTemp->pData = (DATA *) malloc(sizeof(DATA));
    pTemp->pData->pRreshti = (char*)
        malloc(strlen(fjalja)+1);
    strcpy(pTemp->pData->pRreshti, fjalja);

    /* inico fundin e listës */
    pTemp->pData->pData = NULL;

    /* inico treguesin pFundi me objektin e shtuar */
    pFundi = pTemp->pData;
}
}
```

Përpos listës lidhëse përdoren edhe tipet e tjera të njohura si rendi (queue), trangu (tree), b-tree etj., të cilat kanë si përparësitë e veta ashtu edhe mangësitë. Këto tipe, sikurse lista lidhëse, përdorin memorien dinamike dhe treguesit, mirëpo dallohen në mënyrën se si i organizojnë objektet dhe lidhjen në mes tyre. P.sh. tipi b-tree është i njohur për shpejtësinë e kërkimit të objekteve në listë dhe përdoret shumë në database dhe në programe, kur kërkimi i ndonjë objekti në listë duhet të jetë i shpejtë.

4.6 Treguesit në funksione

Në programet e shkruara në gjuhën C kemi raste kur nevojitet të pasojmë treguesin e funksioneve si parametër në funksione të tjera. Kjo mundëson krijimin e kodit gjenerik. Në gjuhën C++ pasimi i treguesve në funksione është shumë më pak i përdorshëm, për arsye se gjuha C++ mundëson metoda të tjera më efikase për krijimin e kodit gjenerik (p.sh. me anë të funksioneve virtuale ose klasëve abstrakte).

Në rastet kur kemi nevojë t'i pasojmë treguesit e funksioneve, është e udhës që së pari t'i definojmë tipet e funksioneve, të cilët mund të definohen kështu:

```
/* LPFIGURA definohet si funksion që nuk pranon asnjë parametër
dhe nuk kthen asnjë parametër */

typedef void (*LPFIGURA) (void) ;

/* LPRENDITJA; funksion që pranon një parametër int dhe një
char, por nuk kthen asnjë parametër */

typedef void (*LPRENDITJA) (int, char) ;

/* LPRENDITJA; funksion që pranon si parametër int dhe char dhe
kthen si parametër tipin int*/

typedef int (*LPRENDITJA) (int, char) ;
```

Në funksione duam ta pasojmë si parametër adresën e funksionit, prandaj definojmë si variabël tregues në tip të funksionit. Funksionet janë sikurse variablat *array* kur pasohen si parametra, prandaj nuk duhet përdorë operatorin *&* para emrit të funksionit, pra emri i funksionit është adresa e fillimit të funksionit.

Si shembull për tregues të funksioneve do të marrin programin për vizatimin e figurave gjeometrike. (Në kodin që vijon, trupi i funksioneve është lënë pa kod dhe vetëm me komente).

```
#include <stdio.h>

typedef void (*LPFIGURA) (void) ;

void vizato(LPFIGURA figura)
{
    /* këtu mund ta vizatoni dritaren ose diçka tjetër */
}
```

```

/* pastaj vizatoni figurën që e kryen funksioni figura */
(*figura) ();

/* variabla figura është tregues, prandaj përdorim
operatorin * para funksionit */
}

void rrethi (void)
{
    /* vizato rrethin këtu */
}

void katrori (void)
{
    /* vizato katrorin këtu */
}

int main (int argc, char* argv[])
{
    vizato(rrethi);
    vizato(katrori);

    return 0;
}

```

Ndoshta do të pyetni çfarë përparësie ka kodi i mësipërm në funksionin kryesor ndaj kodit që vijon, pra thirrjes së drejtpërdrejtë të funksioneve për vizatimin e figurave gjeometrike:

```

int main (int argc, char* argv[])
{
    rrethi();
    katrori();

    return 0;
}

```

Natyrisht se nuk ka kurrfarë përparësie, mirëpo kodi i mësipërm është vetëm për të ilustruar se si mund ta përdorim treguesin e funksioneve. P.sh. ta zëmë se kemi disa funksione për vizatimin e figurave të ndryshme gjeometrike dhe një *array* e cila i përmban treguesit e të gjitha funksioneve për figura gjeometrike. Atëherë, kodi për vizatimin e ndonjë skice, do të thjeshtësohej duke përdorë strukturat përsëritëse për t'i thirrë funksionet e ruajtura në *array*. Ta zëmë se i kemi funksionet për ta lëvizur kursorin e mausit (miut). Meqë kjo do të jetë specifike për sistemet operative, në vend që ta lëvizim kursorin, ne do ta shtypim komandën që funksioni do ta kryente.

```

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

#define MAX_OPCIONI      255

typedef int (*LPKOMANDA) (void) ;

typedef enum {FUND, JO_FUND} Statusi ;

typedef struct
{
    char      *emri;          /* emri i opcioneve */
    LPKOMANDA  komanda;       /* funksioni për opcion */
} Opcionet;

Statusi lart(void)
{
    printf("Levize kursorin lart\n");

    return JO_FUND;
}

Statusi poshte(void)
{
    printf("Levize kursorin poshte\n");

    return JO_FUND;
}

Statusi fund(void)
{
    printf("Fundi i programit\n");

    return FUND;
}

Statusi IPavlefshem(void)
{
    printf("Komanda e pavlefshme\n");

    return JO_FUND;
}

Opcionet meny[] = {
    "lart",          lart,
    "poshte",        poshte,
    "fund",          fund,
    "",              IPavlefshem
};

```



```

int main (int argc, char* argv[])
{
    char    opcioni[MAX_OPCIONI];
    int     iOpc;

    do
    {
        printf("Shtype opcionin: ");
        /* lexo opcionin e shtypur nga përdoruesi */
        gets(opcioni);

        /* tento derisa përdoruesi ta shtypë komanden
           e vlefshme, apo të arrijmë në fund të menysë */
        for (iOpc = 0; *meny[iOpc].emri != '\0'; iOpc++)
            if (strcmp(opcioni, meny[iOpc].emri) == 0)
                break;

        if (iOpc == MAX_OPCIONI)
            continue;

        while ((*meny[iOpc].komanda)());

    } while (1);

    return 0;
}

```

Funksioni main nuk do të ndryshojë sikur lista në meny të ishte shumë më e gjatë, apo sikur ta shtonim ndonjë element në meny, përderisa elementi i fundit (i pavlefshëm) mban *stringun* (vargun) e zbrazët.

Sikur të mos i përdornim treguesit e funksioneve, atëherë funksioni main do të shkruhej kështu:

```

int main (int argc, char* argv[])
{
    char    opcioni[MAX_OPCIONI];
    Statusi s;

    do
    {
        printf("Shtype opcionin: ");
        /* lexo opcionin e shtypur nga shfrytëzuesi */
        gets(opcioni);

        if (strcmp(opcioni, "lart") == 0)
            s = lart();
        else if (strcmp(opcioni, "poshte") == 0)
            s = poshte();
        else if (strcmp(opcioni, "fund") == 0)
            s = fund();
        else
            s = IPavlefshem();

    } while (s != FUND);

    return 0;
}

```

Sikur lista në meny t'i kishte me mija elemente, atëherë kodi për zgjedhjen e funksionit të duhur (në if dhe else) do të ishte disa faqe, kurse në rastin e parë do të ishte i njëjtë.

Në rastin e dytë, sikur të donim ta shtonim ndonjë element në meny (p.sh. për ta lëvizur kursorin djathtas) atëherë përpos që do të duhej ta shkruanim funksionin për ta lëvizur kursorin djathtas, do të duhej ta ndryshonim edhe funksionin main (në shprehjet if dhe else).

Në rastin e parë duhet vetëm ta shkruajmë funksionin për ta lëvizur kursorin djathtas dhe ta shtojmë një element në meny, kurse funksioni main nuk do të duhej ndryshuar.

Në situatat kur struktura e kodit është komplekse dhe ka gjasa të ndryshohet në të ardhmen, preferohet të përdoren strukturat gjenerike (me përdorimin e treguesve të funksioneve) me ç'rast shrytëzuesit nuk do të duheshin ta ndryshonin pjesën e kodit kompleks. Në këtë mënyrë i ikim paraqitjes së gabimeve në ndryshimin e kodit.

Treguesit në funksione përdoren edhe në parametrat e funksioneve në libraritë standarde, të cilat mundësojnë që përdoruesi i këtyre funksioneve të ketë zgjidhje të tjera, në varësi nga nevojat specifike.

Të supozojmë se kemi krijuar një librarë e cila përmban funksionin për renditjen e elementeve në listë me metodën e zgjedhur nga krijuesi i librarisë. Shfrytëzuesit e kësaj librarie mund ta furnizojnë treguesin në funksionin i cili bën renditjen e elementeve në array me metodën e zgjedhur nga vetë shfrytëzuesi⁵ dhe mënyrën e renditjes së elementeve të listës (p.sh. prej numrit më të vogël të numrit më i madh apo anasjelltas).

Normalisht funksioni rendit si dhe funksionet e tjera të përgjithshme, të përdorshme në shumë probleme, duhen të implementohen në fajll të veçantë, mirëpo për demonstrim e kemi definuar këtë funksion në të njëjtën fajll me funksionin main.

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

```
typedef void (*LPRENDITJA) (int lista[], int iNumri) ;
```

```
void quicksort(int lista[], int iPari, int iFundit)
```

```
{
    int      pivot;
    int      i, j;
    int      iTemp;

    if (iPari < iFundit)
    {
```

⁵ Në renditjen e elementeve në array ka disa metoda të njohura si *insert sort*, *quick sort*, *bubble sort* etj.. secila prej të cilave ka përparësitë dhe mangësitë e veta.

```

pivot = lista[iPari];
i      = iPari;
j      = iFundit;

while (i < j) {
    while (lista[i] <= pivot && i < iFundit)
        /*rrite i-në për ta kërkuar numrin e madh*/
        i++;
    while (lista[j] >= pivot && j > iPari)
        /*zvoglo j-në për ta kërkuar numrin e vogël*/
        j--;
    if (i < j) {
        iTemp = lista[j];
        lista[j] = lista[i];
        lista[i] = iTemp;
    }
    iTemp = lista[j];
    lista[j] = lista[iPari];
    lista[iPari] = iTemp;

    quicksort(lista, iPari, j - 1);
    quicksort(lista, j + 1, iFundit);
}

```

/* funksioni i përdorshëm nga shfrytëzuesit e librarisë,
që mundëson përdorimin e metodave të ndryshme për
renditjen e listës së furnizuar nga vetë shfrytëzuesit
apo përdorimin e renditjes bazë (quicksort) */

```

void rendit(int lista[], int iNumri, LPRENDITJA lpRendit)
{
    if (lpRendit != NULL) {
        (*lpRendit)(lista, iNumri);
        return;
    }
    quicksort(lista, 0, iNumri-1);
}

```

```

void bubblesort(int lista[], int iNumri)
{
    int i, j, iTemp;

    for (i = 0; i < iNumri; i++)
        for (j = i; j < iNumri; j++)
            if (lista[i] > lista[j]) {
                iTemp = lista[i];
                lista[i] = lista[j];
                lista[j] = iTemp;
            }
}

```

```
}  
void main (int argc, char* argv[])  
{  
    int lista[8] = { 99, 85, 83, 23, 12, 11, 11, 1};  
  
    /* rendite listën duke përdorë renditjen normale  
       të furnizuar nga vetë funksioni rendit */  
    rendit(lista, 8, NULL);  
  
    /* rendite listën duke përdorë renditjen  
       bubble sort */  
    rendit(lista, 8, bubblesort);  
}
```

Siç mund ta vëreni, në programin e mësipërm kemi bërë renditjen e listës duke përdorë dy metoda:

- metodën e furnizuar nga vetë funksioni rendit (quick sort), dhe
- metodën e furnizuar nga përdoruesi (bubble sort).

Ushtrime

1. Krijoni funksionin `InicoData` që pranon si parametër fjalinë (`const char *pFjalja`) dhe tregues të tipit `DATA`. Ky funksion duhet ta rezervojë memorien e nevojshme për fjalinë dhe për objektin e tipit `DATA` dhe ta inicojë treguesin e pasuar si parametër. Duhet të keni parasysh se parametri i dytë duhet të jetë tregues në tregues të tipit `DATA` (`DATA **ppData`) pasi që iniconi treguesin brenda në funksion (t'iu përkujtojmë se në gjuhën C pasohen kopjet e parametrave).

```
void InicoData(const char *pFjalja, DATA **ppData);
```

P.sh. nëse kemi treguesin `pFillimi` dhe dëshirojmë ta inicojmë me fjalinë fillestare, atëherë do ta thërrisim funksionin kështu:

```
InicoData(pFjalja, &pFillimi);
```

Pra në funksionin `InicoData` duhet pasuar adresën e treguesit.

2. Krijoni një funksion që pranon si parametër treguesin e tipit (`const DATA *pData`) dhe e liron memorien e rezervuar për objektin e treguar nga treguesi `pData` si dhe objektet e tjera që pasojnë në listën lidhëse.
3. Krijoni një funksion që pranon si parametër treguesin e tipit `DATA` (`const DATA *pData`) dhe shtyp fjalitë e treguara nga çdo objekt në listën lidhëse.
4. Në programin për lëvizjen e kursorit, shtoni funksionin për të lëvizur kursorin djathtas dhe majtas. Shtoni edhe elementet e nevojshme në meny për funksionet e shtuara.

Përmbledhje

Gjuhët C dhe C++ mundësojnë programimin e nivelit të ulët me anë të treguesve. Treguesit janë tipe që përmbajnë adresën fizike në memorie të ndonjë variable, apo ndonjë funksioni të definuar më parë. Treguesit nuk mund të përdoren të veçuara; ata duhet të tregojnë në ndonjë variabël apo funksion të definuar më parë. Zakonisht treguesit përdoren me variablat *array*, me ç'rast treguesit mundësojnë manipulimin me anëtarët e variablës *array* në mënyrë më efikase. Në gjuhën C, treguesit janë shumë të përdorshëm në manipulimin e *stringut* (vargut), si dhe në tejkalimin e problemit përmes pasimit të parametrave formalë.

Treguesit përdoren edhe në raste kur numri i të dhënave nuk është i ditur gjatë kompajlimit të programit, pra kur numri i të dhënave bëhet i ditur vetëm gjatë ekzekutimit të programit. Në këto raste treguesit përdoren për të treguar në memorien e rezervuar në mënyrë dinamike gjatë ekzekutimit të programit, për ruajtjen e të dhënave.

Programimi në C++

Pjesa e dytë

II

Gjuha C++

Gjuha C++ është shkruar nga Bjarne Stroustrup, duke trashëguar efikasitetin e gjuhës C dhe duke e kombinuar me metodat objekt-orientuese të gjuhës Simula. Gjuha C mund të quhet nënbashkësi e gjuhës C++, sepse pothuajse të gjitha programet e shkruara në gjuhën C mund të kompajlohen me kompajlerin e gjuhës C++.

Në gjuhën C++ i njëjti program mund të shkruhet në disa stile. Ta zëmë, programi mund të shkruhet me metodën procedurale si në C, Fortran dhe Pascal, apo në metodën objekt-orientuese, metodë që në fillim është përdorë në gjuhët si Simula dhe Smalltalk.

Programimi objekt-orientues kërkon që programeri ta shohë problemin duke i menduar pjesët e programit si "objekte" që sajajnë problemin, për sjelljet dhe tiparet e këtyre "objekteve", si dhe relacionet në mes tyre. P.sh. programi i cili simulon sistemin planetar të diellit, do t'i ketë objektet të cilat përfaqësojnë trupat qiellorë (diellin, tokën, hanën, Jupiterin etj). Sjelljet e planeteve do të ishin lëvizjet (shpejtësia e rrotullimit dhe drejtimi, shpejtësia e rrotullimit rreth diellit, këndi i rrotullimit etj). Secili planet ka madhësinë, masën dhe përbërjen. Secili planet mund të ketë satelitë, të cilët kanë sjelljet dhe tiparet e veta. Madhësia, masa, përbërja si dhe tiparet e tjera të planeteve, paraqesin atributet e objekteve të cilat i paraqesin planetet. Objektet kontrollojnë simulimin e sistemit diellor, kurse disa objekte kontrollojnë paraqitjen e tyre. Kodi i shkruar i këtij programi do t'i përmbante objektet e sistemit diellor, objektet kontrolluese, objektet ekspozuese si dhe relacionet ndërmjet tyre.

Programimi objekt-orientues mundëson trashëgiminë e sjelleve të objekteve të deklaruara më parë. P.sh. nëse kemi deklaruar sjelljet e planeteve, dhe nëse sjelljet e satelitëve janë të ngjashme me ato të planeteve, atëherë nuk ka nevojë që t'i deklarojmë këto sjellje, por vetëm t'i trashëgojmë prej sjelljeve të planeteve. Kjo gjë mundëson që programeri t'i shfrytëzojë ngjashmëritë ndërmjet objekteve dhe programi të jetë më i shkurtër, mundëson që programi të kodohet më shpejtë dhe mundësia e gabimeve të jetë më e vogël.

5.1 Nevoja për Programimin objekt-orientues

Kompjuterët filluan të përdoren në organizata të mëdha qysh në fund të viteve të 50-ta. Hardueri filloi të përparonte me një tempo shumë të shpejtë. Në Konferencën Internacionale për Inxhinjeringun e Softuerit, u theksua se

përderisa shpenzimet për harduer zvogëloheshin dukshëm, shpenzimet për prodhimin e programeve (softuerit) karakterizoheshin me këto veti:

- Rritja e shpenzimeve për prodhimin e programeve,
- Vonimi i prodhimit të programeve,
- Programet vështirë transferoheshin në makina të ndryshme,
- Shkalla e mospjekësive të programeve ishte shumë e lartë, dhe
- Programet shpesh nuk zgjidhnin problemin origjinal.

Arsyeja pse hardueri përparonte me një tempo shumë të shpejtë, kurse softueri vetëm sa komplikohet dhe shkalla e mospjekësive rritej, ishte se hardueri përbëhej prej komponenteve të ripërdorshme. P.sh.. disa prodhues koncentroheshin në prodhimin e procesorit (CPU-s), kurse disa prodhues koncentroheshin në prodhimin e monitorit etj. Pasiqë këto komponente ishin më të vogla sesa i tërë kompjuteri, koncentrimi në këto komponente mundësonte precizitetin më të madh dhe lokalizimin e problemeve të mundshme. Kjo mundësonte testimin e komponenteve veçmas, dhe bashkangjitja e këtyre komponenteve të testuara dhe të përpunuara garantonte suksesin e tërë sistemit.

Në softuer ishte e kundërta: programet komplikoheshin më tepër (nevoja për programe më të mëdha rritej), kurse përdorimi i komponenteve apo i ndonjë pjese të kodit të shkruar më parë ishte gati i pamundshëm.

Në vitet e 70-ta një grup shkencëtarësh të lëmisë së Informatikës (ndër ta edhe Dijkstra, Hoare dhe Wirth) shkruan një gjuhë programuese që mundësonte programimin e ashtuquajtur *programim strukturor*. Këto gjuhë kanë pasur një ndikim mjaft të madh në zhvillimin e softuerit. Mirëpo, ndërsa këto programe ishin të përshtatshme për prodhimin e programeve relativisht të vogla, teknika e këtyre gjuhëve nuk ishte e përshtatshme për programe të mëdha. P.sh. këto gjuhë nuk mundësonin përdorimin e funksioneve ekzistuese. Me fjalë të tjera, programet e reja duheshin shkruar prej fillimit. Problemet paraqiteshin për arsye se programet përdornin metodën procedurë-orientuese, metodë e cila përqëndrohet rreth një apo më shumë strukture të të dhënave të ruajtura në mënyrë globale dhe që përdoret në pjesë të ndryshme të programit. Çdo ndryshim i të dhënave globale shkaktonte efekte të papara në funksionet që i përdornin këto të dhëna.

Një përparim i dukshëm në lëmin e informatikës u arrit me zbulimin e gjuhëve modulare. Idea e kësaj metode ishte që sistemi të ndahej në sisteme të vogla (nënsisteme), të cilat nënsisteme do të mund të zëvendësoheshin pa i ndikuar nënsistemet e tjera.

5.2 Cikli i aplikacioneve

Hulumtuesit shkencorë në lëmin e informatikës kanë provuar se në ciklin e aplikacioneve më shumë kohë shpenzohet për gjetjen e gabimeve (për përmirësimin e programeve dhe për mirëmbajtje) sesa në vetë kodimin e programeve. Sipas disa të dhënave, rreth 50% e kohës së programimit shpenzohet për mirëmbajtje dhe për prodhim, kurse disa të dhëna të tjerë thonë se deri në 75% e kohës së programimit shpenzohet për mirëmbajtjen e programeve. Për këtë arsye nevojitet një metodë e programit që të ishte përshtatshme për krijimin e programeve si dhe për mirëmbajtjen e tyre. Kjo kishte rëndësi të madhe, sidomos për sistemet e mëdha, për prodhimin e të cilave qenë bërë shpenzime të mëdha dhe mirëmbajtja e programeve të t cilave do të ishte një zgjidhje afatgjate. Në të ardhmen këto sisteme nuk do të ishte e udhës që të zhvillohen nga fillimi për t'u përshtatur ambientit të ri të harduerit apo nevojave personale për funksione të ndryshme. Do të ishte udhës të zhvillohen vetëm pjesët e sistemit në të cilat kanë efekt ndërrimet e ambientit.

Mirëmbajtja e sistemeve zakonisht ndahet në tri grupe:

- **Adaptiv** - i cili merr rreth 18% të kohës së përgjithshme të mirëmbajtjes. Në këtë grup bëjnë pjesë ndryshimet e bëra në programe për shkak të ndryshimeve në ambientin e sistemit, siç janë rritja e kamatave dhe procedura e rritjes së kamatave në banka. Këto ndryshime reflektohen në program, kështu që edhe sistemi në të shumtën e rasteve duhet të ndryshohet. Zakonisht sistemet e reja tentojnë t'u ikin ndryshimeve të kodit, duke përdorë parametrat jashtë sistemit. Mirëpo një përqindje mjaft e madhe shpenzohet për mirëmbajtjen e këtyre parametrave.
- **Persosës** - rreth 65% e kohës së shpenzuar. Në këtë grup bëjnë pjesë ndryshimet e sistemit me synim përmirësimin e sistemit pa e ndryshuar funksionin e sistemit. Këto ndryshime përfshijnë përshejtimin e funksioneve të caktuara në raste kur sistemi është i ngadalshëm etj.
- **Korrektues** - rreth 17% e kohës së shpenzuar. Këto ndryshime bëhen për arsye të gabimeve të gjetura në sistem gjatë aplikimit të sistemit në jetën e përditshme.

5.3 Qëllimet e inxhinjeringut të softuerit

Qëllimet e inxhinjeringut të softuerit janë këto:

- **Funksioni.** Programet duhet t'i plotësojnë kërkesat në specifikimin e programit dhe kërkesat e shfrytëzuesve të programit.
- **Mirëmbajtja e lehtë.** Sistemi duhet të jetë i lehtë të mirëmbahet dhe ndryshimet e programeve nuk do të duheshin ta rregullonin strukturën e kodit.
- **Efikasiteti.** Programet duhen të jenë efikase në punën që kryejnë.
- **Të qenët të sigurtë dhe pa gabime.** Programet duhet të funksionojnë edhe në kushte jonormale. Programet duhet të shohin për gabime dhe të reagojnë me korrektësi ndaj këtyre gabimeve.
- **Përdorimi i lehtë.** Programet duhet të jenë të lehta të përdoren dhe të mësohen nga shfrytëzuesit.
- **Ripërdorimi.** Programet duhet të jenë të ripërdorshme dhe modulare. P.sh. ndryshimi i një module nuk do të ndikonte në modulet të tjera të programit.
- **Koha dhe shpenzimet.** Shpenzimet si dhe koha për prodhimin e softuerit duhet të jenë të parashikueshme.
- **Të qenët portativ.** Sistemi duhet të jetë portativ në platforma të ndryshme.
- **Programimi në grupe.** Gjuha programuese duhet ta mundësojë dhe ta përkrahë zhvillimin e programeve në grupe.

Kompleksiteti i softuerit është shkaku kryesor i të gjitha problemeve në programe. Abstraksioni i të dhënave tenton ta zvogëlojë kompleksitetin duke fshehur detajet e problemeve. Principi i fshehjes së informatave qëndron në fshehjen e detajeve, të cilat nuk do të duheshin të kishin efekt në pjesët e tjera të sistemit. Këto principe arrihen me anë të tipeve të të dhënave abstrakte të njohura si ADT (Abstract Data Types). Qëllimi kryesor i tipeve të të dhënave abstrakte është fshehja e detajeve dhe grumbullimi i funksioneve që manipulojnë me këto të dhëna.

Principet e përmendura më parë vlejnë edhe në problemet e jetës së përditshme, si p.sh. shumë shoferë të automjeteve dinë ta ngasin automobilin; ta startojnë, ta ndërrojnë shpejtësinë, ta ndalin automjetin etj. Mirëpo, nuk është e thënë se këta shoferë duhet të dinë se si është ndërtuar automjeti, në mënyrë që ta ngasin atë.

E njëjta gjë vlen për tipet abstrakte, me ç'rast shfrytëzuesit (programerët) duhet të dinë vetëm si t'i përdorin tipet abstrakte dhe funksionin që ato e

kryejnë, mirëpo nuk kanë nevojë të jenë të informuar për strukturën e këtyre tipeve të të dhënave abstrakte apo ndërtimin tyre të brendshëm. Gjuha C++ mundëson krijimin e tipeve abstrakte të të dhënave me anë të klasëve.

5.4 Libraritë Standarde

Libraritë standarde janë pjesë përbërëse e shumë kompajlerëve të gjuhës C++. Këto librari janë të shkruara në vetë gjuhën C++ dhe mundësojnë kodim më të shpejtë dhe më të lehtë të programeve më komplekse. Rekomandohet që funksionet dhe klasët që ndodhen në këto librari t'i përdorni më parë se ta shkruani nga fillimi një funksion apo klasë që kryen të njëjtën punë. Funksionet dhe klasët e librarive standarde janë të testuara dhe mundësojnë që programi të shkruhet me më pak gabime. Disa kompajlerë përmbajnë disa librari që janë specifike për sisteme të caktuara, si për shembull kompajleri Visual C++ përmban librari për programim në sistemin operativ Microsoft Windows. Po ashtu, disa kompajlerë që janë specifike për sistemin operativ UNIX përmbajnë librari që mundësojnë programimin për X Windows, etj. Në këtë libër nuk do të koncentrohemi në këto librari pasi që janë specifike për kompajlerë dhe sisteme operative të ndryshme.

Sikurse gjuha C, edhe gjuha C++ përmban libraritë standarde për shtypjen e të dhënave dhe leximin e tyre. Funksionet dhe klasët elementare për shtypjen e të dhënave dhe leximin e të dhënave janë të definuara në fajllin `iostream`. Ky fajll duhet të përfshihet në program me anë të direktivave të preprocesorit `#include`. P.sh. programi i shkruar më parë në gjuhën C për shtypjen e fjalës "Tungjatjeta!", do të ishte i vlefshëm edhe për kompajlerët e gjuhës C++, mirëpo ky program në gjuhën C++ do të shkruhej kështu:

```
// programi qe shtyp Tungjatjeta [ 1 ]
#include <iostream.h>

void main (void)
{
    // shtype Tungjatjeta ! [ 5 ]
    cout << " Tungjatjeta ! \n" ;
}
```

Rreshti 1 dhe 5 janë komente që nuk lexohen nga kompajleri gjatë interpretimit të kodit në kod objekti. Komentet përdoren vetëm për ta bërë programin më të lexueshëm.

5.4.1 Komentet

Në gjuhën C++ komentet shkruhen pas dy simboleve të njëpasnjëshme // si dhe brenda simboleve /* dhe */ (sikur në gjuhën C). Komentet e shkruara pas simboleve // mund të jenë të gjata vetëm një rresht. Nëse doni të keni komente më të gjata se një rresht dhe doni t'i përdorni këto simbole për koment, atëherë çdo rresht duhet ta parapirni me këto simbole:

```
// Komenti fillon në këtë rresht
// dhe mbaron në këtë rresht
```

Kurse, sipas stilit të gjuhës C, komentit mund të shkruhet si më poshtë:

```
/*
   Komenti fillon këtu dhe
   mbaron në rreshtin tjetër
*/
```

Njëri stil nuk është më i preferueshëm sesa tjetri, dhe përdorimi i tyre varet nga parapëlqimet personale.

5.4.2 Përdorimi i librarive standarde

Gjuha C++, siç thamë më parë, së pari filloi të përdoret në vitin 1983, mirëpo kjo gjuhë u standardizua në komitetin ANSI të vitit 1999. Gjatë këtyre viteve gjuha C++ është përmirësuar aq shumë, saqë ka dallime të mëdha në mes të kompajlerëve të implementuar në ditët e hershme të gjuhës C++ dhe kompajlerëve të implementuar kohët e fundit. Njëri prej ndryshimeve është edhe paraqitja e komandës namespace (lexo: nejmspejs). Para implementimit të kësaj komande në gjuhën C++, zakonisht përfshihen libraritë standarde duke përdorur direktivat e preprocesorit #include dhe të fajllit të librarisë p.sh. <iostream.h>. Kjo mjaftonte për përdorimin e klasëve dhe funksioneve në libraritë standarde. Mirëpo pas paraqitjes së komandës namespace, libraritë standarde për shtypjen dhe leximin e të dhënave mund të përfshihen me direktivën e preprocesorit:

```
#include <iostream>
```

d.m.th. fajllit `iostream` pa prapashtesën ".h". Për t'i përdorë klasët dhe funksionet e kësaj librarie si më lart, duhet të përdorni edhe direktivën:

```
using namespace std;
```

Tash e tutje do ta përdorim stilin e ri, pasi që ky stil është më i pastër dhe shtyn në përdorimin e drejtë të gjuhës C++.

Pra, kodi për shtypjen e fjalës "Tungjatjeta!" tani do të ishte:

```
#include <iostream>

void main ()
{
    std::cout << "Tungjatjeta!\n";
}
```

ose

```
#include <iostream>

using namespace std;

void main ()
{
    cout << "Tungjatjeta!\n";
}
```

Siç e shihni nga shembulli i mësipërm, në gjuhën C++ për shtypjen e të dhënave kemi përdorë objektin cout (lexo: siout). Objekti cout (akronim ky i formuar nga fjalët angleze Console OUTput) shtyp të dhënat në mekanizmin e sistemit për ekspozimin e të dhënave (i cili zakonisht është monitori). Duhet ta keni parasysh se mekanizmi i sistemit për ekspozim nuk është gjithnjë monitori. P.sh. në sistemin operativ UNIX, administratori i sistemit mund ta drejtojë shtypjen e të dhënave drejt në printer apo në mekanizma të tjerë, kështu që objekti cout do t'i shtypë të dhënat në mekanizmin e sistemit të definuar për ekspozim. Objekti cout mund të përdoret për shtypjen e të dhënave të tipit numër (int, float, double), string (varg) si dhe tipe të definuara nga vetë përdoruesit.

Tipet e definuara nga përdoruesit duhet ta përmbajnë operatorin << që merr si parametër objektin ostream dhe tipin e definuar nga përdoruesi, p.sh.:

```
class Data
{
    ...

    friend ostream& operator<<(ostream& os, const Data& dt);
};
```

Për komandën `friend` dhe funksionin e kësaj komande do të flasim më vonë, mirëpo këtu vetëm demonstruam se `cout` mund t'i shtypë edhe tipet e definuara nga përdoruesit. Implementimi i operatorit `<<` cakton se si shtypen këto tipe.

Libraritë standarde për paraqitjen e të dhënave përmbajnë disa funksione manipuluese që përcaktojnë formatin dhe "dekorimin" e të dhënave të shtypura me `cout`, p.sh..

```
cout << "Shuma e numrave 5 dhe 7 është "
      << setw(5)
      << 12;
```

Funksioni `setw` cakton hapësirën që numri do të zërë pas shtypjes në mekarizmin për ekspozim. Numri 12 zë vetëm dy hapësira pas shtypjes (d.m.th. një hapësirë për shifrën 1 dhe tetrën për shifrën 2), mirëpo pas ekzekutimit të funksionit `setw(5)` numri 12 zë 5 hapësira, me ç'rast hapësirat e papërdorura mbeten të zbrazëta. Pra rezultati i kodit të mësipërm do të ishte:

```
Shuma e numrave 5 dhe 7 është 12
```

Nëse parametri i funksionit manipulues `setw` është më i vogël se numri i shifrave të numrave të shtypur, atëherë funksioni manipulues `setw` injorohet.

Funksioni `setprecision` (në standardizimin më të ri të gjuhës C++ emri i këtij funksioni është ndërruar në `setf`) përdoret për caktimin e numrave realë, pra për caktimin e shifrave pas presjes dhjetore. Libraria standarde përmban disa funksione të tjera për manipulimin e formatit të të dhënave, të cilat janë paraqitur në tabelën e mëposhtme. Këto funksione janë përmendur shkurtimisht për funksionin që kryejnë, ndërsa i është lënë lexuesve t'i praktikojnë dhe t'i testojnë sjelljet e tyre në ambiente të ndryshme.

Funksioni	Veprimi që kryen funksioni
<code>setw(n)</code>	Ndërron hapësirën e të dhënave në <code>n</code>
<code>Setprecision(n)</code>	Ndërron numrin e shifrave pas presjes dhjetore për numrat realë.
<code>Setiosflags(konstanta)</code>	Ndërron parametrat për shtypjen e të dhënave (shih tabelën më poshtë).
<code>Dec</code>	Ndërron shtypjen e numrave në numra decimalë.
<code>Oct</code>	Ndërron shtypjen e numrave në

	numra oktalë.
Hex	Ndërron shtypjen e numrave në numra heksadecimalë.

Funksionet për manipulimin e formatit të të dhënave të shtypura.

Funksioni `setiosflags` pranon si parametër konstantat e definuara në objektin `ios`. Këto konstanta tregojnë se si të shtypen të dhënat gjatë përdorimit të objektit `cout`, p.sh.:

```
cout << " | |"
      << setw(12)
      << setiosflags(ios::left)
      << 1999
      << " | |" ;
```

do të shtypte

```
| |1999      | |
```

Konstantja	Kuptimi
<code>ios::showpoint</code>	Gjithmonë shtyp presjen për numrat dhjetorë si dhe 6 shifra pas presjes dhjetore.
<code>ios::showpos</code>	Shtyp + para numrave pozitivë.
<code>ios::fixed</code>	Numrat shtypen me maksimum 3 shifra para dhe 2 shifra pas presjes dhjetore. Numrat e mëdhenj paraqiten me eksponent.
<code>ios::scientific</code>	Shtyp numrat në formatin me eksponent.
<code>ios::dec</code>	Shtyp numrat në formatin e numrave decimalë.
<code>ios::oct</code>	Shtyp numrat në formatin e numrave oktalë.
<code>ios::hex</code>	Shtyp numrat në formatin e numrave heksadecimalë.
<code>ios::left</code>	Të dhënat e shtypura renditen në të majtë.
<code>ios::right</code>	Të dhënat e shtypura renditen në të djathtë.

Tabela për konstantat e përdorura në funksionin `manipulates setiosflags`.

Futja e të dhënave në gjuhën C++ është e ngjashme me mënyrën e shtypjes së të dhënave. Për futjen e të dhënave duhet ta përdorni objektin `cin` (akronim nga fjalët Console INput). Objekti `cin` përdoret për leximin e të dhënave të tipit të definuar në kompajler (`int`, `char`, `long`, `float` etj) si dhe për tipet e të dhënave të definuara nga vetë shfrytëzuesit që përmbajnë operatorin `>>` me parametër objektin `istream`, p.sh.:

```
class Data
{
    ...
    friend istream& operator>>(ostream& is, Data& dt);
};
```

Leximi i të dhënave bëhet nga tastatura e kompjuterit dhe ruhet në variablat e definuara më parë p.sh.:

```
#include <iostream>

using namespace std;

void main ()
{
    int i;

    cin >> i;

    cout << "Kenë shtypur numrin " << i ;
}
```

Objekti `cin` mund të përdoret për leximin e numrave të njëpasnjëshëm të ndarë me hapësirë. P.sh.:

```
#include <iostream>

using namespace std;

int main ()
{
    int numri1, numri2, numri3;

    cout << "Shtypni tre numra të ndarë me hapësirë\n";

    cin >> numri1 >> numri2 >> numri3;

    cout << "Kenë shtypur numrat "
         << numri1
         << " "
         << numri2
         << " "
         << numri3
         << "\n";
}
```

```

        << numri2
        << " dhe "
        << numri3;

    return 0;
}

```

Nëse gjatë ekzekutimit do të shtypni rreshtin:

```
12 5 7
```

atëherë variabla numri1 do ta mbajë vlerën 12, variabla numri2 do ta mbajë vlerën 5 dhe variabla numri3 do të mbajë vlerën 7.

Objekti cin e njeh tipin e variablës së definuar dhe thirr funksionin (operatorin) ngarkues për atë tip. P.sh. në shembullin e mëposhtëm i lexojmë dy numra të tipeve të ndryshme:

```

#include <iostream>

using namespace std;

void main ()
{
    int numriNatyrar;
    float numriReal;

    cout << "Shtypni një numër natyral :";

    cin >> numriNatyrar;

    cout << "Shtypni një numër real :";

    cin >> numriReal;

    cout << "Keni shtypur numrat "
         << numriNatyrar
         << " dhe "
         << numriReal;
}

```

Objekti cin përmban funksionet anëtare, p.sh. funksionin get për lexim të vetëm një shkronje apo getline për leximin e rreshtit, p.sh.:

```

#include <iostream>

using namespace std;

```

```
void main ()
{
    char c;
    char rreshti[255];

    cout << "Zgjedhni njëzërn prej opcioneve në meny:\n";

    cout << "c          Lexoni vetëm një shkronjë\n";
    cout << "r          Lexoni tërë rreshtin\n";

    c = cin.get();

    switch (c)
    {
        case 'c':
            // lexo vetëm një shkronjë
            c = cin.get();
            cout << "Kenë shtypur shkronjën " << c;
            break;

        case 'r':
            // lexo rreshtin
            cin.getline(rreshti, 255);
            cout << "Kenë shtypur rreshtin \n"
                 << rreshti;
    }
}
```

5.5 String (vargu)

Deri tani për ruajtjen e një stringu kemi përdorë tipin `char array` apo `char*`, në varësi të inicimit të variablës me variabël të tipit `char array` ose konstantë.

Në gjuhën C++ për ruajtjen e stringut mund ta përdorim klasën `string` të definuar në librarinë standarde me namespace `std`, në fajllin `string`, p.sh.:

```
#include <iostream>
#include <string>

using namespace std;

void main ()
{
    string s = "Tungjatjeta!";

    cout << s ;
}
```

Klasa `string` e ka lehtësuar punën shumë në krahasim me përdorimin e mëparshëm të stringut në stilin e gjuhës C (të ruajtur në `array` që përfundon me shkronjën 0). Në gjuhën C, nëse stringu nuk përfundonte me numrin zero, atëherë çdo manipulim më atë string me siguri do të shkaktonte gabim gjatë ekzekutimit të programit. Mirëpo, me përdorimin e klasës `string` në gjuhën C++, programerët s'kanë pse të brengosen se si ruhet stringu dhe a mbaron apo jo me 0, etj. Manipulimi me klasën `string` është shumë më i lehtë. Për ilustrim, më poshtë është kodi i programit për bashkimin e dy vargjeve (të klasës `string`) një mbas tjetrit:

```
#include <iostream>
#include <string>

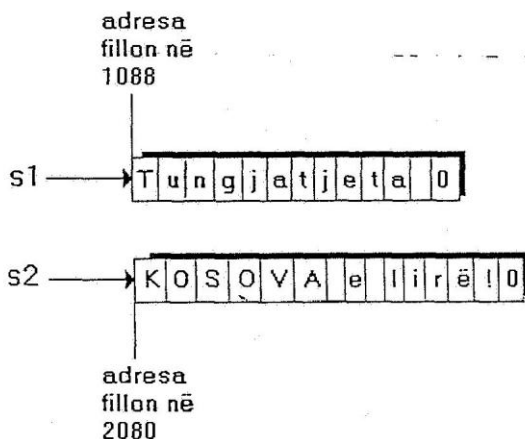
using namespace std;

void main ()
{
    string s1 = "Tungjatjeta ";
    string s2 = "Kosova e lirë!";

    string s3 = s1 + s2;

    cout << s3 ;
}
```

Siç po e vëreni, në shembullin e mësipërm kemi përdorë operatorin + për bashkëngjitjen e dy vargjeve (stringëve), porsikur t'i mbledhim dy integerë. Në gjuhën C kjo nuk do të ishte e mundur, pasi që variablat *s1* dhe *s2* janë adresa (tregues të tipit char). Shuma e këtyre variablave do të ishte një adresë tjetër që do të kishte të dhëna krejtësisht të ndryshme nga ajo që do të pritnit të përfitoni. P.sh. të supozojmë që adresa e treguesit *s1* është 1088 dhe e treguesit *s2* 2080 e paraqitur në mënyrë figurative në figurën që vijon:



Atëherë variabla tregues *s3* do të ketë vlerën 3168, adresë e cila do të përmbante të dhëna të pavlefshme apo të dhëna të ruajtura në variablat të tjera, adresa e të cilave është 3168. Çdo manipulim apo tentim i leximit të treguesit *s3*, me siguri do të shkaktonte gabim gjatë ekzekutimit të programit. Në gjuhën C, bashkëngjitja e stringëve do të mund të bëhej me anë të funksioneve të librarisë standarde, p.sh..

```
char s3[100];
char* s1 = "Tungjatjeta ";
char* s2 = "KOSOVA e lirë!";

strcpy(s3, s1);
strcat(s3, s2);

printf(s3);
```

Siç mund ta vëreni, në gjuhën C++ kodi për të kryer të njëjtën punë është më i qartë dhe më i shkurtër. Mirëpo për ta shkruar këtë kod nevojitet klasa përkatëse që përmban funksionet e duhura (në këtë rast kemi klasën `string`

e cila është definuar në libraritë standarde dhe përmban funksionet e duhura për manipulim me vargje simbolesh). Kjo klasë implementon operatoin + që mundëson bashkëngjitjen e stringëve (shih për operatorët dhe funksionet ngarkuese në kapitullin për klasët). Mënyra e implementimit të klasës `string` apo e ndonjë klase tjetër, nuk do të duhej ta brengoste përdoruesin e tyre. Përdoruesit duhet të jenë të informuar për funksionet dhe ndonjëherë edhe për variablat që i përkasin klasëve që përdorin, në mënyrë që të manipulojnë me vlerat e klasëve të ndryshme. P.sh. klasa `string` mund t'i ketë të njëjtat funksione anëtare (të njëjtin *interfejs*) mirëpo mund të implementohet në mënyra të ndryshme. Kështu, brenda në klasën `string`, të dhënat mund të ruhen si në *stringun* e gjuhës C (`char array` me përfundim me zero), listë të germave etj., mirëpo funksioni i jashtëm i klasës `string` do të ishte i njëjtë pavarësisht nga metoda e zgjedhur për implementimin e kësaj klase.

Përmbledhje

Në historinë e programimit është vërtetuar se sukseset e projekteve bazohen në përdorimin e kodit të shkruar dhe të testuar më parë, ndarjen e problemeve në nënprobleme etj.

Gjuha C, sikurse dhe gjuhët e tjera të ngjashme, mundësonin përdorimin e kodit të shkruar dhe të testuar më parë, nëpërmjet librave që përmbajnë funksionet e përdorshme për zgjidhjen e shumë problemeve. Mirëpo pasi që projektet dita-ditës po komplikohen gjithnjë e më shumë (programet kryejnë më shumë funksione etj.) filloi të ndihej nevoja për ndarjen e problemeve në metoda të tjera për zgjidhjen e problemeve në mënyrë sa më efikase. Metoda objekt-orientuese është metodë e cila është e përshtatshme për projekte të komplikuar, metodë e cila mundëson ndarjen e komponenteve të ripërdorshme për probleme të tjera. Komponentet e ripërdorshme kanë përparësi ndaj funksioneve të ripërdorshme duke qenë se komponentet i grumbullojnë të gjitha funksionet që përdoren për to, dhe e lehtësojnë zgjidhjen e përgjithshme të problemeve. Gjuha C++ është gjuhë programuese që trashëgoi efikasitetin e gjuhës C si dhe vuri në zbatim metodën objekt-orientuese, gjë që bëri që gjuha C++ të jetë më e preferuara për zgjidhjen e problemeve të shumë projekteve.

Klasët (class)

Programerët në gjuhën C++ janë në gjendje të krijojnë tipet e veta dhe t'i përdorin ato sikurse tipet e definuara nga gjuha C++.

Të gjitha programet të përdorura në shembujt e mëparshëm përdorin tipet e thjeshta si `int`, `float` etj. Këto tipe i kanë operacionet aritmetike `+`, `-` etj.

Mirëpo në jetën e përditshme kemi numër të pafund të tipeve, dhe gjuha C++ nuk mund t'i ofrojë të gjitha. Siç kemi përmendur më parë, gjuha C++ lejon deklarimin e tipeve të reja dhe të operacioneve të tyre. Kjo arrihet me anë të komandës `class`, e cila përfaqëson tipin e ri dhe si dhe operacionet e këtij tipi. Në mënyrë që të mësojmë më lehtë për klasët, le ta krijojmë vetë një tip të ri, tipin kohë.

6.1 Tipi i ri kohë

Për t'u njohur me tipin ndërtues `class`, kemi marrë si shembull tipin e ri kohë. Koha përfaqësohet me anë të orëve dhe minutave. Ju mund ta mendoj vlerën e tipit kohë si të ndarë në dy integjerë: i pari ora që mund të ketë vlera prej 0 deri 23, si dhe i dyti minutat me vlera të mundshme 0 deri 59.

ora	15
minutat	53

ora	0
minutat	31

ora	23
minutat	45

Disa vlera të tipit kohë

Vlera e tipit kohë:

ora	15
minutat	11

përfaqëson kohën 3:11 mbasdite .

Për tipin kohë do t'i deklarojmë tri operacione të vlefshme.

- Mbledhja

Ora	15	+ 20 =
Minutat	11	

ora	15
minutat	31

Ora	15	+ 70 =
Minutat	11	

ora	16
minutat	21

- Zbritja

Ora	15	- 12 =
Minutat	32	

ora	15
minutat	20

Ora	15	- 20 =
Minutat	11	

ora	14
minutat	51

- Shtypja e kohës në formë standarde

Cout <<

ora	10
minutat	30

shtyp 10:30 am

Cout <<

Ora	22
Minutat	43

shtyp 10:43 pm

Komanda `class` jua bën të mundshme t'i programoni operacionet për tipet e reja të deklaruara, temë kjo për të cilën do të flasim në këtë kapitull si dhe në kapitujt e ardhshëm.

6.2 Definimi i klasës (class)

Siç përmendëm më sipër, tipet e reja deklarohen me përdorimin e komandës class. Sintaksa e definimit të klasës me anë të shprehjes class ka këtë formë:

```
class < emri >
{
    private :
        < të dhënat anëtare >
    public :
        <definimi i konstruktit të klasës >
        < funksionet anëtare të klasës >
};
```

Definimet e klasës zakonisht gjenden në fillim të programit, mirëpo deklarimi i tyre mund të bëhet kudo në program, përderisa klasët janë të deklaruara para se të përdoren.

<të dhënat anëtare> përfaqësojnë të dhënat (variablat) anëtare të klasës. Në shembullin tonë të dhënat anëtare për klasën koha do t'i definonim kështu:

```
class koha
{
    private:
        int ora, minutat;
    public:
        <definimi i konstruktit të klasës>
        <funksionet anëtare të klasës>
};
```

Me përdorimin e variablave anëtare, i tregojmë kompajlerit të gjuhës C++ se vlerat e tipit koha përfaqësohen me dy integjerë: ora dhe minutat. Me fjalë të tjera, çdo objekt i tipit koha përmban dy integjerë. Për ta kompletuar definimin e klasës koha duhet t'i definojmë funksionet anëtare të klasës. Këto funksione janë për mbledhjen, zbritjen dhe shtypjen e klasës koha. Në shembullin tonë, le t'i quajmë këto funksione shtojMinutat për mbledhje, zbritMinutat për zbritje dhe shtypeKohen për shtypjen e objektit koha. Deri tani klasa koha duket kështu:

```
class koha
{
    private:
        int ora, minutat;
    public:
        koha( int o , int m) // konstruktori i klasës koha
        { ora = o; minutat = m; }
```

```

koha shtojiMinutat ( unsigned int m ) ;
koha zbritiMinutat ( unsigned int m ) ;
void shtypeKohen ( ) ;
};

```

thirrja e funksionit anëtar të objektit nga klienti njihet si dërgimi i porosisë së klientit në objekt dhe objekti njihet si pranues i porosisë.

Tani do t'i shqyrtojmë funksionet anëtare të objektit koha. Objekti koha pasi ta pranojë porosinë shtojiMinutat(m) duhet të kthejë objekt të ri të tipit koha që përfaqëson kohën pas shtimit të numrit m të minutave. Një zgjidhje e thjeshtë e këtij problemi është shndërrimi i orëve në minuta: ora * 60 dhe mbledhja e rezultatit me minutat m, pastaj shndërrimi i anasjelltë i minutave të rezultuara në orë. Kjo zgjidhje nuk është e preferueshme, sepse nëse m është numër i madh, pas mbledhjes ora mund të ketë vlerë më të madhe se 23, vlerë e papranueshme për objektin koha. Ky problem mund të zgjidhet duke pasur parasysh se minutatTotale duhet të ketë vlerën më të vogël se 24 x 60.

```

koha shtojiMinutat(unsigned int m)
{
    int minutatTotale = ( 60 * ora + minutat + m ) % (24*60);
    return koha ( minutatTotale/60, minutatTotale % 60 );
}

```

Zgjidhja për funksionin zbritiMinutat është e ngjashme me atë të shtojiMinutat, vetëm se në këtë funksion variablës minutatTotale duhet t'i shtohet vlera 24 x 60 nëse vlera totale e variablës minutatTotale është negative.

```

koha zbritiMinutat(unsigned int m)
{
    int minutatTotale = ( 60 * ora + minutat - m ) % (24*60);
    if (minutatTotale < 0)
        minutatTotale = minutatTotale + 24*60 ;
    return koha ( minutatTotale/60, minutatTotale % 60 );
}

```

Zgjidhja për funksionin shtypeKohen:

```

void shtypeKohen ( )
{
    if ( ora == 0 )
        cout << 12 ;
}

```

```

        else if ( ora > 12 )
            cout << ora - 12 ;
        else
            cout << ora ;

        cout << " : " << minutat ;
        if ( ora < 12 )
            cout << " am";
        else
            cout << " pm" ;
    }

```

Definimi komplet i klasës koha është paraqitur në kodin që vijon:

```

class koha
{
private:
    int ora, minutat;
public:
    koha( int o , int m) { ora = o; minutat = m; }
    koha shtojiMinutat ( unsigned int m) ;
    koha zbritiMinutat ( unsigned int m) ;
    void shtypeKohen ( ) ;
};

koha shtojiMinutat(unsigned int m)
{
    int minutatTotale = ( 60 * ora + minutat + m )
                        % (24*60);
    return koha ( minutatTotale/60, minutatTotale % 60 );
}

koha zbritiMinutat(unsigned int m)
{
    int minutatTotale = ( 60 * ora + minutat - m )
                        % (24*60);
    if (minutatTotale < 0 )
        minutatTotale = minutatTotale + 24*60 ;
    return koha ( minutatTotale/60, minutatTotale % 60 );
}

void shtypeKohen ( )
{
    if ( ora == 0 )
        cout << 12 ;
    else if ( ora > 12 )
        cout << ora - 12 ;
    else
        cout << ora ;
    cout << " : " << minutat ;
    if ( ora < 12 )

```

```

        cout << " am";
    else
        cout << " pm" ;
}

```

6.3 Konstruktoret dhe Dekonstruktoret

Konstruktoret dhe deskonstruktoret janë funksione speciale të klasës. Zakonisht këto "funksione" i tregojnë kompajlerit si t'i krijojë tipin e ri dhe sasinë e memories së nevojitur për të dhënat anëtare të këtij tipi, si dhe si të lirojë memorien e rezervuar për të dhënat anëtare. Kur shfrytëzuesi e deklaron një -variabël të tipit të ri, atëherë kompajleri automatikisht e thirr konstruktorin bazë.

Konstruktori ka emrin e njëjtë sikur klasa e deklaruar dhe nuk kthen asnjë tip vlere (as void) si dhe mund të ketë çfarëdo numri të parametrave të çfarëdo tipi. Konstruktori bazë është konstruktori pa asnjë parametër. Nëse gjatë definimit të klasës së re nuk deklarojmë asnjë konstruktor, atëherë kompajleri krijon automatikisht një konstruktor bazë (pa parametra).

Destruktori është e kundërta e konstruktorit, po ashtu funksion special, i cili paraprihet me shenjën ~ dhe me emrin e klasës. Kompajleri thërret destruktoren e çdo tipi kur variabla e atij tipi del nga hapësira e veprimtarisë.

Për shembull në klasën koha konstruktori dhe destruktori do të definoreshin kështu:

```

class koha
{
    private:
        int ora, minutat;
    public:
        koha( int o , int m);           // konstruktori
        ~koha();                       // destruktori
};

```

Klasët mund të kenë më shumë se një konstruktor, përderisa numri i parametrave apo tipi i tyre ndryshon. Mirëpo klasët mund të kenë vetëm një destruktur, për arsye se destruktoret nuk thirren në kod. Destruktoret thirren nga kompajleri në mënyrë indirekte kur objekti del nga hapësira e veprimtarisë.

Variablat anëtare të inicuar në konstruktor automatikisht thirren konstruktorin bazë të tyre e pastaj inicohen me vlerat përkatëse duke thirrur konstruktorin shndërrues apo kopjues.

Konstruktorët dhe destruktoret nuk trashëgohen nga klasët e nivelit më lartë, mirëpo këto thirren në mënyrë indirekte nga klasët e nivelit më të lartë. P.sh. konsidero shembullin që vijon:

```
#include <iostream>

using namespace std;

class klasaA
{
public:
    klasaA();
    ~klasaA();
};

klasaA::klasaA()
{
    cout << "Konstruktori i klasës \"klasaA\" "
          << endl;
}

klasaA::~~klasaA()
{
    cout << "Destruktori i klasës \"klasaA\" "
          << endl;
}

int main (int argc, char* argv[])
{
    klasaA a;

    return 0;
}
```

Edhe pse në funksionin main nuk shtypim asgjë ky program do t'i shtyp këta rreshta:

```
Konstruktori i klasës "klasaA"
Destruktori i klasës "klasaA"
```

Nëse e deklarojmë klasën klasaB me prejardhje nga klasa klasaA do të vërehet se konstruktori dhe dekonstruktori i klasës klasaA do të thirret në mënyrë indirekte nga kompajleri, p.sh.:

```
class klasaB :public klasaA
{
public:
    klasaB();
    ~klasaB();
}
```

```

};

klasaB::klasaB()
{
    cout << "Konstruktori i klasës \"klasaB\" \" << endl;
}

klasaB::~klasaB()
{
    cout << "Destruktori i klasës \"klasaB\" \" << endl;
}

int main (int argc, char* argv[])
{
    klasaB b;

    return 0;
}

```

Kodi i mësipërm do t'i shtypë këta rreshta:

```

Konstruktori i klasës "klasaA"
Konstruktori i klasës "klasaB"
Destruktori i klasës "klasaB"
Destruktori i klasës "klasaA"

```

Edhe pse në funksionin main kemi deklaruar vetëm variablën e tipit `klasaB`, konstruktori dhe destruktori i klasës `klasaA` është thirrë në mënyrë indirekte. Siç e vëreni, konstruktori i klasës `klasaA` thirret para konstruktorit të klasës së nivelit më të ulët, p.sh. klasës `klasaB`. Mirëpo destruktori i klasës `klasaA` thirret pas destruktorit të klasës `klasaB`. Këtë renditje të thirrjes së konstruktorit dhe të destruktorit duhen ta kemi parasysh gjatë definimit të tipeve të reja (zakonisht në raste kur rezervoni memorien në mënyrë dinamike për variablat anëtare të klasës).

6.3.1 Definimi i funksioneve të klasëve

Në faqet e mëparshme përmendëm klasën `string`, e cila është e definuar në libraritë standarde dhe nuk ka nevojë të rridefinohet. Mirëpo, për të demonstruar se si krijohen tipet e reja dhe problemet që duhen zgjidhur gjatë krijimit të këtyre tipeve, do ta krijojmë klasën `string` duke e shpjeguar këtë klasë hap pas hapi.

Funksionet përbërëse të klasëve (funksionet brenda në klasë) definojnë operacionet e objekteve që përfaqësohen me anë të klasës. Në gjuhën C++ klasët janë sikur të definojmë funksionet brenda në `struct`, prandaj klasët

njihen edhe si "struct me funksione", përderisa në gjuhën C nuk lejohet definimi i funksioneve brenda në struct. Funksionet e definuara në klasë njihen si funksione anëtare të klasës. P.sh. për klasën string funksionet anëtare të kësaj klase janë pjesë përbërëse e kësaj klase dhe nuk ka nevojë të përdorim ndonjë funksion nga libraritë të tjera standarde apo të definuara nga vetë përdoruesit, në mënyrë që të manipulojmë me string. D.m.th. të gjitha funksionet që nevojiten për manipulim me ndonjë tip të definuar mund të jenë pjesë përbërëse e klasës që definon tipin e të dhënave.

Vëreni, ta zëmë, funksionin që kthen gjatësinë e stringut. Në gjuhën C ishte e nevojshme të përfshihej fajlli <string.h> për përdorimin e funksionit strlen. Në gjuhën C++ funksioni për gjetjen e gjatësisë së stringut mund të jetë funksion anëtar i klasës string, p.sh.:

```
class String
{
    int    m_iGjatesia;
    char*  m_pString;

public:
    int    gjatesia();
}
```

Këtu kemi definuar klasën string sa për ilustrim, me dy variabla anëtare që e ruajnë gjatësinë e stringut dhe vetë stringun (në stilin e gjuhës C) si dhe funksionin anëtar gjatesia që kthen gjatësinë e stringut. Duhet ta keni parasysh se klasa string mund të definohet dhe të implementohet në disa mënyra dhe se definimi i mësipërm nuk do të thotë se është definimi më i favorshëm për klasën string. P.sh.. konteksti i stringut mund të ruhet në array si p.sh.:

```
class String
{
    int    m_iGjatesia;
    char    m_pString[100];

public:
    int    gjatesia();
}
```

Mirëpo këtu kemi një kufizim, sepse nuk mundemi të kemi string më të gjatë se 99 simbole (duke llogaritur edhe zeron që e përfundon stringun). Nëse përdorim treguesin në char (char*) atëherë duhet të rezervojmë memorie të re për ta ruajtur stringun dhe pasi të përfundojmë me këtë objekt, duhet ta lirojmë memorien e rezervuar. Vetëm të përkujtojmë se në C++ për ta

rezervuar memorien, përdorim komandën `new` dhe për ta liruar memorien përdorim komandën `delete`.

Funksioni `gjatesia` është anëtar i klasës `String` dhe mund të thirret vetëm nga instanca të objektit të tipit `string`, p.sh..

```
String s = "Tungjatjeta!" ;
int iGjatesia = s.gjatesia();

cout << "Gjatësia e stringut \"Tungjatjeta!\" është : "
      << iGjatesia;
```

Funksionet anëtare të klasës mund të manipulojnë me të gjitha të dhënat anëtare të klasës, dhe në rastin e mësipërm funksioni kthen numrin e simboleve (gjatësinë) e stringut (objektit) për të cilin e thërrasim funksionin anëtar `gjatesia`.

Para se të përdoret funksioni anëtar, siç është `gjatesia`, ai duhet të definohet:

```
int String::gjatesia()
{
    return m_iGjatesia;
}
```

Nëse bëni definimin e funksionit brenda në klasë, atëherë operatori `::` nuk është i nevojshëm:

```
class String
{
    int    m_iGjatesia;
    char*  m_pString;
public:
    int gjatesia() { return m_iGjatesia; }
};
```

Për funksionet e shkurtëra, si funksioni `gjatesia`, mund të definohen në klasë, mirëpo për funksionet e gjata preferohet të definohen jashtë klasës duke përdorur operatorin `::` (i quajtur *scope operator* – lexo: skoup opërejter). Operatori `::` përcakton se funksioni në anën e djathtë të tij i përket klasës në anën e majtë të tij. P.sh. rreshti i kodit:

```
int String::gjatesia()
```

definon se `gjatesia` i përket klasës `String`.

Operatori `::` përdoret edhe për të bërë dallimin e funksioneve globale kundrejt atyre të klasës së definuar, kur të dy kanë emër të njëjtë. P.sh., ta

zëmë se kemi një funksion global gjatesia dhe një funksion gjatesia anëtar të klasës string. Atëherë në implementimin e funksionit anëtar të klasës string (p.sh. funksioni anëtar shkronjaEFundit) thirret funksioni anëtar i klasës string gjatesia, p.sh.:

```
char String::ShkronjaEFundit()
{
    // në këtë rast thirret funksioni anëtar
    // i klasës (gjatesia)
    return m_pString[gjatesia()-1];
}
```

Atëherë si do të mund ta thërrasim funksionin global për implementimin e funksionit anëtar të klasës në rastë kur e kemi funksionin anëtar me të njëjtin emër sikurse ai global?

Në këtë rast mund ta përdorim operatorin :: para funksionit global gjatesia, p.sh.:

```
int String::NdonjeFunksionAnetar()
{
    // në këtë rast thirret funksioni global (gjatesia)
    return ::gjatesia();
}
```

6.3.2 Konstruktoret shndërrues

Gjuha C++ (sikurse gjuha C) mundëson iniciimin e variablave gjatë definimit, p.sh.:

```
int ivjet = 26;
```

Gjuha C++ mundëson të njëjtën metodë edhe për tipet e definuara nga përdoruesit, me implementimin e konstruktoreve shndërrues dhe të operatorëve për të cilët do të flasim në këtë kapitull.

Pasi që klasa string përfaqëson vargun që në gjuhën C++ shpeshherë prezantohet në tipin const char*, atëherë do të ishte e arsyeshme ta mundësonim iniciimin e klasës string me tipin const char*. P.sh.:

```
String s = "Tungjatjeta!";
```

Kjo mundësohet në disa mënyra, njëra prej të cilave është deklarimi i konstruktorit shndërrues:

```
String(const char*);
```

Me ç'rast implementimi i këtij konstruktori do të ishte:

```
String::String(const char* pString)
{
    m_iGjatesia = strlen(pString);
    m_pString = new char [m_iGjatesia+1];
    strcpy(m_pString, pString);
}
```

Në disa raste duam që ta inicojmë stringun me një shkronjë (simbol) të vetme, p.sh.:

```
String s = 'a' ;
```

Në këtë rast nevojitet konstruktori:

```
String(const char);
```

Dhe implementimi do të dukej si më poshtë:

```
String::String(const char c)
{
    m_iGjatesia = 1;
    m_pString = new char [m_iGjatesia+1];
    m_pString[0] = c;
    m_pString[1] = 0; // përfundo stringun me zero
}
```

Në raste kur e definojmë variablën pa e inicuar me ndonjë vlerë, kompajleri thirr automatikisht konstruktorin bazë, si p.sh.:

```
String s;
ose
String* pS = new String;
```

Nëse klasa e definuar nuk ka asnjë konstruktor, atëherë kompajleri i definon automatikisht konstruktorët bazë për klasën e definuar. Mirëpo këta konstruktorë kanë efekte të padëshirueshme për klasën, sikur klasa String, që rezervojnë në mënyrë dinamike memorien për ruajtjen e të dhënave.

Në shumë raste është e mundur që në vend të konstruktorit bazë të thirret konstruktori shndërrues si p.sh.:

```
String* pS = new String("Tungjatjeta!");
```

Në raste kur në klasë kemi definuar konstruktorë me parametra dhe klasa e definuar nuk ka asnjë konstruktor bazë, atëherë kompajleri nuk krijon

konstruktor bazë. Këto klasë nuk mund të përdoren në *array* apo në klasët e tjera të librarive standarde, si p.sh. klasët *map*, *list* etj, të cilat përmbajnë vetëm objektet e klasëve me konstruktor bazë.

Nëse klasa *String* e ka të definuar vetëm konstruktorin:

```
String(const char*)
```

atëherë shtrohet pyetja si është e mundur ta deklarojmë një variabël anëtare të tipit *String* në një klasë sikurse është klasa në vijim:

```
class Data
{
public:
    String m_sData;
};
```

Në këtë rast kompajleri krijon një konstruktor bazë për klasën *Data* dhe tenton ta iniciojë variablën *m_sData* duke provuar që ta thirrë konstruktorin bazë të kësaj variabël. Mirëpo pasi për klasën *String* nuk ka konstruktor bazë, atëherë kompajleri definimin e klasës *Data* do ta paraqesë si gabim.

Në raste të këtilla, gjuha C++ mundëson iniciimin e variablave anëtare para thirrjes së konstruktorit të klasës që i përmban këto variabla. Në shembullin e mësipërm duhet deklaruar konstruktorin bazë të klasës *Data*:

```
class Data
{
public:
    Data();
public:
    String m_sData;
};
```

Dhe implementimi i konstruktorit do të ishte:

```
Data::Data()
:m_sData("")
{
}
```

Siç e vëreni rreshti i kodit `:m_sData("")` është shkruar para implementimit të konstruktorit, prandaj kompajleri e thirr këtë rresht para se ta thirrë konstruktorin e klasës.

Nëse kemi më shumë se një variabël për të inicuar, atëherë ato i ndajmë me presje, p.sh.:

```
class Data
{
public:
    Data();
public:
    String m_sData;
    String m_sMuaji;
    int    m_ivjet;
};

Data::Data()
:m_sData(""), m_sMuaji("Janar"), m_ivjet(0)
{
}
```

Siç po e vëreni, në këtë listë mund t'i inicojmë edhe variablat e tipit të kompajlerit (tipin int, char etj). Në të shumtën e rasteve ky lloj inicimi bëhet për ta bërë kodin më efikas. P.sh. konsidero nëse klasa String përmban konstruktorin bazë si dhe konstruktorin shndërrues String(const char*), atëherë kodi që vijon do të shkaktojë që së pari të thirret konstruktori bazë i klasës String, e pastaj të thirret konstruktori shndërrues String(const char*).

```
Data::Data()
{
    m_sData = "Test";
}
```

Nëse ky inicim bëhet në listë:

```
Data::Data()
: m_sData("Test") {}
```

atëherë kompajleri do ta thirrë vetëm konstruktorin shndërrues String(const char*). Ky shembull është vetëm për të demonstruar rrjedhën e kodit dhe faktin se thirrja e shpeshtë e konstruktorit bazë të klasës String nuk do të shkaktonte ndonjë pengesë në efikasitetin e programit. Efikasiteti do të vërehet në klasët ku ka kalkulime të gjata në konstruktor bazë ose ku konstruktorët inicojnë listë të gjatë të variablave anëtare të klasës.

6.3.3 Konstruktoret kopjues

Konstruktore kopjues janë konstruktorët që kanë si parametër variablat e tipit të njëjtë si të klasës që i përket. Nëse marrim parasysh definimin e klasës `String` më sipër si dhe definimin e konstruktorit shndërrues, atëherë kodi që vijon do të ketë efekt krejt tjetër nga ai i menduar:

P.sh., le të jetë funksioni `KtheNeShkronjaTeMedha` funksion anëtar i klasës `String` që shndërron shkronjat e stringut në shkronja të mëdha.

```
#include <iostream>
#include "String.h" //fajlli ku kemi definuar klasën String

using namespace std;

int main ()
{
    String sTung = "Tungjatjeta!";
    String sShkronjaTeMedha = sTung;

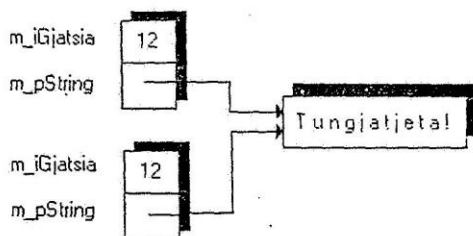
    sShkronjaTeMedha.KtheNeShkronjaTeMedha();

    return 0;
}
```

Stringu `sTung` është inicuar me një konstantë, me ç'rast është thirrë konstruktori `String(const char*)` dhe e dhëna anëtare e klasës `String` `m_pString` është inicuar me memorie të rezervuar në konstruktor, pastaj konstantja "Tungjatjeta!" është kopjuar në memorien e treguar nga treguesi `m_pString`. Në rreshtin tjetër kemi inicuar variablën `sShkronjaTeMedha` me variablën `sTung`. Pasi që variabla `sTung` është e tipit `String` dhe pasi që nuk e kemi definuar konstruktorin kopjues, atëherë kompajleri përdor konstruktorin kopjues të gjeneruar automatikisht nga kompajleri.

Klasa `String` e definuar më parë, përmban tregues të tipit `char*` që tregon në memorie të caktuar në konstruktor dhe gjatë inicimit të variablës (objektit) `sShkronjaTeMedha`, variabla anëtare `m_pString` e këtij objekti tregon në të njëjtën memorie sikurse variabla anëtare e variablës `sTung`.

Pra, në këtë rast funksioni `KtheNeShkronjaTeMedha` i shndërron shkronjat në të mëdha në memorien e treguar nga variablat anëtare `m_pString` të të dy objekteve `sShkronjaTeMedha` dhe `sTung`:



Në gjuhën C++ nëse klasët përmbajnë tregues, atëherë klasa e definuar patjetër duhet të ketë konstruktor kopjues që garanton se çdo variabël anëtare e klasës tregon në memorie të veçantë, përndryshe kodi do të kishte efekt të padëshirueshëm. Në rastin tonë, për klasën `String` do ta definonim konstruktorin kopjues kështu:

```
String::String(const String& s)
{
    m_iGjatesia = s.m_iGjatesia;
    m_pString   = new char [m_iGjatesia+1];
    strcpy(m_pString, s.m_pString);
}
```

Tani çdo objekt i tipit `String` gjatë kopjimit tregon në memorie të veçantë. Në konstruktorët kopjues parametri zakonisht është i tipit konstantë (`const`) për të lejuar edhe kopjimin e konstantave. Duhet të kenë parasysh se në konstruktorët kopjues patjetër duhet të pasohet variabla si referencë, p.sh.:

```
String(const String& s)
```

Për arsye se pasimi normal tenton kopjimin e variablës dhe pasi që kjo variabël është e tipit të klasës që definon konstruktorin kopjues, shkaktohet përsëritja e thirrjes së konstruktorit kopjues, p.sh.:

```
String::String(const String s)
{
    ...
}
```

Meqë në gjuhën C++ pasohen kopjet e parametrave, kompajleri tenton ta kopjojë variablën `s`. Kjo shkakton thirrjen e konstruktorit kopjues, i cili prapë si parametër ka variablën për kopjim. Disa kompajlerë e detektojnë këtë dhe e paraqesin si gabim. Problemi i mësipërm zgjidhet me pasimin e parametrave

si referencë, dhe siç kemi përmendur më parë, kjo nuk shkakton kopjimin e variablave, por vetëm pasimin e adresës së objektit.

Një ndër arsyt kryesore për definimin e konstruktorëve kopjues është për t'i ikur mundësisë së të treguarit të memories së përbashkët nga dy objekte të ndryshme. Me definimin e konstruktorit kopjues nuk i kemi ikur problemit plotësisht, pasi që kodi që vijon shkakton të njëjtin problem.

```
#include <iostream>
#include <String.h>

using namespace std;

void main ()
{
    String sTung = "Tungjatjeta!";           // 1
    String sShkronjaTeMedha;                 // 2

    sShkronjaTeMedha = sTung;                 // 3

    sShkronjaTeMedha.KtheNeShkronjaTeMedha(); // 4
}
```

Programi 6.3.3.1

Ky kod shkakton të njëjtin problem sikurse në konstruktorin kopjues. Në rreshtin e parë kompajleri thërret konstruktorin shndërrues:

```
String(const char*);
```

Në rreshtin e dytë kompajleri thërret konstruktorin bazë:

```
String( );
```

Kurse në rreshtin e tretë thirret operatori kopjues:

```
operator=(const String& s);
```

T'ju përkujtojmë se gjatë deklarimit të variablës, nëse ajo inicohet me ndonjë vlerë, atëherë thirret konstruktori shndërrues ose kopjues, p.sh.:

```
Data d = Data(12, 11);
```

Nëse pyetni pse në deklarimin e variablës, kur ajo inicohet, thirret konstruktori kopjues, kurse në iniciimet e tjera të variablës, përpos gjatë deklarimit, thirret operatori kopjues (=), përgjigja është kjo:

Kur një objekt (variabël) inicohet gjatë deklarimit, atëherë kompajleri thërret konstruktorin kopjues. Kjo është për arsye të optimizimit të kodit, përndryshe do të duhej të thirrrej konstruktori bazë dhe pastaj operatori kopjues:

```
Data::Data();
Data::operator=(const Data& d);
```

Në kodin:

```
Data d;
d = Data(12, "11");
```

duhet pajtë të thirret konstruktori bazë për rreshtin e parë dhe pastaj operatori kopjues. Shumë kompajlerë bëjnë optimizimin e kodit në mënyra të ndryshme dhe është e mundshme që kompajleri ta optimizojë edhe fragmentin e kodit për thirrjen e vetëm konstruktorit kopjues.

Pasi që në rreshtin e tretë të kodit në programin 6.3.3.1 thirret operatori = dhe në klasën String nuk e kemi definuar këtë operator, atëherë kompajleri e thirr operatorin bazë (të definuar automatikisht nga kompajleri) që e kopjon memorien drejtpërsëdrejti. Mirëpo, në klasën String kemi tregues që tregon në memorie të rezervuar në mënyrë dinamike dhe kopjimi i memories së një Stringu drejtpërsëdrejti në tjetrin, shkakton që të dy objektet (treguesit e objekteve String) të tregojnë në të njëjtën memorie.

Për këtë arsye, duhet definuar operatorin kopjues (=) dhe duhet rezervuar memorien për objektin në të cilin thirret ky operator.

Operatori kopjues për klasën String do të definohej kështu:

```
const String& String::operator=(const String& s)
{
    if (this == &s)        // nëse tentohet të inicohet
        return *this;      // objekti me vetveten, mos
                            // ndërmerr asgjë

    // fshihe memorien e rezervuar më parë
    delete [] m_pString;

    m_iGjatesia = s.m_iGjatesia;
    m_pString = new char [m_iGjatesia+1];
    strcpy(m_pString, s.m_pString);

    return *this;
}
```

6.3.4 Konstruktorët shndërrues dhe zëvendësimi i tipeve të ndryshme

Konstruktorët shndërrues njihen edhe si funksione shndërruese. Kjo për arsye se në vendet ku përdoren tipet për të cilat ekziston konstruktori shndërrues dhe pritet tipi i klasës që implementon konstruktorët, kompajleri automatikisht i thërret konstruktorët shndërrues dhe zëvendëson tipin e mëparshëm. Në kodin e mëparshëm:

```
String sTung;  
sTung = "Tungjatjeta!";
```

Në shikim të parë duket se kompajleri e thërret operatorin e klasës String:

```
String::operator=(const char*);
```

Mirëpo, pasi që ky operator nuk është definuar për klasën String, atëherë kompajleri e shndërron konstantën "Tungjatjeta!" në String me thirrjen e konstruktorit shndërrues:

```
String::String(const char*)
```

dhe pastaj e thërret operatorin:

```
String::operator=(const String&)
```

Këtu kemi demonstruar se si variablat e tipeve të ndryshme shndërrohen në tipe të tjera me thirrjen e konstruktorëve shndërrues. Në disa raste shndërrimi i tipeve duhet të jetë i anasjelltë, prej tipit më kompleks në ato më të thjeshta. Ta shohim fragmentin e kodit të mëposhtëm:

```
char szEmri[100];  
String s("Agimi");  
strcpy(szEmri, s);
```

T'ju përkujtojmë se funksioni `strcpy` është funksion që kopjon një *array* të shkonjave në një *array* tjetër (string të tipit të gjuhës C) dhe ky funksion ekziston në librarinë standarde. Prototipi i këtij funksioni është:

```
char *strcpy( char *strDestinimi, const char *strBurimi );
```

Në fragmentin e mësipërm ajo që tentojmë të bëjmë, është ta kopjojmë variablën e tipit String në *array* të tipit char. Kompajleri do ta paraqesë këtë

si gabim, për arsye se nuk ka njohuri si ta shndërrojë tipin `string` (parametri i dytë) në tipin e pritur `const char*`.

Nëse në rastet e këtyra duam që kompajleri ta shndërrojë klasën automatikisht në tipe të ndryshme, atëherë duhet ta definojmë operatorin e quajtur `cast`, që e shndërron klasën e definuar në tip tjetër nga ai original.

P.sh. në rastin tonë, në klasën `String` duhet ta definojmë operatorin:

```
operator const char*() const
```

Dhe implementimi i këtij operatori do të ishte si vijon:

```
String::operator const char*() const
{
    return m_pString;
}
```

Tani në kodin:

```
char szEmri[100];
String s("Agimi");
strcpy(szEmri, s);
```

`strcpy(szEmri, s)` do të ishte në rregull për arsye se kompajleri e shndërron variablën `s` të tipit `string` në `const char*` duke e thirrë operatorin `cast`.

Kompajlerët i definojnë automatikisht këta konstruktorë dhe operatorë, nëse nuk janë prezentë në definimin e klasës:

- Konstruktori bazë - Ky konstruktor përfshihet vetëm nëse nuk kemi ndonjë konstruktor tjetër të definuar në klasë. Pra ky konstruktor përfshihet kur klasa e definuar nuk ka asnjë konstruktor.
- Konstruktori kopjues - është konstruktori që merr si parametër variablën e tipit të klasës të cilës i përket konstruktori p.sh.:

```
class X
{
public:
    X(const X&);
};
```

- Operatori `=` - Ky operator definohet automatikisht nga kompajleri, duke pasur si parametër të njëjtin tip të klasës së definuar p.sh.:

```
class X
{
```

```
public:
    const X& operator=(const X&);
};
```

Duhet ta keni parasysh se konstruktorët kopjues dhe operatorët = të definuar nga kompjaleri, kopjojnë memorien drejtpërsëdrejti, dhe se efekti i këtij kopjimi shumëherë është i padëshirueshëm (siç janë problemet që i kemi për klasën String).

6.3.5 Operatori this

Deri tani kemi përdorë objekte vetëm nga klientët e objekteve (në kodin i cili i deklaronte objektet) dhe të gjitha këto objekte rezervojnë memorie të caktuar për t'i ruajtur variablat anëtare. D.m.th. secili objekt përmban memorie të veçantë për variablat anëtare. Duke marrë parasysh se klasët i përmbajnë edhe funksionet anëtare, atëherë ju mund të pyetni:

A përmbajnë memorie të veçantë për funksionet anëtare objektet e deklarua të tipit të njëjtë?

Përgjigjja është 'Jo'. Duke pasur parasysh se funksionet janë të njëjta pavarësisht nga vlerat e objekteve, atëherë nuk do të ishte efikase të kishim memorie të veçantë për funksionet anëtare të secilit objekt. Pra, të gjitha objektet e tipit të njëjtë i shfrytëzojnë të njëjtat funksione (në memorie).

Atëherë ju do të pyetni:

Si është e mundur që funksionet anëtare të objekteve të identifikojnë objektin dhe me të dhënat e cilit objekt manipulojnë?

Për ta kuptuar këtë, le ta marrim si shembull klasën Data:

```
class Data
{
public:
    Data(int iDita, int iMuaqi, int iViti);
    void ShtypeDaten();
private:
    int m_iDita;
    int m_iMuaqi;
    int m_iViti;
};
```

Kodi në vazhdim deklaron dy objekte të ndryshme (të tipit të njëjtë) :

```
Data x(2, 7, 2000);
Data y(28, 11, 2000);
```

Nëse e thirrjmë funksionin anëtar ShtypeDaten:

```
x.ShtypeDaten();
```

atëherë kompajleri e shndërron këtë kod në:

```
ShtypeDaten(this);
```

ku this është adresa e objektit x. Në këtë mënyrë i njëjti funksion ShtypeDaten përdoret për të gjitha objektet e tipit Data dhe pasi që funksioni merr si parametër adresën e objektit, atëherë mund të identifikojë me cilin objekt ka punë.

Pra të gjitha funksionet anëtare të objekteve (përpos funksioneve anëtare statike) marrin një parametër të fshehtë nga shfrytëzuesi, e ky është treguesi this - adresa e vetë objektit.

Me përdorimin e treguesit this brenda funksioneve anëtare të klasës përdorim vetë adresën e objektit, p.sh.

```
const String& String::operator=(const String& s)
{
    if (this == &s)                // nëse tentojmë të inicojmë
        return *this;              // objektin me vetveten, mos
                                    // ndërmerr asgjë
}
```

Nëse kemi implementuar funksionin anëtar ShtypeDaten të klasës së dhëna kështu:

```
void Data::ShtypeDaten()
{
    cout << m_iDita << " / "
          << m_iMuaji << " / "
          << m_iViti;
}
```

atëherë, mbasi treguesi this brenda funksionit anëtar të objektit është vetë adresa e objektit, implementimi i funksionit ShtypeDaten (në vijim) është plotësisht i njëjtë me implementimin e të njëjtit funksion më sipër.

```
void Data::ShtypeDatën()  
{  
    cout << this->m_iDita << " / "  
        << this->m_iMuaji << " / "  
        << this->m_iViti;  
}
```

6.4 Klientët

Pas definimit të klasës (p.sh. koha), funksionet mund të krijojnë objekte të reja të tipit koha, dhe mund t'u dërgojnë porosi objekteve nëpërmjet funksioneve anëtare të klasës.

Deklarimi i variablave të tipit të definuar nga ju (përdoruesi) bëhet njësoj sikur deklarimi i variablave të kompajlerit, përveçse gjatë deklarimit të variablave të definuara nga përdoruesi duhet të përfshihen edhe argumentet e konstruktorit të tipit (kuptohet, nëse tipi i definuar ka konstruktor me parametra).

Le të jetë koha klasë e definuar si më parë, atëherë variabla e re mund të deklarohet si më poshtë:

```
koha agimi(6, 15);
```

Ky deklarim krijon objekt të ri të tipit koha me vlerë 6:15 am.

Konstruktori mund të përdoret edhe në mënyrë tjetër, të krijojë objekt pa deklarimin e variablës p.sh.:

```
koha (8,21)
```

Kodi i mësipërm krijon një objekt të përkohshëm të klasës koha që përfaqëson kohën 8:21am. Objektet e përkohshme mund të përdoren edhe kështu:

```
agimi = koha(5,41);
```

Në këtë mënyrë krijojmë një objekt të ri, i cili vlerën e mëparshme të variablës agimi e ndërron me vlerën e re 5:41 am.

Klasët me më shumë se një konstruktor, të dalluar nga numri i argumenteve dhe tipi i tyre, janë më të përdorshme, meqë përmes tyre mund të krijojmë objekte të kësaj klase nga tipe të ndryshme.

6.5 Funksionet ngarkuese

Në gjuhën C++ mund të përdoren disa funksione me emra të njëjtë, përderisa numri i argumenteve është i ndryshëm apo tipi i parametrave është i ndryshëm. Kjo është përparësi, posaçërisht për funksionet që kryejnë punë të njëjta me tipe të ndryshme. Gjatë përdorimit të funksioneve me emra të njëjta (por me tip dhe numër të ndryshëm të argumenteve) gjuha C++ e identifikon

se cilin funksion duhet ta thërrasë në bazë të emërimit të brendshëm të funksionit të përdorur nga kompajleri.

Që një shembull i thjeshtë për shtypjen e numrit të tipit të ndryshëm si dhe shtypjen e kohës:

```
void shtype ( int i )      { cout << i ; }
void shtype ( float i )   { cout << i ; }
void shtype ( koha k )    { k.shtypeKohen ( ) ; }
```

thirrjet e funksionit si `shtype(5)`, `shtype(agimi)` (agimi është variabël e tipit koha), `shtype(7.8)` bëjnë që C++ ta thirrë funksionin përkatës për çdo tip të argumentit. Funksionet ngarkuese bëjnë të mundshme eliminimin e makrove të përdorura në gjuhën C për të dhëna të tipeve të ndryshme, p.sh. makroja për gjetjen e numrit maksimal e definuar si:

```
#define max((a),(b))      (a) > (b) ? (a) : (b)
```

mund të marrë për parametër çdo tip që e njeh operatorin `>`. Makrot nuk janë tipe kontrolluese dhe, siç kemi përmendur më parë, këto zëvendësohen me vlerën e tyre gjatë procesit të priprocesorit. Në gjuhën C++ makroja `max` mund të zëvendësohet me anë të funksioneve ngarkuese, p.sh.:

```
int max(int a, int b)
{
    return a > b ? a : b;
}
```

```
float max(float a, float b)
{
    return a > b ? a : b;
}
```

Këtu kemi definuar funksionet ngarkuese vetëm për dy tipe: `int` dhe `float`. Nëse dëshirojmë që këto funksione të pranojnë edhe tipe të tjera, atëherë do të duhej një funksion për çdo tip. Kjo do të ishte më pak e dëshirueshme kur kemi parasysh se në gjuhën C, një makro e vetme vrente për të gjitha tipet që njohin operatorin `>`. Po e përsërisim prapë se makrot nuk janë tipe kontrolluese dhe se shpesh bëhen shkaktare të gabimeve po nuk u përdorën me kujdes. Në gjuhën C++ kur kemi të bëjmë me ndonjë funksion, sikurse funksioni i mësipërm `max` ku kodi i funksionit është i njëjtë për çdo tip, atëherë mund të përdorim shabllonet të cilat mundësojnë vetëm një funksion për të gjitha tipet që njohin operatorin `>`, si p.sh.:

```
template < class T >
```



```
T max(T a, T b)
{
    return a > b ? a : b;
}
```

Kjo definon funksionin max për çdo tip që përmban operatorin >. Kompajleri gjatë kompajlimit të kodit do ta gjenerojë kodin për çdo tip të përdorur në program me këtë funksion.

6.6 Vlerat e variablave ngarkuese (vlerat bazë)

Sic e përmendëm më sipër, gjuha C++ në krahasim me gjuhën C mundëson krijimin e disa funksioneve me të njëjtin emër, përderisa numri dhe tipi i parametrave ndryshon për çdo funksion. Përpos kësaj, gjuha C++ mundëson edhe parametrat e quajtur *parametrat ngarkues*, të cilët mundësojnë që programi të shkurtohet duke e zëvendësuar një apo më shumë funksione ngarkuese.

Parametrat ngarkues definojnë në prototipin e funksionit dhe nuk përsëriten në implemetimin e këtij funksioni. P.sh. le të kemi një funksion që kontrollon vlerat e gabimeve në faqe të caktuara dhe e kthen shumën e gabimeve në këto faqe me gabimet e mëparshme.

```
int KontrollloGabimet(int faqja, gabimetEmeParshme = 0);
```

Nëse fillojmë t'i kontrollojmë gabimet prej faqes së parë, atëherë gabimet e mëparshme janë të barabarta me zero dhe parametri i dytë *gabimetEmeParshme* nuk ka nevojë të pasohet në funksion:

```
int gabimet;
```

```
int faqja = 1;
```

```
gabimet = KontrollloGabimet(faqja);
```

```
faqja++;
```

```
// pasoji këtu gabimet e gjetura në faqen e parë
gabimet = KontrollloGabimet(faqja, gabimet);
```

Duhet ta keni parasysh se funksionet e shkruara me parametra ngarkues mundësojnë që kodi të jetë më i shkurtër dhe se parametrat ngarkues nuk kanë nevojë të pasohen në këtë funksion gjatë thirrjes së funksionit. Kurse, parametrat që nuk janë parametra ngarkues duhen pajtë të pasohen në funksion gjatë thirrjes së funksionit. Nëse parametrat ngarkues nuk pasohen në funksion, atëherë kompajleri i zëvendëson me vlerat e inicuar në prototipin e funksionit.

Renditja e parametrave ngarkues mund të bëhet vetëm duke filluar prej anës së djathtë të listës së parametrave në funksion e duke vazhduar nga ana e majtë. Është e mundshme që të gjithë parametrat e funksionit të jenë parametra ngarkues, mirëpo nuk është e mundshme që disa parametra ngarkues të jenë në anën e majtë të listës së parametrave dhe disa parametra jongarkues të jenë në anën e djathtë. P.sh. prototipi i funksionit të mëposhtëm

nuk është në rregull për arsye se parametri ngarkues gabimetEmeParshme gjendet në anën e majtë të listës kurse në anën e djathtë kemi parametrin jongarkues.

```
// ky prototip nuk është i vlefshëm
int KontrollloGabimet(int gabimetEmeParshme = 0, int faqja);
```

Siç përmendëm më sipër, është plotësisht e vlefshme që të gjithë parametrat në listë të jenë parametra ngarkues, p.sh.:

```
int KontrollloGabimet(           int faqja = 1,
                                int gabimetEmeParshme = 0);
```

Kjo është e vlefshme për arsye se nuk kemi ndonjë parametër në listë që nuk është parametër ngarkues e që në anën e majtë të listës ka ndonjë parametër ngarkues. D.m.th. parametrat ngarkues gjithnjë duhet të fillojnë nga ana e djathtë.

Në gjuhën C++ parametrat ngarkues mund të zëvendësohen me funksionet ngarkuese, mirëpo për të arritur përmes tyre të njëjtin synim sikurse me parametrat ngarkues, do të duhej ta definonim më shumë se një funksion ngarkues me numër të ndryshëm të parametrave. P.sh. funksioni KontrollloGabimet me parametër ngarkues i definuar më parë do të duhej të definohej kështu pa parametra ngarkues:

```
int KontrollloGabimet(int faqja);
int KontrollloGabimet(int faqja, int gabimetEmeParshme);
```

dhe implementimi i funksionit të parë do të ishte:

```
int KontrollloGabimet(int faqja)
{
    KontrollloGabimet(faqja, 0);
}
```

kurse funksioni me të dy parametrat do të kishte kodin specifik për kontrollimin e gabimeve:

```
int KontrollloGabimet(int faqja, int gabimetEmeParshme)
{
    // Këtu funksioni do t'i kontrollonte gabimet
    // ...
}
```

Siç e vëreni, parametrat ngarkues mundësojnë që kodi të jetë më i shkurtër sesa po të përdoreheshin funksionet ngarkuese.

6.7 Variablat dhe funksionet statike

Në gjuhën C dhe në disa gjuhë të tjera procedurore, shkëmbimi i të dhënave ndërmjet disa pjesëve të kodit bëhet me anë të parametrave të funksioneve. Mirëpo, në programe të gjata, kur kemi të bëjmë me shkëmbimin e disa të dhënave në tërë programin, nuk do të ishte praktike që këto variabla të shkëmbeheshin me anë të parametrave të funksioneve. Këto të dhëna zakonisht shkëmbehen me anë të variablave globale, të cilat kanë kohëzgjatje veprimi gjatë tërë ekzekutimit të programit.

Variablat globale shpesh e lehtësojnë programimin, duke mundësuar që kodi të jetë më i shkurtër sesa shkëmbimi i të dhënave me anë të parametrave të funksioneve. Mirëpo, përdorimi i variablave globale pa kujdes shkakton efekte të padëshirueshme dhe gjetja e gabimeve në kodin ku janë përdorë variablat globale është shumë e vështirë. Pasi që gjuha C++ deri diku mundëson shmangien e përdorimit të variablave globale, atëherë rekomandohet që këto variabla të përdoren me kujdes dhe mundësisht t'i shmangeni sa më shumë përdorimit të tyre.

Gjuha C++ mundëson deklarimin e variablave statike që në një mënyrë zëvendësojnë variablat globale. Njëherësh, kontrollimi i këtyre variablave është më i lehtë sesa kontrollimi i variablave globale.

Në klasë, siç kemi përmendur më parë, mund të deklaroni variabla anëtare që i përkasin klasës, dhe çdo instancë e klasës (objekt i tipit të klasës) ka memorie të veçantë të rezervuar për variablat anëtare. Variablat anëtare statike të klasës bëjnë përjashtim, sepse ekziston vetëm një kopje e variablës statike për të gjithë instancat e klasës që definojnë variablën anëtare statike. Në këtë mënyrë, objektet e tipit të klasës që përmban variablën statike, shkëmbejnë të dhënat nëpërmjet këtyre variablave, me të gjitha objektet e të njëjtit tip. Pra, në një mënyrë variablat statike zëvendësojnë variablat globale. Ndryshimi i vlerave të variablave statike është më i kontrolluar dhe nëse variablat statike definohen si private, atëherë vlerat e këtyre variablave mund të ndryshohen vetëm nëpërmjet klasës që i përmban këto variabla.

Ta shohim klasën e definuar në kodin që vijon:

```
class Data
{
public:
    void NderroVleren(int iVlera);
private:
    static int m_iNumri;
};
```

// Inicimi i variablave statike bëhet si në kodin që vijon.

```
// Ky rresht zakonisht shkruhet në fajlin ku implementohen
// funksionet anëtare të klasës (p.sh. fajlli data.cpp).
```

```
int Data::m_iNumri = 0;
```

```
// implementimi i funksionit anëtar i cili ndërron
// vlerën e variablës statike
void Data::NderroVleren(int iVlera)
{
    m_iNumri = iVlera;
}
```

Siç mund ta vëreni, vlerën e variablës anëtare të klasës Data mund ta ndërronim vetëm me anë të funksionit anëtar NderroVleren. Vlera e variablës m_iNumri do të jetë e dukshme për të gjitha objektet e tipit Data në tërë programin dhe pasi që ndërrimi i vlerës së kësaj variable bëhet vetëm me anë të funksionit NderroVleren, atëherë kontrollimi i kësaj variable është shumë më i lehtë sesa i variablave globale.

Instanca e variablave statike ekziston edhe pa ekzistimin e ndonjë instance të klasës që përmban variablën statike, d.m.th. ekzistenca e variablës statike është e pavarur nga instanca e klasës. Mirëpo nëse variabla anëtare statike është private apo protected, atëherë vlera e kësaj variable mund të ndryshohet vetëm nëpërmjet klasës që e përmban variablën.

Përpos variablave statike, klasët mund të kenë edhe funksione statike. Sikurse variablat statike, edhe funksionet statike janë të pavarura nga instanca e klasës dhe mund të thirren duke përdorë operatorin për skop ::. Në funksionet statike mund të përdorim vetëm variablat anëtare të klasës që janë statike. Kjo duke qenë se funksionet anëtare statike mund të thirren pa ekzistimin e instancës së klasës, prandaj do të ishte e pamundshme të përdornim variablat që ekzistojnë vetëm në instanca të klasës. Nëse tentoni të përdorni variablat anëtare normale në funksione statike, atëherë këtë kompajleri do ta paraqesë si gabim. P.sh. vëreni kodin e mëposhtëm:

```
class Data
{
public:
    void NderroVleren(int iVlera);
    static FunksioniStatik();           // funksioni statik
private:
    static int    m_iVariablaStatike;
    int          m_iVariablaJoStatike;
};
```

```
// implementimi i funksionit statik është i njëjtë me atë
```

```
// të funksioneve normale
void Data::FunksioniStatik()
{
    m_iVariablaStatike = 10; // në rregull

    // në funksione statike nuk mund të përdorim variablat
    // anëtare jostatike
    m_iVariablaJoStatike = 10; // gabim
}
// funksioni statik mund të thirret kështu
Data::FunksioniStatik();
```

6.8 Krijimi i tipeve abstrakte dhe fshehja e të dhënave

Qëllimi i tipeve të të dhënave abstrakte është fshehja e të dhënave dhe kompleksitetin në manipulimin me këto tipe. Në gjuhën C++ përdoren tri shprehje brenda klasëve për definimin e nivelit të përdorimit të funksioneve dhe variablave anëtare të klasëve. Këto shprehje janë:

- **public** - që lejon përdorimin e funksioneve dhe variablave anëtare edhe nga klientët e klasës.
- **private** - definon se funksionet dhe variablat e klasës nuk mund të përdoren nga klientët e klasës. Këto funksione dhe variabla mund të përdoren vetëm brenda në klasë.
- **protected** - sikurse **private**, mirëpo lejon përdorimin e të dhënave dhe funksioneve brenda klasëve që kanë prejardhje nga klasa që përmban të dhënat apo funksionet anëtare (për këtë do të flasim më vonë).

Nëse të dhënat anëtare në klasë i deklarojmë si **public**, p.sh..

```
class String
{
public:
    // konstruktori
    String(char* pString);
public:
    // të dhënat anëtare të klasës
    int    m_iGjatesia;
    char*  m_pString;
};
```

atëherë këto të dhëna mund të përdoren drejtpërdrejt:

```
String s("Tungjatjeta!");

cout << "Gjatësia e stringut \"Tungjatjeta!\" është: "
      << s.m_iGjatesia;
```

Mirëpo mënyrës së këtillë të kodimit duhet t'i ikni sepse është në kundërshtim me metodën e krijimit të tipeve të të dhënave abstrakte. Me lejin e përdorimit të të dhënave anëtare të klasës zbulojmë jo vetëm përbërjen e klasës, por edhe se ndërrimi i vlerave të këtyre të dhënave nuk mund të kontrollohet nga klasa dhe se ky ndërrim i të dhënave anëtare mund të ketë efekt të padëshirueshëm.

Nëse e iniciojmë një objekt të tipit String të deklaruar më sipër si:

```
String s("Tungjatjeta!");
```

dhe pasi që kemi akses (qasje, hyrje) në të dhënat anëtare, atëherë mund ta ndërrojmë variablën anëtare `m_iGjatesia`, p.sh.:

```
s.m_iGjatesia = 5; // gabim
```

Siç e shihni, këtu, pasi që i kemi deklaruar të dhënat anëtare si publike, përdoruesi i klasës String mund ta ndërrojë vlerën e gjatësisë së stringut, i cili përmban vargun e simboleve "Tungjatjeta!" që është i gjatë 12 shkronja. Pra objekti `s` përmban të dhëna që nuk janë të sakta dhe në përputhje me definimin e objektit.

Për këtë arsye preferohet që variablat anëtare të definohen si private dhe përdorimi i tyre të kontrollohet nëpërmjet vetë klasës me anë të funksioneve anëtare. Në këtë mënyrë mund të kontrollohet objekti dhe të përmbajë të dhëna të sakta gjatë tërë kohës së ekzekutimit të programit, apo jetëzgjatjes së objektit.

Udhëzimet private, protected dhe public i grupojnë variablat dhe definojnë lejen e aksesit të tyre dhe të funksioneve anëtare deri në shprehjen tjetër apo deri në fund të klasës, nëse ky udhëzim është i fundit në definimin e klasës, p.sh.:

```
class Personi
{
public:
    string Emri();
    int Vjet();
private:
    string m_sEmri;
    int m_iVjet;
```

```
};
```

Funksionet anëtare:

```
    string Emri();  
    int    Vjet();
```

janë publike dhe mund të përdoren nga klientët e klasës Personi, kurse variablat anëtare:

```
    string m_sEmri;  
    int    m_iVjet;
```

janë private dhe mund të përdoren vetëm brenda në klasë (funksionet anëtare të klasës).

Ushtrime

1. Definoni një klasë që përmban dimensionet e një dhonje si dhe numrin e dyerve dhe të dritareve.
2. Definoni një klasë që paraqet një shtëpi dhe përmban disa klasë që përfaqësojnë dhomat (pra klasa e definuar në ushtrimin 1). Klasa që përfaqëson shtëpinë duhet t'i ketë funksionet anëtare për t'i llogaritur dimensionet e tërë shtëpisë. Natyrisht se këto dimensionet duhen të gjenden duke llogaritur dimensionet e dhomave. Pra klasa që përfaqëson shtëpinë nuk përmban ndonjë variabël anëtare për për t'i ruajtur dimensionet.
3. Definoni funksionin anëtar të klasës që përfaqëson shtëpinë, e që kthen numrin e gjithsejtë të dyerve dhe dritareve në tërë shtëpinë.
4. Definoni operatorin = të klasës që përfaqëson shtëpinë. Operatori = duhet të pranojë si parametër tipin konstant të të njëjtës klasë, p.sh.:

```
Shtepia operator=(const Shtepias sh):
```

Mos harroni ta përdorni operatorin this në implementimin e operatorit =.

5. Definoni dy funksione ngarkuese për tipet int dhe float për kthimin e maksimumit të tre numrave të ndryshëm.
6. Definoni dy funksione ngarkuese (minimumi) të cilat e kthejnë numrin më të vogël prej dy parametrave të pasuar në funksion (përdorni tipin int dhe float për dy funksionet e ndryshme).
7. Definoni funksionin shtypeFjaline i cili pranon parametrin ngarkues të tipit char*. Parametri ngarkues duhet ta ketë vlerën fillestare NULL, pra:

```
shtypeFjaline(char* pFjalia = NULL);
```

Funksioni shtypeFjaline shtyp fjalinë e pasuar nëpërmjet parametrin pFjalia, mirëpo nëse parametri pFjalia është i barabartë me NULL atëherë ky funksion duhet ta shtypë fjalinë "<NULL>".

8. Tentoni të definoni një funksion që përdor shabllonet për kthimin e minimumit të dy numrave. Shih funksionin max.
9. Tentoni të definoni një funksion që përdor shabllonet për kthimin e maksimumit të tre numrave.

Përmbledhje

Gjuha C++ ofron shprehjen `class` me të cilin përdoruesit mund të krijojnë tipe të reja. Këto tipe të reja mund t'i simulojnë tipet e furnizuara nga vetë gjuha C++. P.sh. përdoruesit mund t'i implementojnë operatorët si `+`, `-`, `*` etj. për tipet e reja.

Në shprehjen `class` mund të përdoren tri shprehjet `public`, `protected` dhe `private`, të cilat shprehje definojnë nivelin e aksesit të funksioneve dhe të të dhënave anëtare të tipit të ri. Këto shprehje mundësojnë fshehjen e të dhënave anëtare, pra krijimin e të dhënave abstrakte.

Gjatë definimit të klasëve, gjuha C++ ofron konstruktor dhe operator bazë nëse këta nuk implementohen në tipin e ri. Mirëpo duhet të keni kujdes me klasët, të cilat e rezervojnë memorien në mënyrë dinamike për të dhëna anëtare. Në këto raste duhet patjetër të implementoni konstruktorin kopjues dhe operatorin `=`.

Organizimi i kodit në C++

Në gjuhën C++ organizimi i kodit është pothuajse i ngjashëm me atë të gjuhës C. Në gjuhën C++ fajlli header përdoret për definimin e protipit të funksioneve, si dhe për definimin e klasëve, kurse fajllat e tjerë (zakonisht fajllat e emërtuara me prapashtesën `cpp` ose `cc`) përdoren për implementimin e funksioneve globale, si dhe të funksioneve anëtare të klasëve. Në fajllat ku përdoren funksionet apo klasët e definuara më parë në fajlla të tjera, duhet të përfshihet vetëm fajlli header në të cilën është definuar prototipi i funksionit (funksioneve) apo i klasës (klasëve).

P.sh.. nëse në fajllin `data.h` kemi definuar prototipin e funksionit dhe klasën që vijon:

```
int max(int a, int b);

class Data
{
public:
    int KtheMaksimumin(int iViti);
    void RuajeVitin(int iViti) { m_iViti = iViti; }
public:
    int m_iViti;
};
```

atëherë kudo që nevojitet të përdoret funksioni `max` apo klasa `Data`, duhet ta përfshini fajllin header `data.h`. P.sh.:

```
#include <iostream>
#include "data.h" // përfshije fajllin ku është definuar
                 // funksioni max dhe klasa Data

void main ()
{
    Data d;
    d.RuajeVitin(1999);
}
```

Siç përmendëm më parë, implementimi i funksioneve të klasëve bëhet në fajllin me prapashtesë `cpp` apo `cc`. P.sh. në fajllin `data.cpp` definojmë funksionin global `max` dhe funksionin e klasës `Data`:

```
#include "data.h"

int max(int a, int b)
{
    return a > b ? a : b;
}

int Data::KtheMaximumin(int iviti)
{
    return max(m_iviti, iviti);
}
```

Në fajllin `data.cpp` kemi përfshirë dosjen header ku është definuar prototipi i funksionit `max` dhe klasa `Data`. Funksioni `max` është funksion i pavarur (nuk i përket asnjë klase) dhe shkruhet sikur në gjuhën C, kurse funksioni `KtheMaximumin` i përket klasës `Data` dhe në implementimi i këtij funksioni duhet të paraprihet me emrin e klasës `Data` dhe operatorin për skope `::`.

Në të shumtën e rasteve, funksionet e shkurtëra definohen në fajllin header si p.sh. funksioni i klasës `Data`, `Ruajevitin`. Për funksionet e gjata nuk preferohet implementimi në fajllin header.

Në të shumtën e rasteve, klasët e implementuara më parë, që përdoren në shumë projekte të ndryshme, kompajlohen dhe bashkëngjiten në librari, kështu që në projektet e ardhshme nevojiten vetëm fajllat header në të cilat janë definuar klasët. Në përfitimin e programit ekzekutues, kompajleri e përdor librarinë e krijuar më parë, për ta lidhur implementimin e klasëve të përdorura.

Libraritë standarde, siç kemi përmendur më parë, përmbajnë klasën `String`, mirëpo këtu do ta implementojmë një version të thjeshtë të kësaj klase për të demonstruar se si organizohet kodi në fajlla të ndryshme.

Do të fillojmë me definimin e klasës `String` që do ta ruajmë në fajllin `String.h`.

```
Fajlli:    String.h

// komandat e priprocesorit për t'i ikur përfshirjes
// së shumëfishtë të këtij fajlli
#ifndef String_h
#define String_h
```

```

#include <iostream>

using namespace std;

class String
{
public:
    // konstruktorët
    String();
    String(const String&);
    String(const char* pString);
    String(const char);

    // destruktori
    ~String();

public:
    int Gjatesia() { return m_iGjatesia; }

    String& operator=(const String&);
    bool operator==(const String&) const;
    bool operator!=(const String&) const;
    String operator+(const String&) const;
    char operator[](int index) const;
    operator const char*() const;

private:
    // të dhënat anëtare të klasës
    int m_iGjatesia;
    char* m_pString;

    // funksionet për shtypjen dhe leximin e stringut
    friend ostream& operator << (ostream& os,
                                const String& s);
    friend istream& operator >> (istream& is, String& s);
};

#endif

```

Në fajllin String.cpp kemi implementuar konstruktorët, funksionet dhe operatorët anëtarë të klasës String.

Fajlli String.cpp:

```

#include <string.h> // libraria standarde për manipulim
                  // me array të tipit char

#include "String.h" // fajlli ku kemi definuar klasën String

String::String()
{
    m_iGjatesia = 0;
    m_pString = new char [1];
}

```

```

        m_pString[0] = '\\0';
    }

String::String(const String& s)
{
    m_iGjatesia = s.m_iGjatesia;
    m_pString = new char [m_iGjatesia+1];
    strcpy(m_pString, s.m_pString);
}

String::String(const char* pString)
{
    m_iGjatesia = strlen(pString);
    m_pString = new char [m_iGjatesia+1];
    strcpy(m_pString, pString);
}

String::String(const char c)
{
    m_iGjatesia = 1;
    m_pString = new char [2];
    m_pString[0] = c;
    m_pString[1] = '\\0';
}

// destruktori
String::~String()
{
    delete m_pString;
}

String& String::operator=(const String& s)
{
    if (this == &s) // nëse tentojmë të inicojmë
        return *this; // objektin me vetveten mos
                    // ndërmerr asgjë

    // fshije memorien e rezervuar më parë
    delete [] m_pString;

    m_iGjatesia = s.m_iGjatesia;
    m_pString = new char [m_iGjatesia+1];
    strcpy(m_pString, s.m_pString);

    return *this;
}

bool String::operator==(const String& s) const
{
    return strcmp(m_pString, s.m_pString) == 0;
}

```

```

bool String::operator!=(const String& s) const
{
    return strcmp(m_pString, s.m_pString) != 0;
}

String String::operator+(const String& s) const
{
    String tmp;
    tmp.iGjatesia = m_iGjatesia + s.m_iGjatesia;

    delete tmp.m_pString;
    tmp.m_pString = new char [tmp.m_iGjatesia+1];
    strcpy(tmp.m_pString, m_pString);
    strcat(tmp.m_pString, s.m_pString);

    return tmp;
}

char String::operator[](int indexi) const
{
    return m_pString[indexi];
}

String::operator const char*() const
{
    return m_pString;
}

// funksionet për shtypjen dhe leximin e stringut
ostream& operator << (ostream &os, const String& s)
{
    return os << (const char*)s;
}

istream& operator>>(istream& is, String& s)
{
    char tmp[256];
    is >> tmp;
    s = String(tmp);
    return is;
}

```

Në fajllin `main.cpp` kemi implementuar funksionin kryesor të programit, pra funksionin `main`. Këtu kemi përdorë klasën `String` të implementuar më sipër, për të demonstruar se si implementimi i një klase bën që kodi të jetë më elegant, me më pak gabime dhe mund të përdoret prapë në shumë raste të tjera.

Fajlli `main.cpp`:

```

#include <iostream>
#include "String.h"

using namespace std;

int main (int argc, char* argv[])
{
    String s;
    s = "Tungjatjeta ";
    cout << s << endl;

    String sKosova("KOSOVA e lirë!");
    cout << sKosova << endl;

    String sFjalia = s + sKosova;
    cout << sFjalia << endl;

    String s1("Shqipëria");
    String s2("Kosova");

    if (s1 != s2)
    {
        cout << "Duhet patjetër të punojmë t'i bëjmë një."
              << endl;
    }

    return 0;
}

```

Në kodin e mësipërm kemi demonstruar shumicën e funksioneve të përmendura në kapitullin e kaluar. Vlen të përmendim se operatorët << dhe >> nuk mund të jenë anëtarë të ndonjë klase, pra këta operatorë duhen të jenë globalë. Mirëpo për t'i lejuar këta operatorë që të përdorin edhe të dhëna private të klasës, atëherë ata duhen të definohen si *friend* (shok në anglisht). Udhëzimi *friend* lejon që një klasë t'i përdorë edhe variablat anëtare private dhe *protected* të klasës tjetër. P.sh. nëse e kemi klasën *klasaA* që përmban një variabël anëtare private dhe kjo klasë i beson klasës *klasaB* dhe dëshiron që kjo klasë të jetë në gjendje t'i përdorë edhe të dhënat private të klasës *klasaA*, atëherë këto klasa (*klasaA* dhe *klasaB*) do të definoheshin kështu:

```

class klasaA
{
    friend klasaB;    // defino klasën klasaB si shok
                    // dhe lejo që klasaB të përdorë
public:              // edhe të dhëna private të klasës klasaA
    klasaA();
private:

```



```

        int m_iNumriA;
    };

    class klasaB
    {
    public:
        klasaB();
    private:
        klasaA m_aData;
        int m_iNumriB;
    };

    klasaB::klasaB()
    {
        // edhe pse m_iNumri është variabël private
        // përdorimi i kësaj variable në rreshtin që
        // vijon është plotësisht i mundur për arsye
        // se klasa klasaB është shok i klasës klasaA
        m_aData.m_iNumriA = 10;
    }

```

Nëse klasa klasaA do të kishte një variabël anëtare të klasës klasaB, atëherë kjo klasë nuk do të mund ta përdorte variabëlën private m_iNumriB, anëtare të klasës klasaB. Kjo është për arsye se klasaB nuk e ka definuar klasën klasaA si shok (friend). Pra, udhëzimi friend është vetëm njëkahësh dhe se nëse të dy klasët dëshirojnë të lejojnë përdorimin e variablave private, atëherë që të dyja klasët duhen të definojnë njëra-tjetrën si friend (shoqe).

Në programin e mëparshëm kemi definuar funksionet apo operatorët << dhe >> si friend. Kjo është plotësisht e mundshme dhe kjo teknikë shpesh përdoret në gjuhën C++ për lejimin e përdorimit të variablave private anëtare për disa operatorë.

Deklarimi i klasës ose i funksioneve si friend është në kundërshtim me krijimin e të dhënave abstrakte. Siç kemi përmendur më parë, qëllimi i të dhënave abstrakte është fshehja e kompleksit dhe mënyra e përbërjes së tipeve të reja. Kurse udhëzimi friend bën të kundërtën, pra lejon që një klasë të përdorë të dhëna private drejtpërsëdrejti dhe efekti të jetë shpesh i padëshirueshëm. Mirëpo, udhëzimi friend lejon përdorimin e të dhënave private vetëm për klasët që kanë "besim" se nuk do të keqpërdoren nga klasët që përdorin këtë shprehje. Asgjëmandgut, udhëzimi friend rekomandohet të përdoret me kujdes dhe sa më pak.

Ushtrime

1. Organizoni në metodën e shpjeguar në këtë kapitull, klasët e definuara në ushtrimet e kapitullit të kaluar (klasët që përfaqëdojnë shtëpinë, dhomat etj).

Trashëgimi dhe hierarkia në klasë

"Bashkimi bën fuqinë"

Kudo në jetën e përditshme, përparimi bëhet me përdorimin e veglave të krijuara më parë dhe shfrytëzimin e zbulimeve të mëparshme. Një fjalë e vjetër popullore angleze thotë: *"Pse ta zbulojmë rrotën prej fillimit?"*. Edhe në programin vlen e njëjta filozofi, pra përdorimi i kodit të shkruar më parë, siç ndodh me libraritë standarde të ofruara nga kompajleri si dhe libraritë e krijuara nga vetë përdoruesit për disa funksione specifike. Gjuha C mundëson përdorimin e kodit me përmbledhjen e funksioneve të përdorshme dhe krijimin e librarisë. Kurse, gjuha C++ lehtëson punën edhe më tej, duke mundësuar përdorimin e kodit të shkruar më parë me anë të klasëve. Këto klasë, në krahasim me libraritë e shkruara në gjuhën C, shpeshherë njihen edhe si përmbledhje logjike e funksioneve. Funksionet e shkruara më parë dhe të ruajtura në librari, mund të përdoren vetëm ashtu si janë dhe nuk mund të ndërrohen për t'iu përshtatur problemit specifik. Nëse puna e këtyre funksioneve nuk është ekzakte për problemin e ri, atëherë duhet shkruar prej fillimit të njëjtat funksione për t'iu përshtatur problemit të ri. Me fjalë të tjera, duhet zbuluar rrotën prej fillimit. Gjuha C++ mundëson që ta ndryshojmë "rrotën" ekzistuese për t'ia përshtatur problemit të ri.

Shpeshherë, as klasët e shkruara më parë nuk janë të përshtatshme për përdorim në çdo problem. Këto klasë mund të jenë të përgjithshme, pra jo të përshtatshme për ndonjë problem specifik. Në anën tjetër, klasët specifike që janë të përshtatshme për zgjidhjen e ndonjë problemi, nuk janë të përshtatshme për zgjidhjen e ndonjë problemi tjetër. Gjuha C++ mundëson trashëgiminë e klasëve, d.m.th. trashëgiminë e kodit të klasëve të shkruara më parë dhe shtimin dhe ndryshimin vetëm të pjesës së kodit që nuk i përshtatet problemit të ri. Nëse një klasë përmban funksione të përgjithshme që janë të përshtatshme për probleme të përgjithshme, mirëpo nuk janë të përshtatshme për probleme specifike, atëherë mund ta krijojmë një klasë tjetër që i trashëgon funksionet e klasës së përgjithshme (pa pasur nevojë ta shkruajmë kodin e mëparshëm). Në gjuhën C++ mund ta rrisim funksionin e klasës duke shtuar funksione dhe variabla anëtare në klasë.

Siç kemi përmendur më parë, klasët i paraqesin objektet e jetës së përditshme dhe funksionet e këtyre objekteve. Objektet shpesh janë të lidhura mes vete dhe klasët që i paraqesin këto objekte krijojnë hierarki duke trashëguar vetitë e njëjta të objekteve nga njëri në tjetrin. P.sh. të analizojmë një grup punëtorësh në një punëtori të vogël. Të gjithë punëtorët e kryejnë ndonjë punë të caktuar specifike, kurse një grup punëtorësh bashkë e kryen të njëjtën punë. Të gjithë punëtorët në këtë punëtori i kanë atributet e mëposhtme:

- Emrin dhe mbiemrin
- Moshën
- Adresën ku banojnë
- Numrin e orëve që punojnë
- Detyrën e punës
- Rrogën
- Kontratën për punë

Mirëpo, disa punëtorë, përveç attributeve të përmendura më sipër, kanë edhe attribute të veçanta, p.sh. udhëheqësit e një reparti kanë grupin apo repartin të cilin e mbikqyrin. Meqë të gjithë punëtorët i kanë disa attribute të njëjta, nëse është nevoja që ata të paraqiten në program, atëherë do të ishte e udhës që këto attribute të definohen në një klasë të përgjithshme që paraqet punëtorët, pa marrë parasysh funksionet e tyre, p.sh.:

```
class Punetori
{
public:
    string      m_sEmriMbiemri;
    string      m_sAdresa;
    string      m_sDetyra;
    int         m_iRroga;
    int         m_iOret;
    int         m_iMoshë;
};
```

Klasët që paraqesin punëtorë specifike do t'i trashëgojnë atributet dhe funksionet e implementuara për klasën Punetori. Klasët specifike mund të shtojnë funksione specifike që kanë të bëjnë p.sh. me udhëheqësit e repartit, inxhinierët, drejtorët etj.

Hierarkitë e krijuara nga klasët e ndryshme, shpeshherë përfundojnë në të ashtuquajturin Framework që standardizohet duke krijuar librari standarde, të cilat librari përdoren për sisteme të ndryshme. P.sh. në platformën e sistemit operativ Microsoft Windows kemi hierarkinë e klasëve në C++ të quajtur MFC (akronim për Microsoft Foundation Classes). Kjo hierarki përmban klasën që paraqet objektin e thjeshtë (klasa objekt) prej të cilit janë trashëguar

pothuajse të gjitha klasët e tjera. Kjo hierarki (MFC) përdoret për programimin e sistemeve për platformën *Microsoft Windows*. Me përdorimin e kësaj hierarkie, programet mund të shkruhen shumë më shpejt sesa do të shkruheshin nga fillimi.

8.1 Trashëgimi i klasëve

Në gjuhën C++ jo të gjitha klasët duhen të shkruhen nga fillimi, sepse shumica e klasëve trashëgohen nga klasët ekzistuese, duke shfrytëzuar kodin e shkruar më parë. Klasa origjinale, nga e cila trashëgohen klasët e tjera, njihet si *klasa bazë*. Kurse klasët e tjera, të trashëguara nga klasa bazë, njihen si *klasë me prejardhje*.

Kur shpjegojmë trashëgiminë e klasëve në gjuhën C++, ka mbetur si traditë që kjo të bëhet duke marrë si objekt konton e bankës. Edhe në këtë libër do ta shpjegojmë trashëgiminë e klasëve duke marrë parasysh objektin konto si dhe llojet e ndryshme të kontove.

Për ta ilustruar trashëgiminë e klasëve, do të marrim parasysh se banka mundëson disa lloje të kontove, p.sh.:

- Konto për hyrje e dalje të parave.
- Konto për kursim.
- Konto për çeqe
- Konto rrjedhëse, etj.

Të gjitha këto lloje të kontove kanë attribute të ngjashme, mirëpo secila prej tyre ka attribute specifike. Siç përmendëm më parë, kur kemi të bëjmë me objekte të ngjashme, në metodën e programimit objekt-orientues duhet që këto ngjashmëri të përmbliidhen në një objekt të përbashkët (gjeneral). Pastaj, të gjitha objektet e veçanta trashëgohen prej këtij objekti gjeneral dhe shtojnë funksione specifike që kanë të bëjnë vetëm me objektet e trashëguara.

8.1.1 Kontoja e thjeshtë dhe klasa bazë

Së pari do të definojmë klasën bazë për objektet që përfaqësojnë kontot. Kjo klasë do t'i përmbajë funksionet elementare që janë të përbashkëta për të gjitha kontot. P.sh. në të gjitha kontot mund të nxirri ose të depozitoni para, të gjitha kontot kanë bilansin etj. Do të fillojmë me definimin e klasës bazë dhe duke i krijuar klasët e tjera do ta ndryshojmë edhe klasën bazë. Kështu është edhe në jetën e përditshme të programimit, kur kemi të bëjmë me trashëgiminë e klasëve. Së pari shkruhet klasa bazë si skicë e parë dhe me fillimin e shkrimit të klasëve me prejardhje, vërehen mungesat apo tepricat në klasën bazë. Pra edhe në implementimin e hierarkisë së klasëve bëjmë implementimin përsëritës.

Klasa fillestare bazë për kontot do të jetë:

Fajlli konto.h

```
#ifndef Konto_h
#define Konto_h
```

```
#include <iostream>
```

```
class Konto
```

```
{
    friend ostream& operator<< (ostream&, Konto&);
```

```
public:
```

```
    Konto();
```

```
    float Bilansi();
```

```
    void Depozito(float fShuma);
```

```
    void Terhiq(float fShuma);
```

```
private:
```

```
    static int m_iNumriIFundit;
```

```
    int m_iNumriIKontos;
```

```
    float m_fBilansi;
```

```
};
```

```
#endif
```

Fajlli konto.cpp

```
#include <iomanip>
```

```
#include "konto.h"
```

```
int Konto::m_iNumriIFundit = 0;
```

```
Konto::Konto()
```

```
: m_fBilansi(0), m_iNumriIKontos(++m_iNumriIFundit)
```

```
{
```

```
}
```

```
float Konto::Bilansi()
```

```
{
```

```
    return m_fBilansi;
```

```
}
```

```
void Konto::Depozito(float fShuma)
```

```
{
```

```
    m_fBilansi += fShuma;
```

```

}

void Konto::Terhiq(float fShuma)
{
    m_fBilansi -= fShuma;
}

ostream& operator<< (ostream& os, Konto k)
{
    int iPrecizitetiVjeter    = os.precision();
    long lFlags               = os.flags();

    os << setiosflags(ios::showpoint) << setprecision(2)
        << k.Bilansi()
        << setiosflags(lFlags)
        << setprecision(iPrecizitetiVjeter);

    return os;
}

```

Klasa Konto definon funksionet gjenerike që janë të përbashkëta për llojet e ndryshme të kontove:

- Kundrimin e bilansit
- Depozitimn e parave
- Tërheqjen e parave

Çdo konto ka numrin unik për identifikimin e kontos. Për këtë arsye kemi përdorë variablën statike për ruajtjen e numrit të fundit të përdorur, duke e rritur për një sa herë që e deklarojmë një instancë të re të kontos. Në këtë mënyrë garantojmë që numri i të gjitha instancave të kontove në program të jetë unik. T'ju përkujtojmë se variablat statike janë sikurse variablat globale dhe ndërrimi i vlerës së variablës statike është i dukshëm në të gjitha instancat e klasës Konto. Duhet ta kenë parasysh se përdorimi i variablës statike këtu është vetëm për ilustrim. Në programe të jetës së përditshme të dhënat zakonisht ruhen në bazë të dhënash (*database*) dhe numri unik i çdo rekordi administrohet nga vetë baza e të dhënave.

Numri unik i kontos në programin e që përdor klasën Konto, garantohe vetëm në një instancë të programit ekzekutues. Pra, nëse e ekzekutoni programin dy apo më shumë herë, atëherë numri i kontos do të ishte unik vetëm brenda instancave të programeve.

Nëse në vend të variablës statike përdorim variabla normale anëtare, atëherë sa herë që krijojmë një instancë të klasës Konto do të duhej të kujdesemi që ta ndërrojmë numrin e fundit të përdorur në të gjitha instancat e klasës Konto. Kjo nuk është e lehtë të arrihet dhe për më shumë do të ishte joefikase. Me

përdorimin e variablës statike, kemi vetëm një kopje të variablës në të gjitha instancat e kontove dhe çdo ndryshim i vlerës së kësaj variable do të transmetohej automatikisht në të gjitha instancat e Kontos.

8.1.2 Kontoja me interes

Kontot me interes janë kontot e kursimit në të cilat interesi (kamata) është më i mëdh sesa në kontot e thjeshta dhe interesi në këto konto paguhet në perioda të caktuara p.sh. një herë në vit.

Kontoja me interes i ka të gjitha vetitë e kontos së thjeshtë, mirëpo ky tip i kontos ka edhe një funksion që shton interesin (kamatën) për paratë e depozituara në këtë konto. Ne do ta paraqesim konton me interes me anë të klasës `KontoMeInteres` e cila do t'i trashëgojë vetitë nga klasa e thjeshtë `Konto`, p.sh.:

Fajlli `KontoMeInteres.h`

```
#ifndef KontoMeInteres_h
#define KontoMeInteres_h

#include "Konto.h" // fajlli ku kemi definuar klasën Konto

class KontoMeInteres : public Konto
{
public:
    void ShtoInteresin( );
    static void NderroPerqindjenEInteresit(float fPerqindja);
private:
    static float m_fPerqindjaEInteresit;
};

#endif
```

Fajlli `KontoMeInteres.cpp`

```
#include "KontoMeInteres.h"

float KontoMeInteres::m_fPerqindjaEInteresit = 0.0;

void KontoMeInteres::ShtoInteresin()
{
    m_fBalansi *= (1.0 + m_fPerqindjaEInteresit / 100.0);
}

void KontoMeInteres::NderroPerqindjenEInteresit(float fPerqindja)
```

```
{
    m_fPerqindjaEInteresit = fPerqindja;
}
```

Rreshti i kodit:

```
class KontoMeInteres : public Konto
```

definon se klasa `KontoMeInteres` trashëgohet nga klasa `Konto`. Udhëzimi `public` para emrit të klasës `Konto` tregon nivelin e aksesit të funksioneve të trashëguara nga klasa `KontoMeInteres`. Udhëzimi `public` definon se funksionet e klasës bazë (në këtë rast klasës `Konto`) në klasën e trashëguar, kanë të njëjtin nivel të aksesit sikurse klasa bazë. P.sh. funksionet dhe variablat publike në klasën bazë janë po ashtu publike në klasën e trashëguar, funksionet dhe variablat e nivelit `protected` në klasën bazë janë po ashtu `protected` në klasën e trashëguar, si dhe funksionet dhe variablat private janë po ashtu private në klasën e trashëguar. Zakonisht niveli i trashëgimit është gjithnjë `public`, mirëpo ky nivel nuk është bazë, kështu që udhëzimi `public` patjetër duhet të përdoret nëse e keni ndër mend këtë nivel të trashëgimit.

Në klasën `KontoMeInteres` prapë e kemi përdorë variablën anëtare statike, në mënyrë që përqindja e kamatës të jetë e njëjtë për të gjitha instancat e kësaj klase.

Nëse e kundroni me kujdes kodin e mëparshëm, në të cilin e kemi definuar klasën `Konto`, do të vëreni se variabla anëtare e klasës `Konto` `m_fBilansi` është private dhe nuk mund të përdoret drejtpërsëdrejti siç është përdorë në funksionin `ShtoInteresin` anëtar të klasës `KontoMeInteres`. Për këtë arsye, kompajleri do ta paraqesë si gabim kodin e klasës `KontoMeInteres` të shkruar më parë.

Për t'i ikur këtij problemi, duhet ndërruar nivelin e aksesit të variablës `m_fBilansi` anëtare të klasës `Konto`. Nëse e bëjmë publike, atëherë variabla `m_fBilansi` mund të përdoret edhe nga jashtë klasës `Konto` e kjo shpeshherë do të ishte e padëshirueshme. Gjuha C++ përmban nivelin e quajtur `protected` (lexo: protektid) që përdoret për situata si kjo që e kemi tani. Ky nivel ua lejon klasave të trashëguara aksesin në variabla dhe funksione të këtij niveli, mirëpo nuk lejon që këto funksione apo variabla të nivelit `protected` të përdoren nga jashtë klasës (klientët e klasës).

Pra, klasa `Konto` duhet ndërruar në mënyrë që variabla anëtare `m_fBilansi` të jetë `protected`.

```
class Konto
```

```

        friend ostream& operator<< (ostream&, Konto);

public:

    Konto();

    float  Bilansi();
    void   Depozito(float fShuma);
    void   Terhiq(float fShuma);

private:
    static int    m_iNumriIFundit;
    int           m_iNumriIKontos;

protected:
    float         m_fBilansi;
};

```

Kur kemi të bëjmë me hierarki të klasëve shpeshherë kemi nevojë të bëjmë ndërrime në definimin e klasës siç ishte ndërrimi i mëparshëm. Nevoja për ndërrime shpesh vërehet duke implementuar klasët e trashëguara.

Duhet cekur se operatori << për shtypje është definuar vetëm për klasën Konto dhe ky operator nuk trashëgohet në klasën KontoMeInteres. Në shumë kompajlerë ky operator mund të përdoret edhe për klasën e trashëguar, pasi që kompajleri e shndërron klasën e trashëguar automatikisht në klasë bazë.

8.1.3 Super Kontoja

Super Kontoja është e ngjashme me konton me interes, mirëpo interesi (kamata) është më i madh, duke ofruar bonus për paratë e depozituara, por duke e bërë të detyrueshëm një pagesë të vogël për shërbimin e nxerjes së parave.

Funksionet anëtare të klasës KontoMeInteres:

```

void   Terhiq(float fShuma);
void   ShtoInteresin( );

```

duhen të definohen prapë për klasën SuperKonto. Funksionet e definuara për klasën SuperKonto do t'i zëvendësojnë funksionet e klasës KontoMeInteres.

Kurse, funksionet e klasës KontoMeInteres që nuk janë të definuara prapë në klasën SuperKonto, trashëgohen.

Super konton do ta shprehim me anë të klasës SuperKonto e cila trashëgon vetitë e përbashkëta nga klasa KontoMeInteres. T'ju përkujtomë se KontoMeInteres është e trashëguar nga klasa bazë Konto dhe për këtë arsye edhe klasa SuperKonto ka prejardhje dhe trashëgon funksionet nga klasa Konto.

Fajlli SuperKonto.h

```
#ifndef SuperKonto_h
#define SuperKonto_h

#include "KontoMeInteres.h" // fajlli ku kemi definuar klasën
                             // KontoMeInteres

class SuperKonto : public KontoMeInteres
{
public:
    void ShtoInteresin( );
    void Terhiq(float fShuma);

    static void RuajeBonusin(float fBonusi);
    static void RuajeDeniminPerTerheqje(float fDenimi);
private:
    static float m_fBonusi;
    static float m_fDenimiPerTerheqje;
};

#endif
```

Fajlli SuperKonto.cpp

```
#include "SuperKonto.h"

float SuperKonto::m_fBonusi = 0.0;
float SuperKonto::m_fDenimiPerTerheqje = 0.0;

void SuperKonto::ShtoInteresin()
{
    m_fBilansi += (1.0 + (m_fPerqindjaEInteresit + m_fBonusi)
                  / 100.0);
}

void SuperKonto::Terhiq(float fShuma)
{
    m_fBilansi -= m_fBilansi + m_fDenimiPerTerheqje;
```



```

class KontoMeCeqe : public Konto
{
public:
    KontoMeCeqe() : m_iNumriIFunditICekut(0) {}
    void LarjaEQekut(int iNumriICekut, float fShuma);

protected:
    int m_iNumriIFunditICekut;
};

#endif

```

Fajlli KontoMeCeqe.cpp

```
#include "KontoMeCeqe.h"
```

```

void KontoMeCeqe::LarjaEQekut(int iNumriICekut, float fShuma)
{
    m_iNumriIFunditICekut    = iNumriICekut;
    m_fBilansi                -= fShuma;
}

```

Siç e vërejmë në klasën `KontoMeCeqe` kemi definuar kontruktoren bazë që inicion variablën anëtare `m_iNumriIFunditICekut` me zero. Duhet ta keni parasysh se kontruktoret nuk trashëgojnë nga klasët bazë dhe se kontruktori bazë i klasës `KontoMeCeqe` nuk e zëvendëson kontruktoren bazë të klasës bazë. Në të kundërtën, kur kompajleri thërret kontruktoren e klasës `KontoMeInteres`, së pari e thërret kontruktoren bazë të klasës `Konto` e pastaj kontruktoren e klasës `KontoMeInteres` (shih kapitullin për kontruktoret).

8.1.5 Kontoja me interes dhe me çeqe

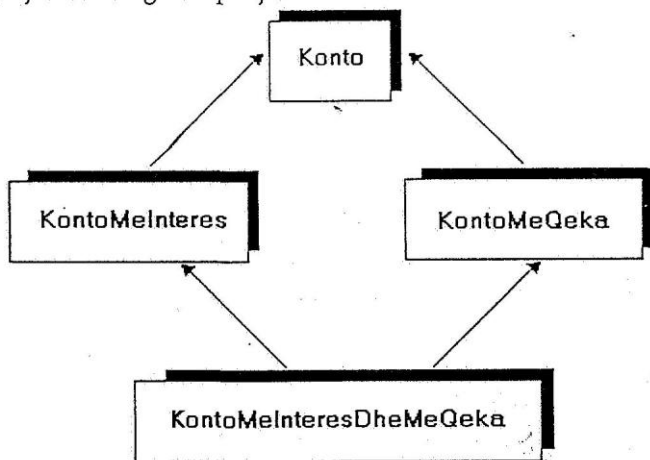
Kontoja me interes dhe me çeqe është kontoja që paguan interes (kamatë) në paratë e depozituara, si dhe përmban çeqe në të njëjtin numër të kontos. Logjikisht, kjo konto është kombinim i kontos me interes si dhe kontos me çeqe. Gjuha C++ është e njohur për mundësitë e shfrytëzimit të kodit të shkruar më parë. Pasi kemi definuar klasët që definojnë konton me interes dhe konton me çeqe në veçanti, do të ishte e logjishme që t'i kombinojmë këto klasë. Këtë na e mundëson gjuha C++, pra kombinimin e klasës `KontoMeInteres` dhe të klasës `KontoMeCeqe` për ta formuar klasën e re që përmban të dy llojet e kontove.

Kombinimi i klasëve të shkruara më parë bëhet me anë të trashëgimit të shumfishtë, p.sh.:

```
class KontoMeInteresDheMeCeqe : public KontoMeInteres,
                                public KontoMeCeqe
{
};
```

Siç e vërejmë klasa `KontoMeInteresDheMeCeqe` nuk përmban asnjë definim të funksioneve anëtare apo variabla anëtare. Kjo është plotësisht në rregull, pasi që klasa `KontoMeInteresDheMeCeqe` trashëgon funksionet dhe variablat anëtare nga klasët `KontoMeInteres` dhe `KontoMeCeqe`. Klasa `KontoMeInteresDheMeCeqe` përmban të gjitha funksionet nga të dy klasët e përmendura më sipër.

Në mënyrë grafike prejardhja e klasës `KontoMeInteresDheMeCeqe` do të paraqitet sikur në figurën që vijon.



Prejardhja e klasës `KontoMeInteresDheMeCeqe`

Mirëpo në rastet e trashëgimit të shumfishtë kemi disa probleme të vogla. P.sh. nëse e definojmë një variabël të tipit `KontoMeInteresDheMeCeqe` dhe thërrasim funksionin `Depozito`:

```
KontoMeInteresDheMeCeqe k1qKonto;
k1qKonto.Depozito(525.00); // gabim ???
```

Shumë kompajlerë modernë këtë do ta paraqesin si gabim, për arsye se klasa `KontoMeInteresDheMeCeqe` e trashëgon funksionin `Depozito` nga të dy klasët. Nëse e vrojtoni kodin e mëparshëm, do të vëreni se klasët `KontoMeInteres` dhe `KontoMeCeqe` përmbajnë funksionin `Depozito` të trashëguar nga klasa `Konto`. Në këtë rast, klasa `KontoMeInteresDheMeCeqe` trashëgon dy kopje të funksioneve dhe variablave anëtare të përbashkëta, të definuara në klasën `Konto`.

Në shembullin e mësipërm, kompajleri nuk vëren për cilën kopje e kemi fjalën, edhe pse në këtë rast të dy kopjet e funksionit `Depozito` janë të njëjta, për arsye se janë të trashëguara nga e njëjta klasë. Në raste të tjera, këto funksione mund të jenë të ndryshme, p.sh. nëse klasa `KontoMeCeqe` e definon prapë këtë funksion, ose kjo klasë ka prejardhje tjetër nga klasa `Konto`.

Pa marrë parasysh nëse kopjet janë të njëjta apo jo, gjuha C++ na mundëson që ta thërrasim kopjen specifike duke e paraprirë me emrin e klasës, p.sh.:

```
kiqKonto.KontoMeInteres::Depozito(525.07);
```

Në kodin e mësipërm thërrasim kopjen e trashëguar nga klasa `KontoMeInteres`.

Në rastet kur kemi kopjet e njëjta të funksioneve anëtare dhe variablave anëtare, nuk është e përshtatshme dhe nuk rekomandohet ta paraprijmë emrin e funksionit apo të variablës anëtare me emrin e klasës. Kjo jo vetëm që shkakton konfuzion gjatë leximit të kodit, por për më shumë, është logjikisht e arsyeshme të kemi vetëm një kopje në raste kur funksionet janë të njëjta.

Gjuha C++ na mundëson të kemi vetëm një kopje të funksioneve anëtare duke përdorë udhëzimin virtual gjatë trashëgimit nga klasa bazë. Udhëzimi virtual definon se të gjitha trashëgimtarët e ardhshme të klasës do të kenë vetëm një kopje të funksioneve anëtare, madje edhe në raste të trashëgimeve të shumëfishta.

Për ta zgjidhur problemin e mësipërm, duhet ndërruar definimin e klasëve `KontoMeInteres` dhe `KontoMeCeqe` kështu:

```
class KontoMeInteres : virtual public Konto
{
    // definimi i funksioneve dhe variablave
    // anëtare sikur më parë
};

class KontoMeCeqe : virtual public Konto
{
    // definimi i funksioneve dhe variablave
```



```
// anëtare sikur më parë
```

```
};
```

Tani klasa `KontoMeInteresDheMeCege` trashëgon vetëm një kopje të funksioneve dhe variabla anëtare të klasës `KontoMeInteres` dhe `KontoMeCege`, të cilat janë të trashëguara nga klasa `Konto`.

Kodi i mësipërm tani nuk do të paraqesë ndonjë problem:

```
KontoMeInteresDheMeCege kiqKonto;  
kiqKonto.Depozito(525.00); // në rregull
```

8.1.6 Zëvendësimet standarde

Në kapitujt e mëparshëm (shih 6.3.2) kemi treguar se si konstruktorët shndërrues përdoren për shndërrimin e një tipi të objekti në një tip tjetër. Këto shndërrime bëhen automatikisht nga kompajleri, sidomos për parametrat e funksioneve që presin tip të ndryshëm nga ai i pasuar. Mirëpo që këto shndërrime të bëhen automatikisht, ju duhet t'i definoni konstruktorët shndërrues.

Për klasët në hierarki nuk nevojiten konstruktorët shndërrues prej një klase më specifike në atë të klasës bazë, nga e cila është trashëguar në mënyrë direkte apo indirekte.

P.sh. nëse kemi një funksion që shtypë bilansin e kontos:

```
void ShtypKonton(Konto k)  
{  
    cout << "Bilansi i kontos është: "  
        << k.Bilansi();  
}
```

Ky funksion mund të përdoret për të gjitha tipet e objekteve të tipit `Konto` ose të objekteve që kanë prejardhje nga kjo klasë (pa pasur nevojë për konstruktorë shndërrues). Kjo është e mundshme për arsye se kompajleri din si ti shndërrojë automatikisht tipet e klasëve specifike në tipe të klasëve bazë. Nëse pyetni se si është e mundur kjo, përgjigja do të ishte se klasët e trashëguara kanë të njëjta funksione dhe variabla anëtare si klasa bazë, si dhe ndonjëherë edhe disa funksione dhe variabla më tepër të definuara në klasën e trashëguar. Shndërrimi bëhet në mënyrë automatike, duke i harruar këto funksione dhe variabla anëtare të tepërta të definuara në klasët e trashëguara,

pra përdorimin vetëm të funksioneve dhe të variablave anëtare të definuara në klasën bazë.

Shndërrimi automatik bëhet edhe në tregues (pointer) të klasëve në hierarki. Kjo d.m.th. se një tregues i tipit Konto mund të tregojë në një objekt të tipit KontoMeInteres. Kjo veti në C++ është e njohur si polimorfizëm, gjë për të cilën do të flasim në kapitull të veçantë.

Duhet ta keni parasysh se shndërrimet automatike nuk mund të bëhen prej një tipi të klasës bazë në një tip të klasës së trashëguar, për arsye se tipi i trashëguar mund të ketë funksione dhe variabla anëtare të tepërta, të cilat kompajleri nuk mund ta dijë si t'i krijojë për klasën bazë.

Për shembull, nëse kemi funksionin për shtypjen e bilansit të kontos me interes dhe me çeq:

```
void ShtypKonton(KontoMeInteresDheMeCeqe k)
{
    cout << "Bilansi i kontos është: "
          << k.Bilansi();
}
```

Ky funksion nuk mund të përdoret për tipet e klasëve Konto apo KontoMeInteres apo KontoMeCeqe, për arsye se kompajleri nuk din si t'i shndërrojë në mënyrë automatike tipet e klasëve Konto dhe KontoMeInteres në tipin e klasës KontoMeInteresDheMeCeqe.

8.2 Llojet e trashëgimit

Siç keni vërejtur më parë, gjatë definimit të një klase e cila trashëgohet nga klasa e definuar më parë, përdorim shprehjen për të treguar nivelin e trashëgimit.

Niveli i trashëgimit definon nivelin e aksesit të funksioneve dhe variablave anëtare të klasës që trashëgon nga klasa bazë. Këto nivele të trashëgimit janë:

- **public** - Ky nivel i trashëgimit definon që funksionet dhe variablave trashëguara nga klasa bazë, kanë të njëjtin nivel të aksesit sikurse ai i klasës bazë. Funksionet dhe variablave trashëguara të nivelit **public** mbesin **public**; ato **protected** mbesin **protected** dhe ato **private** mbesin **private**.
- **protected** - Në këtë nivel të trashëgimit, funksionet dhe variablave **public** anëtare të klasës së trashëguar bëhen **protected**, kurse funksionet dhe variablave të niveleve të tjera mbesin siç janë.
- **private** - Në trashëgimin **private** funksionet dhe variablave **public** dhe **protected** të klasës bazë bëhen **private** në klasën e trashëguar.

Në të shumtën e rasteve përdoret niveli i trashëgimit **public**. Mirëpo në raste kur kemi nevojë t'i fshehim funksionet dhe variablave anëtare të klasës bazë, përdorim trashëgimin **private**. Nëse niveli i trashëgimit nuk përcaktohet, atëherë kompajleri përdor nivelin **private**. Pasi që në nivelin e trashëgimit **private** fshihen të gjitha funksionet dhe variablave anëtare për përdoruesit e klasës, atëherë duhet përcaktuar në mënyrë eksplicite se cilat funksione apo variabla janë të dukshme për përdoruesin.

T'i shqyrtojmë p.sh. kontot të cilat lejojnë vetëm depozitimin e parave dhe kundrimin e bilansit. Nëse në këto konto doni t'i tërhiqni paratë, atëherë duhet mbyllur konton apo transferuar paratë në kontot e tjera. Këtë lloj kontoje do ta definojmë në klasën **KontoPerKursim**, e cila trashëgohet nga klasa **Konto** në nivelin **private**, pra duhet ta bëjmë fshehjen e funksioneve anëtare të klasës **Konto**. Mirëpo, pasi që do të lejojmë kundrimin e bilansit dhe depozitimin e parave, atëherë do t'i përcaktojmë në mënyrë eksplicite këto funksione si **public**:

```
class KontoPerKursim : private Konto
{
    public:
        Konto::Bilansi;
        Konto::Depozito;
};
```

Së pari klasa `KontoPerKursim` definon që të gjithat funksionet dhe variablat anëtare të trashëguara nga klasa `Konto` të jenë private. Mirëpo, në trupin e klasës kemi zëvendësuar nivelin e aksesit të dy funksioneve në mënyrë individuale. Pra funksionet:

```
float Bilansi();
void Depozito(float fShuma);
```

bëhen `public` dhe mund të përdoren nga klientët e kësaj klase.

Ushtrime

1. Në fillim të këtij kapitulli kemi definuar klasën `Punetori`. Definoni klasët specifike që paraqesin udhëheqësit e repartit, inxhinierët dhe drejtorët gjeneralë, duke trashëguar nga klasa `Punetori`.
2. Shpjegoni pse është më e arsyeshme të trashëgohet një klasë e definuar më parë, sesa të definohet klasa e re nga fillimi.
3. Gjeni objektet e jetës së përditshme që kanë lidhje mes vete dhe mund të paraqiten me hierarki klasësh. Caktoni virtytet e përbashkëta të të gjitha objekteve dhe definoni atq në klasën bazë.
4. Shkruani një program që do t'i testonte të gjitha klasët e definuara në këtë kapitull që përfaqësojnë kontot e ndryshme. Përdorni metodën e organizimit të kodit të përmendur në kapitullin e kaluar, ku definimi i klasës bëhet në fajllin header dhe implementimi i klasës bëhet në fajllin tjetër (p.sh. me prapashtesën `cpp`).

Përmbledhje

Në këtë kapitull kemi folë për trashëgimin dhe rëndësinë e trashëgimit në hierarkinë e klasëve. Gjuha C++ mundëson përdorimin e kodit të shkruar më parë, nëpërmjet trashëgimit të klasëve të definuara më parë. Trashëgimi i klasëve është shumë i rëndësishëm sepse lejon përshtatjen e klasëve të trashëguara për problemet specifike, duke lejuar ridefinimin e funksioneve si dhe krijimin e funksioneve të reja për klasët specifike.

Polimorfizmi dhe lidhja dinamike

Gjuha C++, siç kemi përmendur në kapitullin e kaluar, mundëson përdorimin e kodit të shkruar më parë me anë të trashëgimit, pra trashëgimin e funksioneve të përdorshme dhe zëvendësimin e funksioneve apo përmirsimin e funksioneve të papërdorshme. Me anë të trashëgimit i ikim definimit të klasës dhe shkrimit të kodit nga fillimi.

Përparësia tjetër e trashëgimit dhe e hierarkisë së klasëve në C++, është thjeshtësimi konceptual i kodit dhe i problemit të zgjidhur nga programi. Për shembull, të gjitha kontot (qofshin edhe të tipeve të ndryshme) do të ishte e arsyeshme të trajtoheshin si një tip, kur kemi të bëjmë me funksionet e përbashkëta për të gjitha tipet. P.sh. të gjitha llojet e kontove përmbajnë funksionin `Bilansi` i cili mund të thurret pa marrë parasysh se cilit tip i takon objekti. Në jetën e përditshme, kontot njihen si një tip i objektit (kur kemi të bëjmë me funksionet e përbashkëta të të gjitha llojeve të kontove) pa marrë parasysh vetitë e veçanta për llojet e ndryshme. Kur kemi të bëjmë me vetitë e veçanta të disa tipeve të kontove, p.sh.. shtimin e kamatës, atëherë ky funksion përdoret vetëm për këto tipe. Me fjalë të tjera, në jetën e përditshme, kur disa objekte të tipeve të ndryshme kryejnë të njëjtin funksion, ato njihen si një tip, përderisa i përdorim vetëm funksionet e përbashkëta. Kur kemi të bëjmë me funksione të veçanta, atëherë objektet llogariten si objekte të tipeve të ndryshme.

Në gjuhën C++, e njëjta gjë mundësohet kur kemi të bëjmë me hierarki të klasëve dhe përdorimin e funksioneve të klasës bazë. Nëse objektet janë të klasëve të veçanta (jo në hierarki), atëherë këto objekte nuk mund të trajtohen si një tip.

Për t'i ilustruar përpasitë e trajtimit të objekteve të tipeve të ndryshme si një tip, të marrim shembullin në vazhdim:

Në fund të çdo viti banka bën bilansin për çdo konto (pa marrë parasysh tipin e kontos). Për të llogaritur bilansin në fund të vitit, do të ishte e arsyeshme ta definonim një funksion anëtar të klasës `Konto`, ta quajmë `BilansiIViti`, i cili do ta llogarisë bilansin në fund të vitit. Klasët e tjera më specifike duhet ta zëvendësojnë këtë funksion duke shtuar kamatën për një vit.

```
class Konto
```

```

{
    // definimi i funksioneve dhe variablave
    // anëtare sikur më parë

    virtual void BilansiIViti()
    {
        cout << "Bilansi i kontos "
              << m_iNumriIKontos << " është "
              << Bilansi();
    }
};

```

Kurse klasa KontomeInteres do ta zëvendësonte funksionin BilansiIViti kështu:

```

class KontomeInteres
{
    // definimi i funksioneve dhe variablave
    // anëtare sikur më parë

    virtual void BilansiIViti()
    {
        ShtoInteresin();
        Konto::BilansiIViti();
    }
};

```

Për ta llogaritur bilansin në fund të vitit për të gjitha llojet e tipeve të objekteve konto, do ta definojmë një funksion global që pranon si parametër *array* kontove (grup të kontove) dhe numrin e kontove në *array*. Duhet ta k parasysh se këtu kemi përdorë tipin *array* vetëm për ilustrim, dhe se ky ti nuk preferohet të përdoret në situata kur numri i elementeve të grupit mu të rritet apo të zvogëlohet.

Funksionin global do ta definojmë kështu:

```

void FundiIViti(Konto ka[], int iNumriIKontove)
{
    for (int i=0; i < iNumriIKontove; i++)
    {
        ka[i].BilansiIViti();
    }
}

```

Pra, ky funksion i trajton të gjitha kontot si një tip dhe struktura konceptuale funksionit është shumë më e thjeshtë sesa funksioni që do të llogariste llojet kontove të definuara jo në hierarki. Nëse kemi me miliona konto, funksionin e definuar më sipër nuk do të duhej të ndryshojmë kodin për :

përket numrit të kontove. Kurse funksioni që do të llogariste kontot e definuara jo në hierarki, do të duhej të definonte variabla për të gjitha llojet e objekteve dhe ato nuk do të mund të grupoheshin në *array* apo ndonjë objekt tjetër që përmban grupe të objekteve.

Në kodin që vijon kemi deklaruar variabla të tipeve të ndryshme të kontove:

```
void main ()
{
    Konto          k;
    KontoMeInteres ki;
    SuperKonto     ks;

    k.Depozito(500.00);
    ki.Depozito(800.00);
    ks.Depozito(1000.00);

    Konto ka[3] = k, ki, ks;
    FundiIVitit(ka, 3);
}
```

Në funksionin *main* kemi deklaruar variabla të tipeve të ndryshme të kontove dhe i kemi inicuar me anë të funksionit *Depozito*. Për t'i ikur këtij hapi të "tepruar" për iniciimin e variablës, do t'i definonim konstruktorët që marrin si parametër variablën e tipit *float* dhe iniciojnë variablën anëtare *m_fBilansi*. Për këtë arsye, klasët me disa konstruktorë bëjnë lehtësimin e inicimit të tyre dhe e thjeshtësojnë kodin.

Pas ekzekutimit të programit të mësipërm, rezultati do të ishte:

```
Bilansi i kontos 1  është  500.00
Bilansi i kontos 2  është  800.00
Bilansi i kontos 3  është 1000.00
```

Siç mund ta vëreni, edhe pse kemi përdorë objektin e tipit *KontoMeInteres* dhe *SuperKonto*, të cilat e shtojnë interesin në fund të vitit, rezultati është sikur ta kishim përdorë konton e thjeshtë. Arsyeja për këtë rezultat (natyrisht të gabuar) është se kompajleri e shndërron objektin e tipit më konkret në objekt të tipit bazë në mënyrë automatike, nëse pritet objekti i tipit bazë. Nëse e shikoni funksionin *FundiIVitit*, do të vëreni se ky funksion pret si parametër tipin *array* të klasës *Konto*. Gjatë inicimit të tipit *array*, të gjitha objektet (edhe të tipeve më konkrete, si p.sh. në këtë rast objekti i tipit *KontoMeInteres* dhe objekti i tipit *SuperKonto*) shndërrohen në objekte të tipit bazë (*Konto*).

Kur kompajleri e thërret funksionin anëtar `Bilansiviti` për të gjitha objektet në `array`, e vëren se tipit i këtyre objekteve është `xonto` (kuptohet pas janë shndërruar automatikisht në këtë tip) dhe se informatat e tipeve më konkrete janë humbur.

9.1 Funksionet virtual

Për të zgjidhur problemin e shndërrimit automatik nga kompajleri nga një tip në tjetrin, duhet që në vend të tipit bazë ta përdorim treguesin e tipit bazë. Në këtë mënyrë kompajleri e shndërron treguesin e tipit konkret në tregues të tipit bazë, pa i humbur informatat e tepërta të tipit konkret. Duhet t'ju përkujtojmë se gjuha C++ lejon që treguesit e klasës bazë të tregojnë në tipe të klasës më konkrete të trashëguara nga klasa bazë, pa pasur nevojë ta përdorim operatorin cast.

Si është e mundur që treguesi i një tipi që tregon në objekt të ndonjë tipi tjetër ta thërras funksionin e objektit që tregon e jo të tipit të treguesit?

Gjuha C++ lejon të ashtuquajturën “lidhje e vonuar” që mundëson thirrjen e funksioneve të objektit që përmbahet në memorien e treguar nga treguesi e jo të vetë tipit të treguesit. P.sh. nëse kemi një tregues të tipit `Konto`, i cili tregon në objekt të tipit `KontoMeInteres`, thirrja e funksionit `BilansiIVitit` shkakton thirrjen e funksionit të tipit `KontoMeInteres` e jo të tipit `Konto` (pra të tipit të treguesit):

```
Konto*          pKonto;
KontoMeInteres  ki;

ki.Depozito(800.00);

// inico treguesin e tipit Konto*
pKonto = &ki;

pKonto->BilansiIVitit();
```

Në këtë rast, funksioni `BilansiIVitit` do ta shtojë interesin (kamatën) në objektin `ki` për arsye se kompajleri e thërret funksionin anëtar të klasës `KontoMeInteres` e jo të klasës `Konto` (tipit të treguesi `pKonto`).

Lidhja e vonuar mundësohet vetëm për funksionet virtual (funksionet të cilat kanë si parashtesë shprehjen virtual gjatë definimit të tyre). Lidhja e vonuar mundësohet për arsye se kompajleri i gjuhës C++ krijon tabelën e funksioneve virtual (të njohur si `vtable`), dhe se nëse njëri prej funksioneve në listë thirret nga klienti, atëherë kompajleri shikon gjatë ekzekutimit se cili tip të objektit i takon ky funksion dhe e thërret funksionin përkatës.

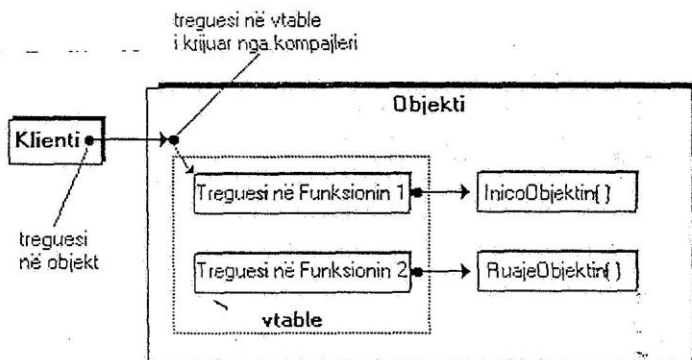
Le ta definojmë klasën `Data` si vijon:

```

class Data
{
public:
    virtual void InicoObjektin();
    virtual void RuajeObjektin();
};

```

Atëherë kompajleri do ta krijojë listën e funksioneve virtual (vtable) sikur në figurën më poshtë:



Vtable për objektin e tipit Data.

Nëse e definojmë një tregues të tipit Data, i cili inicohet me ndonjë objekt të tipit Data apo ndonjë objekt të tipit të trashëguar nga klasa Data, atëherë treguesi i definuar (i tipit Data) tregon në një tregues të krijuar nga kompajleri. Ky tregues tregon në tabelë të treguesve të tjerë (vtable), të cilët tregues tregojnë në funksionet virtual, anëtare të klasës Data apo të klasëve të trashëguara nga kjo klasë.

Për të ilustruar edhe njëherë se funksioni i cilës klasë thirret gjatë ekzekutimit të programit, vëreni kodin e mëposhtëm:

```

class KlasaBaze
{
public:
    virtual void FunksioniVirtual();
    void FunksioniIThjeshte();
};

class KlasaETrasheguar : public KlasaBaze
{
public:

```


thërret funksionin e tipeve të objekteve të ndryshme, mvarësisht nga objekti që tregon treguesi pBaze. Kjo në gjuhën C++ është e njohur si *Polimorfizëm*. Polimorfizmi është shumë i përdorshëm në C++ për arsye se thjeshtëson problemin dhe mundëson pasqyrimin e problemit që kërkon zgjidhje me kodin e shkruar në C++. Sikurse çdo gjë thotë, edhe polimorfizmi i ka vetitë e mira dhe vetitë e këqija. Vetitë e keqe të tij është se e vështirson përcjelljen e strukturës së kodit (leximin e kodit) gjatë testimit dhe gjatë rileximit të kodit për gjetjen e gabimeve. Mirëpo, vetitë e mira të polimorfizmit i tejkalojnë të këqijat dhe rekomandohet që kjo metodë të përdoret pa ngurrim në raste kur është e nevojshme. Pa dyshim, struktura e programeve të shkruara në gjuhën C apo gjuhë të tjera strukturale (si p.sh. Pascal) është më e lexueshme dhe më e lehtë të përcjellet sesa struktura e gjuhëve objekt-orientuese (duke përfshirë edhe gjuhën C++). Natyrisht se është e preferueshme të dokumentohet çdo program i shkruar në çfarëdo gjuhe, mirëpo programet e shkruara në gjuhët objekt-orientuese rekomandohen të dokumentohen me metoda të njohura për modelimin e kodit (si p.sh. metoda UML – Unified Modeling Language). Metodat për modelimin e kodit, shpeshherë lehtësojnë leximin dhe kuptimin e kodit, si dhe mundësojnë përmirësimin e strukturës së përgjithshme të kodit.

Mëqë programet në ditët e sotme janë gjithnjë e më të mëdha, modelimi i programeve është bërë një pjesë e patjetërsueshme e procedurës së programimit. Krijimi i programit pa model, është sikurse ndërtimi i shtëpisë pa plan.

Në fund do ta paraqesim zgjidhjen e problemit të funksionit FundiIVitit dhe të inicimit të variablës array.

```
void FundiIVitit(Konto* pka[], int iNumriIKontove)
{
    for (int i=0; i < iNumriIKontove; i++)
    {
        pka[i]->BilansiIVitit();
    }
}

void main ()
{
    Konto          k;
    KontoMeInteres ki;
    SuperKonto     ks;

    k.Depozito(500.00);
    ki.Depozito(800.00);
    ks.Depozito(1000.00);
}
```

```
Konto* pka[3] = &k, &ki, &ks;  
Fundivitet(ka, 3);  
}
```

Pra, e kemi përdorë treguesin në objektin e klasës bazë (në vend të vetë objektit) për t'i ikur problemit të shndërrimit automatik nga kompajleri prej klasës së trashëguar në klasë bazë, si dhe humbjes së informatave të klasëve konkrete.

9.2 Klasët abstrakte

Në rastet kur kemi të bëjmë me objekte të ngjashme, është e natyrshme ta definojmë *interfejsin* e përbashkët në klasën bazë dhe nga kjo klasë ta trashëgojmë çdo tip tjetër më konkret. Mirëpo, në të shumtën e rasteve klasa bazë është atëherë e përgjithshme sa që është e pakuptimtë ta definojmë ndonjë objekt të tipit të klasës bazë.

Ndonjëherë kemi funksione të ngjashme për objektet konkrete, mirëpo nuk mund t'i definojmë këto funksione në klasën bazë, për arsye se këto funksione nuk kanë kuptim për objektet gjenerale.

Për t'i shpjeguar klasët abstrakte, do të marrim si shembull programin për vizatimin e figurave gjeometrike të ndryshme në monitor. Ky program do ta vizatojë p.sh. trekëndëshin, katrorin, rrethin etj.

Do të ishte e udhës që vizatimi i çdo figure gjeometrike të kontrollohet nga klasë të veçanta që e definojnë figurën gjeometrike. P.sh., klasa *Trekendeshi* ta definojë figurën gjeometrike të trekëndëshit, klasa *Katrori* ta definojë figurën gjeometrike të katrorit si dhe klasa *Rrethi* ta definojë figurën gjeometrike të rrehtit. Të gjitha këto klasë si dhe klasët e tjera që i definojnë figurat e tjera gjeometrike, kanë të dhëna të përbashkëta si dhe funksione të përbashkëta, p.sh.: të gjitha figurat gjeometrike e kanë pozitën në koordinatat x dhe y , dhe meqë të gjitha figurat vizatohen, e kanë funksionin anëtar vizato. Këto të dhëna dhe funksionet e përbashkëta do të ishte e udhës t'i deklaranim në një klasë të përbashkët, p.sh. në klasën *FiguraGjeometrike*. Kjo klasë do t'i përmbante koordinatat x dhe y si dhe funksionin vizato:

```
class FiguraGjeometrike
{
    public:
        void Vizato();
    protected:
        int m_iPozitaX;
        int m_iPozitaY;
};
```

Nëse do ta definonim funksionin anëtar vizato të klasës *FiguraGjeometrike*, atëherë ky funksion do të ishte i zbrazët për arsye se *FiguraGjeometrike* është klasë gjenerike:

```
void FiguraGjeometrike::Vizato()
{
}
```

Duke qenë se objekti i tipit FiguraGjeometrike nuk kryen asnjë funksion të vlefshëm (pra e ka funksionin të zbrazët), a është e arsyeshme të deklarojmë ndonjë objekt të këtij tipi?

Natyrisht se jo. Atëherë ju do të pyetni si mund t'i detyrojmë përdoruesit e hierarkisë së klasëve për figura gjeometrike, të mos deklarojnë objekt të tipit të klasës bazë, por vetëm tregues të tipit të klasës bazë.

Përderisa e definojmë trupin e funksionit vizato në klasën FiguraGjeometrike, nuk mund t'i detyrojmë përdoruesin të mos e krijoj ndonjë objekt të këtij tipi. Duhet cekur se kjo nuk është plotësisht e saktë, pasi që C++ mundëson definimin e konstruktorit të klasës në nivelin private ose protected, p.sh.:

```
class FiguraGjeometrike
{
public:
    void Vizato();
protected:
    int m_iPozitaX;
    int m_iPozitaY;
private:
    // konstruktori i klasës FiguraGjeometrike
    FiguraGjeometrike();
};
```

Kjo metodë për detyrimin e përdoruesit të klasës FiguraGjeometrike të mos deklarojë objekt të këtij tipi, është plotësisht e vlefshme.

Gjuha C++ mundëson zgjidhjen e problemeve në disa mënyra, ku po ashtu detyrimi i përdoruesit për të mos e deklaruar një objekt, por vetëm tregues të objektit, mund të bëhet me anë të definimit të funksionit virtual, të pastër (pa trupin e funksionit) dhe duke e inicuar funksionin me zero, p.sh.:

```
class FiguraGjeometrike
{
public:
    // funksioni virtual i vërtetë
    virtual void Vizato() = 0;
protected:
    int m_iPozitaX;
    int m_iPozitaY;
};
```

Metoda e dytë është më e përdorshme dhe më e arsyeshme, pasi që funksioni i zbrazët nuk ka kuptim dhe në raste të këtilla jemi të interesuar vetëm për interfejsin dhe ngjashmërinë e objekteve. Nëse të gjitha objektet e trashëguara

nga objekti bazë kanë të njëjtin funksion, atëherë është e kuptimshme që definimi i trupit të këtyre funksioneve të bëhet në klasën bazë.

Me iniciimin e funksioneve me zero, krijojmë klasë abstrakte për të cilat funksione nuk kemi nevojë ta definojmë trupin e tyre. Klasët abstrakte mund të përdoren vetëm si skelet apo bazë për klasët e trashëguara. Klasët e trashëguara duhet t'i zëvendësojnë të gjitha funksionet virtual të pastra, përndryshe edhe klasët e trashëguara mbesin abstrakte dhe sikur më parë nuk mund të definojmë objekt të këtyre klasëve.

P.sh.. nëse kemi klasën Baze me dy funksione virtual të pastra:

```
class Baze
{
public:
    virtual void FunksioniIPare()    =0;
    virtual void FunksioniIDyte()    =0;
};

class KlasaETrasheguar : public Baze
{
public:
    virtual void FunksioniIPare()
    {
        cout << "Funksioni i Pare";
    }
};
```

Klasa KlasaETrasheguar është ende abstrakte, pasi që përmban funksionin pastër virtual FunksioniIDyte dhe nëse tentoni të deklaroni objekt të tipit KlasaETrasheguar, kompajleri do ta paraqesë si gabim:

```
KlasaETrasheguar k; //gabim
```

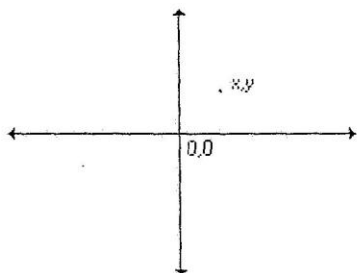
Cila është përparësia e klasëve abstrakte (me funksione virtual të pastra) nga klasët normale me funksione virtual të zbraza?

Përparësia e klasëve abstrakte është se definimi i klasës është me i qartë, duke shprehur në mënyrë eksplicite se funksioni përkatës nuk ka kuptim për klasë bazë. Gjithashtu, për përdorimin e ndonjë objekti me të njëjtin *interfejs* duhet të kemi klasë që zëvendëson të gjitha funksionet virtual të pastra.

Klasët me funksione virtual të zbraza nuk janë të qarta dhe paqetë, na duhet të definojmë konstruktorin në nivel të aksesit protected apo privat

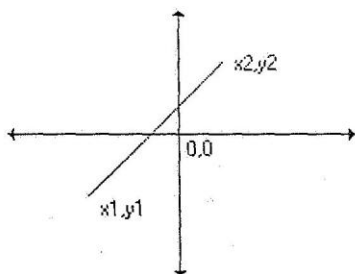
për t'i detyruar përdoruesit të mos deklarojnë ndonjë objekt të tipit të këtyre klasëve.

T'i kthehemi prapë programit për vizatimin e figurave gjeometrike. Për t'i deklaruar klasët që paraqesin figurat gjeometrike (trekëndëshin, katrorin dhe rrethin) do të supozojmë se libraritë e gjuhës programuese përmbajnë këto funksione:



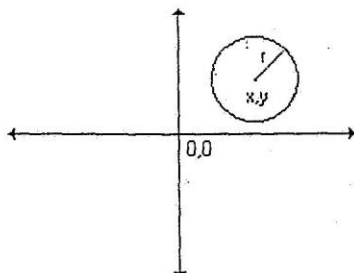
```
// për vizatimin e një pike në
// monitor në pozitën x,y
```

```
void pika(int x, int y);
```



```
// për vizatimin e një vije prej
// pozitës x1,y1 deri
// në pozitën x2,y2
```

```
void vija(int x1, int y1,
          int x2, int y2);
```



```
// për vizatimin e rrethit në
// qendrën x,y dhe rrezen r
void rrethi(int x, int y, int r);
```

Atëherë, klasët që prezantojnë figurat gjeometrike do t'i definojmë kështu:

```

class Trekendeshi
{
public:
    // konstruktori
    Trekendeshi(int x1, int y1,
                int x2, int y2,
                int x3, int y3)
        :FiguraGjeometrike(x1, y1),
          m_iPozitaX2(x2), m_iPozitaY2(y2),
          m_iPozitaX3(x3), m_iPozitaY3(y3) {}

    void Vizato();

private:
    int    m_iPozitaX2;
    int    m_iPozitaY2;
    int    m_iPozitaX3;
    int    m_iPozitaY3;
};

void Trekendeshi::Vizato()
{
    vija(m_iPozitaX, m_iPozitaY, m_iPozitaX2, m_iPozitaY2);
    vija(m_iPozitaX2, m_iPozitaY2, m_iPozitaX3, m_iPozitaY3);
    vija(m_iPozitaX3, m_iPozitaY3, m_iPozitaX, m_iPozitaY);
}

```

Klasën për paraqitjen e katrorit do ta definojmë kështu:

```

class Katrori
{
public:
    Katrori(int x1, int y1, int iBrinja)
        :FiguraGjeometrike(x1, y1), m_iBrinja(iBrinja) {}

    void Vizato();

private:
    int    m_iBrinja;
};

void Katrori::Vizato()
{
    vija(m_iPozitaX - (m_iBrinja / 2),
         m_iPozitaY + (m_iBrinja / 2),
         m_iPozitaX + (m_iBrinja / 2),
         m_iPozitaY - (m_iBrinja / 2));
}

```

```

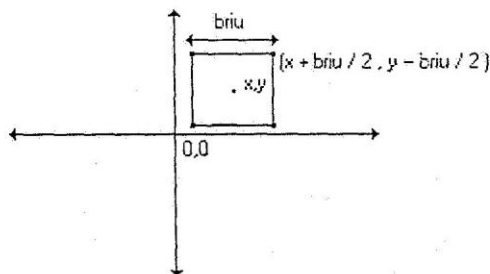
vija(m_iPozitaX + (m_iBrinja / 2),
     m_iPozitaY + (m_iBrinja / 2),
     m_iPozitaX + (m_iBrinja / 2),
     m_iPozitaY - (m_iBrinja / 2));

vija(m_iPozitaX + (m_iBrinja / 2),
     m_iPozitaY - (m_iBrinja / 2),
     m_iPozitaX - (m_iBrinja / 2),
     m_iPozitaY - (m_iBrinja / 2));

vija(m_iPozitaX - (m_iBrinja / 2),
     m_iPozitaY - (m_iBrinja / 2),
     m_iPozitaX - (m_iBrinja / 2),
     m_iPozitaY + (m_iBrinja / 2));
}

```

Funksionin Vizato për klasën Katrori më së miri do ta kuptoni nëse e analizoni figurën e mëposhtme. Pozita X dhe Y paraqet qendrën e katrorit, kurse brinja paraqet krahun e katrorit. Klasa Katrori mund të ketë konstruktor me katër pika që përfaqësojnë cepat e katrorit, mirëpo në këtë rast përdoruesi i klasës Katrori mund t'i përdorë pozitat (koordinatat) që nuk formojnë detyrimisht katrorin. Në rastin tonë, si parametër e pranojmë qendrën e katrorit si dhe gjatësinë e krahut të katrorit.



Klasën që paraqet rrethun do ta paraqesim kështu:

```

class Rrethi
{
public:
    Rrethi(int x1, int y1, int iRrezja)
        : FiguraGjeometrike(x1, y1), m_iRrezja(iRrezja) {}

    void Vizato();

private:

```

```

        int    m_iRrezja;
    };
    void Rrethi::Vizato()
    {
        rrethi(m_iPozitaX, m_iPozitaY, m_iRrezja);
    }

```

Tani, pasi i kemi definuar klasët që përfaqësojnë trekëndëshin, katrorin dhe rrethin (të cilat janë të trashëguara nga e njëjta klasë, FiguraGjeometrike), tipet e këtyre klasëve mund të trajtohen sikur një tip. P.sh. kosidero nëse e kemi të definuar klasën Monitori që përmban variablën anëtare të tipit array për mbajtjen e disa figurave gjeometrike si dhe funksionin anëtar për të shtuar figurën gjeometrike në listën e figurave gjeometrike dhe funksionin Vizato për të vizatuar të gjitha figurat gjeometrike në listën e figurave gjeometrike:

```

class Monitori
{
public:
    Monitori() { m_iFigurat = 0; }

    void Shto(FiguraGjeometrike* pFigura)
    {
        m_aFigurat[m_iFigurat++] = pFigura;
    }

    void Vizato()
    {
        for (int i = 0; i < m_iFigurat; i++)
        {
            m_aFigurat[i]->Vizato();
        }
    }

private:
    FiguraGjeometrike* m_aFigurat[1000];
    int m_iFigurat;
};

```

Klasa Monitori do të përdorej kështu:

```

void main ()
{
    Trekendeshi    t(1, 1, 2, 2, 3, 1);
    Katrori        k(4,4, 5);
    Rrethi         r(0,0, 3);

    Monitori       e;
}

```

```
e.Shto(&t);  
e.Shto(&k);  
e.Shto(&r);  
  
// vizato të gjitha figurat  
e.Vizato();  
}
```

Në strukturën `for` vizatojmë figurat e ruajtura në variablën *array*, të cilat mund të jenë të të njëjtit tip ose të tipeve të ndryshme, përderisa janë të trashëguara nga klasa *FiguraGjeometrike*.

```
for (int i = 0; i < m_iFigurat; i++)  
{  
    m_aFigurat[i]->Vizato();  
}
```

Pra, pa marrë parasysh tipin e objekteve në *array*, të gjitha objektet trajtohen njëjtë, duke thirrë funksionin *virtual vizato* të klasëve që përfaqësojnë figura gjeometrike konkrete.

Ushtrime

1. Krijë një klasë abstrakte që përmban funksionin virtual `ShtypeEmrin`. Pastaj krijë disa klasë të tjera, të trashëguara nga klasa abstrakte e definuar më parë. Të gjitha klasët e trashëguara duhet ta implementojnë funksionin `ShtypeEmrin`, i cili shtyp emrin e vetë klasës. Në programin ekzekutues definë një tregues të klasës abstrakte dhe disa variabla të klasëve të trashëguara. Inico treguesin me adresën e njëres prej variablave të pastaj thirrë funksionin `ShtypeEmrin`, p.sh.:

```
treguesi->ShtypeEmrin( );
```

Inico treguesin me adresën e variablave të tjera dhe pastaj thirrë funksionin `ShtypeEmrin` pas çdo iniciimi. Do të vëreni se i njëjti funksion shtyp emra të ndryshëm, mvarësisht nga ajo se në cilën variabël tregon treguesi i klasës abstrakte.

2. Krijë një klasë të trashëguar nga klasa `FiguraGjeometrike` për vizatimin e figurës gjeometrike pesëkëndëshe (pentagonit).
3. Krijë një klasë të trashëguar nga klasa `FiguraGjeometrike` për vizatimin e dy katrorëve, ku njëri katror është i zhvendosur në të djathtë të qendrës së katrorit tjetër. Zhvendosja të jetë e barabartë me gjatësinë e brinjës (krahut) së katrorit.

Përmbledhje

Gjuha C++ mundëson trashëgimin e një apo më shumë klase të definuar më parë (me qëllim të përdorimit të kodit të shkruar më parë), gjë që shkakton hierarkinë e klasëve të cilat kanë attribute të përbashkëta.

Hierarkia e klasëve bën thjeshtësimin konceptual të problemit të jetës së përditshme mbasi mundëson grupimin e objekteve që kanë attribute të përbashkëta.

Në gjuhën C++ treguesi i tipit të klasës më të lartë në hierarki mund të tregojë në çfarëdo tipi të klasëve në hierarki. Nëse funksionet e klasëve janë virtual, atëherë me thirrjen e funksionit në tregues të objektit të tipit të klasës në nivel të lartë të hierarkisë, thirrjm funksionin e vetë tipit të objektit të treguar e jo të tipit të treguesit. P.sh. vëreni klasët A dhe B:

```
class A
{
public:
    void virtual Funksioni_1()
    {
        cout << "Funksioni i klasës A";
    }
};

class B : public A
{
public:
    void virtual Funksioni_1()
    {
        cout << "Funksioni i klasës B";
    }
};
```

Atëherë, kodi në vijim shtyp fjalinë "Funksioni i klasës B" edhe pse tipi i treguesit është i klasës A:

```
A* pA;
B b;

pA = &b;
pA->Funksioni_1();
```

Në gjuhën C++ kjo njihet si polimorfizëm.

Manipulimi me fajll

Të gjitha programet kanë të bëjnë me manipulimin e të dhënave dhe në programet e koduara deri më tani në ushtrimet tona, këto të dhëna kanë qenë konstanta apo të dhënat të shtypura nga përdoruesi. Mirëpo, përdorimi i konstantave apo të dhënave të shtypura gjatë ekzekutimit të programit nuk janë të majftueshme. Për shembull, të marrim programin për të manipuluar me të dhënat e punëtorëve të një ndërmarrje të madhe. Po të përdornim vetëm konstantat, atëherë do të deklaranim aq konstanta sa ka punëtorë ndërmarrja, dhe sa herë që numri i punëtorëve ndërrohet, duhet të ndërrohet edhe vetë programi. Natyrisht se kjo nuk është e pëlqyeshme. Zgjidhja e dytë do të ishte që përdoruesi i programit t'i ofronte të dhënat e punëtorëve gjatë ekzekutimit të programit. Këto të dhëna do të ruheshin në memorie dinamike, dhe me përfundimin e programit, të gjitha të dhënat e ofruara nga përdoruesi do të humbnin. Edhe kjo zgjidhje nuk do të ishte e dëshirueshme, pasi që sa herë që ta ekzekutonim programin, do të duhej t'i shtypim të gjitha të dhënat prej fillimit.

Zgjidhje ideale do të ishte që t'i furnizojmë vetëm ato të dhëna që nuk janë furnizuar më parë. Pra, sa herë që furnizojmë të dhëna të reja për punëtorë, këto të dhëna duhen të ruhen në memorie të qëndrueshme (fajll). Kur paraqitet nevoja të manipulohet me këto të dhëna (p.sh. kërkimin e të dhënave të një punëtori) këto të dhëna do të lexoheshin nga fajlli ku janë ruajtur paraprakisht. Programet e jetës së përditshme i ruajnë të dhënat në fajll, kur kemi të bëjmë me numër të madh të të dhënave, përdoren programet e quajtura *database* (bazë e të dhënave) që kujdesen për të dhënat dhe bëjnë optimizimin e aksesit të këtyre të dhënave. Për të shpjeguar hollësisht mënyrën se si ruhen të dhënat në *database* dhe si bëhet manipulimi me të dhënat e ruajtura në *database* do të duhej një libër i veçantë.

Në këtë kapitull do të flasim se si ruhen të dhënat në fajll dhe si lexohen këto të dhëna të ruajtura paraprakisht.

Në gjuhën C manipulimi me fajll bëhet duke përdorë tipin `FILE` si dhe funksionet ndihmëse për ta hapur fajllin, për ta mbyllur fajllin, për t'i lexuar të dhënat prej fajllit së hapur etj.

Natyrisht se kjo metodë përdoret edhe në gjuhën C++, mirëpo gjuha C++ përmban klasat që e "mbështjellin" fajllin si dhe funksionet që kanë të bëjnë

me fajll. D.m.th. të gjitha funksionet për manipulim me fajll janë pjesë përbërëse e klasës që e definojnë fajllin.

Struktura e kodit e shkruar me përdorimin e klasëve për fajll në gjuhën C++ është më e thjeshtë dhe më e lexueshme sesa me përdorimin e metodës së gjuhës C, pra me përdorimin e tipit FILE. Mirëpo për shkak të kompletimit të metodave për të manipuluar me fajll do të japim shembull edhe me përdorimin e tipit FILE.

Për të manipuluar me fajll duhet që ta hapim fajllin, duke specifikuar njërën prej metodave të aksesit të fajllit:

- Hapja e fajllit për lexim.
- Hapja e fajllit për-ruajtjen e të dhënave (për shkrim).
- Hapja e fajllit për shkrim e lexim, dhe
- Metoda e hapjes së fajllit
 - Normale
 - Binare

Kur hapim një fajll, duhet specifikuar aksesin e hapjes së fajllit, p.sh. nëse kemi ndër mend vetëm të lexojmë nga fajlli i hapur, atëherë ai fajll duhet hapur për lexim. Nëse kemi ndër mend të shkruajmë dhe të lexojmë në fajll, atëherë ai fajll duhet hapur për lexim dhe shkrim. Duhet të keni kujdes, sepse nëse e hapni një fajll për lexim dhe tentoni të shkruani në të, atëherë programi nuk do ta kryente punën e planifikuar gjatë ekzekutimit.

Para se të shikojmë programin për manipulim me fajll, duke përdorë tipin FILE, do t'ju njoftojmë me disa funksione ndihmëse dhe qëllimin e tyre.

```
FILE *fopen( const char *filename, const char *mode );
```

Funksioni `fopen` hap fajllin e emërtuar në variablën `filename` në njërën prej metodave të paraqitura në variablën `mode`. Pra stringu `mode` tregon llojin e aksesit të fajllit dhe mund t'i ketë këto vlera:

"r" – Hapja e fajllit për lexim. Nëse fajlli i emërtuar në variablën `filename` nuk ekziston, atëherë funksioni `fopen` kthen vlerën `NULL`.

"w" – Hap një fajll të ri për shkrim. Nëse fajlli ekziston, atëherë të dhënat e fajllit ekzistuese do të humbasin dhe fajlli do të përmbajë vetëm të dhënat e ruajtura pas hapjes së fajllit.

"a" – Hapja e fajllit për shkrim në fund të fajllit ekzistues. Nëse fajlli nuk ekziston, atëherë funksioni `fopen` do ta krijojë një fajll të ri.

"r+" – Hapja e fajllit për lexim dhe shkrim. Nëse fajlli nuk ekziston, atëherë funksioni fopen kthen NULL.

"w+" – Sikurse "r+" vetëm nëse fajlli nuk ekziston, atëherë funksioni fopen krijon një fajll të ri, kurse nëse fajlli ekziston, atëherë të dhënat e fajllit do të humbasin.

Pa marrë parasysh metodën e përdorur për hapjen e fajllit, nëse paraqitet problem gjatë hapjes së fajllit, funksioni fopen kthen NULL.

Funksioni fgetc lexon një shkronjë në fajllin e hapur dhe kthen shkronjën EOF (End Of File) nëse paraqitet ndonjë gabim gjatë leximit apo nëse tentojmë të lexojmë në fund të fajllit të hapur. Prototipi i funksionit fgetc është kështu:

```
int fgetc( FILE *stream );
```

Funksioni për ruajtjen e një shkronje në fajllin e hapur është funksioni fputc, i cili merr si parametër shkronjën për ta ruajtur në fajll dhe treguesin e fajllit të hapur:

```
int fputc( int c, FILE *stream );
```

Ky funksion kthen shkronjën e ruajtur në fajll apo EOF nëse paraqitet ndonjë gabim gjatë ruajtjes së shkronjës.

Për ta lexuar më shumë se një shkronjë, mund ta përdorim funksionin fgets, i cili lexon tekstin në array të tipit char dhe prototipi i këtij funksioni është:

```
char *fgets( char *string, int n, FILE *stream );
```

Parametri i parë string është variabël ku ruhet stringu (teksti) i lexuar nga fajlli i hapur më parë (parametrin stream). Parametri i dytë n tregon maksimumin e shkronjave të përmbajtura në string për tu lexuar nga fajlli. Funksioni fgets kthen stringun e lexuar ose NULL nëse paraqitet ndonjë gabim gjatë leximit të fajlli apo tentoni të lexoni në fund të fajllit.

Funksioni i kundërt me funksionin fgets është funksioni fputs:

```
int fputs( const char *string, FILE *stream );
```

Ky funksion kthen vlerë pozitive në rast të ruajtjes së suksesshme të stringut, ose EOF nëse paraqitet ndonjë gabim.

Fajlli i hapur mbyllet me anë të funksionit `fclose` prototipi i të cilit është kështu:

```
int fclose( FILE *stream );
```

Ky funksion kthen 0 në mbylljen e fajllit pa ndonjë gabim, ose EOF nëse paraqitet ndonjë gabim.

Për ilustrimin e manipulimit të fajllit me anë të tipit `FILE` dhe funksioneve të përmendura më sipër, shihni programin që vijon. Ky program kopjon një fajll që ekziston në një fajll tjetër (sikurse komanda e sistemit operativ *Unix* - `cp` apo komanda e sistemit operativ *Windows* - `copy`).

Programi për kopjimin e fajllit:

```
#include <stdio.h>

/* Funksioni për kopjimin e të dhënave prej fajllit fpin në
   fajllin fpout */
void KopjoFajllin(FILE *fpIn, FILE *fpOut)
{
    int ch;

    /* përderisa nuk është fundi i fajllit në fpIn lexo
       nga një shkronjë në fajllin fpIn dhe ruaje në
       fajllin fpOut */

    for ( ; (ch = fgetc(fpIn)) != EOF; )
        fputc(ch, fpOut);
}

void main (int argc, char * argv[])
{
    FILE *fpin, *fpout;

    if (argc != 3)
    {
        fprintf(stderr, "Ky program përdoret kështu:\n");
        fprintf(stderr, "%s FajlliEkzistues FajlliIRi\n",
                    argv[0]);
        return;
    }

    /* tento ta hapni fajllin ekzistues për lexim*/
    if ((fpin = fopen(argv[1], "r")) == NULL)
    {
```

```

        fprintf(stderr, "Fajlli %s nuk mund të hapet",
                argv[1]);
        return;
    }

    /* tento ta hapni fajllin për kopjimin e fajllit të hapur
       më sipër */
    if ((fpout = fopen(argv[2], "w")) == NULL)
    {
        fprintf(stderr, "Fajlli %s nuk mund të hapet",
                argv[2]);
        return;
    }

    KopjoFajllin(fpin, fpout);

    /* mbyllë fajllat */
    fclose(fpin);
    fclose(fpout);
}

```

Funksioni fopen kthen NULL nëse nuk ka sukses në hapjen e fajllit, pra në kodin:

```

if ((fpout = fopen(argv[2], "w")) == NULL)
{
    fprintf(stderr, "Fajlli %s nuk mund të hapet",
            argv[2]);
    return;
}

```

shikojmë se mos ky funksion nuk ka pasur sukses në hapjen e fajllit.

Kjo ndodh nëse tentoni ta hapni për lexim fajllin që nuk ekziston. Prandaj duhet patjetër ta shikoni parametrin e kthyer, sepse nëse fopen nuk ka pasur sukses në hapjen e fajllit, atëherë funksioni i parë që tenton të manipulojë me fajll, do ta përfundonte programin me gabim.

10.1 Manipulime me fajll në C++

Ecuria e manipulimit me fajll në gjuhën C++ është e njëjtë me atë të gjuhës C, mirëpo siç përmendëm më parë, gjuha C++ përmban klasët të cilat përmbajnë funksionet për manipulim me fajll dhe deri diku lehtësojnë manipulimin me fajll. Klasët për manipulim me fajll në librarinë standarde të gjuhës C++ krijojnë një hierarki, ku secila klasë ka prejardhje prej klasës bazë ios (Input Output Stream). Këto klasë janë të ndara duke u bazuar në rrjedhjen e të dhënave (stream; lexo: *strimë*).

P.sh. klasa *ifstream* definon objektin për manipulimin e të dhënave hyrëse (leximin e të dhënave nga fajlli) dhe kjo klasë mund të përdoret vetëm për leximin e të dhënave. Klasa *ofstream* definon objektin për manipulimin e të dhënave dalëse (ruajtjen e të dhënave në fajll) dhe mund të përdoret vetëm për ruajtjen e të dhënave. Klasa *fstream* bën manipulimin e të dhënave hyrëse ose dalëse në fajll, në varësi të parametrit të definuar në funksionin anëtar për ta hapur fajllin. Pra, kjo klasë mund të përdoret për leximin e të dhënave apo ruajtjen e të dhënave.

Definimi i këtyre klasëve bëhet në fajllin header *fstream*, dhe për t'u përdorë këto klasë, duhet ta përfshini fajllin *fstream*:

```
#include <fstream.h>
```

10.1.1 Hapja e fajllit

Pasi që klasët *ifstream* dhe *ofstream* janë të përcaktuara për qëllime specifike, deklarimi i objektit të këtyre tipeve hap fajllin e definuar në konstruktorin me parametra. P.sh. kodi që vijon:

```
ifstream ifFajlli("te_dhena.dat");
```

deklaron objektin *ifFajlli* të tipit *ifstream* dhe gjithashtu hap fajllin "te_dhena.dat". Sikurse në kodin e mëparshëm ku kemi përdorë tipin *FILE* në të cilin shikonim suksesin e tentimit të hapjes së fajllit, edhe këtu duhet shikuar nëse objekti i definuar *ifFajlli* është objekt i vlefshëm (pra fajlli "te_dhena.dat" ekziston). Se a kemi pasur sukses në hapjen e fajllit mund ta kuptojmë kështu:

```
ifstream ifFajlli("te_dhena.dat");

// nëse hapja e fajllit qe e suksesshme
if (ifFajlli)
{
```

```

        // lexoji të dhënat këtu
    }

```

Klasa `ifstream` përmban edhe konstruktorin bazë (pa parametra) i cili kontrktor nuk hap asnjë fajll. Mirëpo për përdorimin e objektit të definuar me konstruktor bazë duhet ta thërrasim funksionin anëtar `open` për hapjen e fajllit, p.sh.:

```

    ifstream ifFajlli;

    ifFajlli.open("te_dhena.dat");

    // nëse hapja e fajllit qe e sukseshme
    if (ifFajlli)
    {
        // lexoji të dhënat këtu
    }

```

Klasa `ifstream` përmban operatorin anëtar `>>` (sikurse objekti `cin`) për leximin e të dhënave. P.sh. për leximin e një numri në fajllin `te_dhena.dat` do ta përdornim kodin që vijon:

```

#include <fstream.h>

void main ( )
{
    int iNumri;
    ifstream ifFajlli;

    ifFajlli.open("c:\\te_dhena.dat");

    if (ifFajlli)
    {
        ifFajlli >> iNumri;
    }
}

```

Pasi që klasa `ifstream` mund të përdoret vetëm për leximin e të dhënave, kjo klasë nuk e përmban operatorin anëtar `<<` (sikurse objekti `cout`), dhe nëse tentoni ta përdorni këtë operator kompajleri do të paraqesë gabim.

Klasa `ofstream` është e ngjashme me klasën `ifstream`, mirëpo kjo klasë përdoret për ruajtjen e të dhënave në fajll. Klasa `ofstream` e përmban operatorin `<<` (sikurse objekti `cout`) për shtypjen e të dhënave, mirëpo në vend që këto të dhëna të shtypen në monitor, ato ruhen në fajllin e hapur, p.sh.:

```
#include <fstream.h>
```

```
void main ( )
```

```
{
    int iNumri = 5;
    ofstream ofFajlli("c:\\te_dhena.txt");

    if (ofFajlli)
    {
        ofFajlli << "Kemi shtypur variablën "
        << "iNumri me vlerë: "
        << iNumri;
    }
}
```

Klasa `fstream` mund të përdoret për leximin apo për ruajtjen e të dhënave në fajll. Se për cilën metodë mund të përdoret kjo klasë, mvaret nga parametrat në konstruktor apo në funksionin `open`. Funksioni anëtar i klasës `fstream` ka këta parametra:

```
void open( const char* szName, int nMode,
           int nProt = filebuf::openprot );
```

Parametri `szName` është emri i fajllit për t'u hapur, kurse parametri `nMode` është loji i hapjes së fajllit (p.sh. për leximin e të dhënave apo për ruajtjen e të dhënave) si dhe metodat e tjera (shih tabelën që vijon).

Metoda	Përshkrimi
<code>ios::in</code>	Hapja e fajllit për leximin e të dhënave. (sikurse "r" në funksionin <code>fopen</code>)
<code>ios::out</code>	Hapja e fajllit për ruajtjen e të dhënave. (sikurse "a" në funksionin <code>fopen</code>)
<code>ios::app</code>	Hapja e fajllit për të shkruar të dhëna në fund të fajllit ekzistues. (sikurse "a" në funksionin <code>fopen</code>)
<code>ios::ate</code>	Pasi të hapet fajlli, treguesi për fajll shkon në fund të fajllit.
<code>ios::binary</code>	Hapja e fajllit me metodën binare.
<code>ios::trunc</code>	Nëse fajlli ekziston, të dhënat e saj fshihen me hapjen e fajllit.
<code>ios::nocreate</code>	Ky parametër bën që të mos krijohet ndonjë fajll i ri, nëse tentoni ta hapni fajllin që nuk ekziston.
<code>ios::noreplace</code>	Nëse tentoni ta hapni fajllin për

ruajtjen e të dhënave dhe fajlli ekziston, atëherë hapja e fajllit paraqet gabim.

P.sh. për hapjen e fajllit për ruajtjen e të dhënave me anë të klasës `fstream` do ta përdornim këtë kod:

```
fstream fFajlli;
fFajlli.open("libri.txt", ios::out);
```

Tani objekti `fFajlli` mund të përdoret sikurse të ishte i tipit `ofstream`. Kod i mësipërm mund të shkruhet shkurtimisht duke përdorë konstruktorin me parametra të klasës `fstream`, pra:

```
fstream fFajlli("libri.txt", ios::out);
```

10.1.2 Hapja e fajllit për lexim dhe shkrim

Klasa `fstream` implementon të dy operatorët e përmendur më sipër operatorin `>>` për leximin e të dhënave dhe ruajtjen e tyre në variabla, si dhe operatorin `<<` për ruajtjen e vlerave të variablave ose të konstantave në fajllin e hapur.

Shembulli në vijim lexon fjalinë e shkruar nga përdoruesi, përdoruesi nuk e shtyp tastierën 'Enter'. Fjala e lexuar i bashkangjitet fjalive ruajtura më parë në fajllin `Fjalite.txt`, p.sh.:

```
#include <fstream.h>

const int gjatesia_e_fjalise = 255;

void main ( )
{
    char        szFjalia[gjatesia_e_fjalise + 1];
    fstream     fFajlli("c:\\Fjalite.txt", ios::app);

    if (!fFajlli)
    {
        // hapja e fajllit nuk qe e suksesshme
        return;
    }

    cout << "Shtype një fjali : " ;

    // lexo fjalinë prej hyrjes standarde
    // të të dhënave (tastatura)
    cin.getline(szFjalia, gjatesia_e_fjalise);
```



```
// ruaja fjaline në fajllin e hapur
fFajlli << szFjalia << endl;

fFajlli.close();
}
```

Funksioni `getline` i përdorur në program për leximin e fjalisë ka këta parametra:

```
istream& getline( char* pch, int nCount, char delim = '\n' );
```

Parametri i parë është variabla *array* në të cilën ruhen të dhënat e lexuara, parametri *nCount* tregon cili është numri maksimal i shkronjave ose simboleve që mund të lexohen nga hyrja standarde e të dhënave, si dhe parametri *delim* tregon se cila shkronjë e përfundon leximin e të dhënave (ku vlerë bazë është shkronja Enter - `'\n'`).

Nëse shtypim më shumë shkronja se që kemi specifikuar në parametrin e dytë të funksionit `getline`, atëherë variabla *szFjalia* do të mbajë vetëm aq simbole (duke pasur parasysh edhe simbolin përfundues të stringut në *array*, simbolin 0) sa është specifikuar në parametrin e dytë (gjatesia e *fjalise*). Në kapitujt e mëparshëm kemi shkruar programin për numërimin e shkronjave në fjali. Këtu do ta ndërrojmë programin që ta numërojë numrin e shkronjave të paraqitura në fajllin e specifikuar në komandën e programit. Numrin e çdo shkronje do ta ruajmë në tipin *array* alfabeti dhe pasi ta përfundojmë leximin e fajllit të dhënë, do të hapim një fajll tjetër për të ruajtur rezultatin e çdo shkronje.

P.sh.:

```
#include <fstream.h>

const int numri_i_germave = 26;

int main (int argc, char* argv[])
{
    int alfabeti[numri_i_germave];
    int ch, i;

    // nese nuk e kemi dhene emrin e fajllit
    // per t'u hapur, dil nga programi
    if (argc < 2)
        return 1;

    // inico te gjithë anëtarët e tipit array
    for (i = 0; i < numri_i_germave; i++)
        alfabeti[i] = 0;
```

```

fstream fFajlli(argv[1], ios::in);

if (!fFajlli)
    return 1;

while(!fFajlli.eof())
{
    ch = fFajlli.get();

    if ((ch > 64) && (ch < 91))        // germë e madhe
        alfabeti[ch-65]++;
    else if ((ch > 96) && (ch < 123)) // germë e vogël
        alfabeti[ch-97] ++;
}

// hape fajllin për ta ruajtur rezultatin
fstream fRezultati("c:\\rezultati.txt", ios::out);

if (!fRezultati)
    return 1;

// shtype numrin e çdo germe të paraqitur
// në fajllin e hapur për lexim

for ( i = 0; i < numri_i_germave; i++)
    fRezultati << "Germa " << char (i + 97)
                << " është paraqitur "
                << alfabeti[i] << " here\n";

return 0;
}

```

Në rreshtin:

```
fRezultati << "Germa " << char (i + 97)
```

kemi përdorë operatorin `cast char (i + 97)` për arsye se nëse do të shkruanim vetëm `i + 97`, atëherë objekti `fRezultati` do ta ruante numrin `i + 97` si integjer, p.sh. numrin 115. Mirëpo, nëse përdorim operatorin `cast`, në vend se ta ruajë numrin 115, e ruan shkronjën 's'.

Pasi që klasa `fstream` ka prejardhje nga klasa `ios`, objektet e tipit `fstream` mund t'i formatojnë të dhënat për t'u ruajtur në fajll duke përdorë funksionet ndihmëse të definuara në fajllin e librisë standarde `<iomanip.h>`. P.sh. kodi që vijon shtyp numrat decimalë prej 1 deri më 100 si dhe vlerat e tyre përkatëse në hex.

```
#include <fstream.h>
```

```
#include <iomanip.h>

int main (int argc, char* argv[])
{
    // hape fajllin për ruajtjen e rezultatit
    fstream fRezultati("c:\\rezultati.txt", ios::out);

    if (!fRezultati)
        return 1;

    // shtypi numrat në vlera decimale dhe heksadecimale
    for ( int i = 1; i <= 100; i++)
    {
        fRezultati << "Numri decimal " << setw(6) << i
                    << "në heksadecimal është baras me "
                    << setiosflags(ios::hex)
                    << i << endl;
        fRezultati << resetiosflags(ios::hex);
    }

    return 0;
}
```

Në kodin e mësipërm kemi përdorë funksionin ndihmës `setw(6)` për të caktuar se numri i shtypur më vonë do të zëjë 6 hapësira edhe pse nuk është numër 6-shifror. Pastaj kemi përdorë funksionin `setiosflags(ios::hex)` për ta caktuar formatin e numrit për t' shtypur (pra për shtypjen e numrave decimalë në hex).

Rreshti që vijon e kthen formatin e numrave në format të mëparshëm. Po që se nuk do ta kthenim formatin e numrave të mëparshëm, atëherë të gjithë numrat do të shtypeshin si heksadecimal.

```
fRezultati << resetiosflags(ios::hex);
```

Rezultati i programit të mësipërm do të dukej kështu:

<i>Numri decimal</i>	<i>1 në heksadecimal është baras me 1</i>
<i>Numri decimal</i>	<i>2 në heksadecimal është baras me 2</i>
<i>Numri decimal</i>	<i>3 në heksadecimal është baras me 3</i>
<i>Numri decimal</i>	<i>4 në heksadecimal është baras me 4</i>
:	
<i>Numri decimal</i>	<i>92 në heksadecimal është baras me 5c</i>
<i>Numri decimal</i>	<i>93 në heksadecimal është baras me 5d</i>
<i>Numri decimal</i>	<i>94 në heksadecimal është baras me 5e</i>
<i>Numri decimal</i>	<i>95 në heksadecimal është baras me 5f</i>
<i>Numri decimal</i>	<i>96 në heksadecimal është baras me 60</i>

Numri decimal	97 në heksadecimal është baras me 61
Numri decimal	98 në heksadecimal është baras me 62
Numri decimal	99 në heksadecimal është baras me 63
Numri decimal	100 në heksadecimal është baras me 64

10.1.3 Përdorimi i fajllit pa renditje

Deri më tani kemi lexuar apo shkruar të dhënat në fajlla në renditje (p.sh. kemi lexuar shkronjat një pas një). Fakti se shkronjat janë të ruajtura në fajll në renditje (një pas një) nuk do të thotë se ato duhen lexuar ashtu. Shpeshherë kemi nevojë të lexojmë një apo disa shkronja në fajll e në bazë të të dhënave të lexuara në fajll (osë të të dhënave të tjera) të lexojmë diku tjetër në fajll, do të thotë të kapërcejmë të dhënat e ruajtura në mes të dy pjesëve të fajllit. Kjo mënyrë e leximit të fajllit përdoret shumë në ato *database* në të cilat fajllat kanë informata masive dhe leximi në renditje do ta ngadalësonte kërkimin e të dhënave të caktuara.

Për ta lejuar këtë mënyrë të përdorimit të fajllit, çdo objekt i tipit `fstream` shënon pozitën e fajllit gjatë leximit apo ruajtjes së të dhënave në fajll. Kjo pozitë ruhet në variabël të tipit `long` dhe paraqet pozitën relative nga fillimi i fajllit. Kur hapim fajllin, pozita e fajllit do të jetë zero dhe çdo lexim apo ruajtje e të dhënave e ndryshon pozitën e shënuesit. Në fajll të hapur çdo herë mund ta shikoni pozitën e shënuesit (tari e tutje: pika) me anë të funksionit anëtar `tellg` (për fajllin e hapur për hyrjen e të dhënave) dhe me anë të funksionit `tellp` (për fajllin e hapur për daljen-ruajtjen e të dhënave).

P.sh. pas hapjes së fajllit "rezultati.txt" funksioni `tellg` do të kthejë 0 (fillimi i fajllit), pra variabla `lPozita` do ta ketë vlerën 0:

```
fstream fRezultati("c:\\rezultati.txt", ios::in);
long lPozita = fRezultati.tellg();
```

Nëse e lexojmë një shkronjë dhe prapë e shikojmë pozitën e pikës në fajll duke e ruajtur në variablën `lPozita`, atëherë variabla `lPozita` do ta ketë vlerën 1, p.sh.:

```
fstream fRezultati("c:\\rezultati.txt", ios::in);

long lPozita = fRezultati.tellg();

fRezultati.get();

lPozita = fRezultati.tellg();

cout << lPozita;
```

Nëse dëshirojmë ta ndërrojmë pozitën e pikës shënuese në fajll të hapur (pa lexuar apo shkruar të dhëna), atëherë mund ta përdorim funksionin anëtar *seekg* (për fajllin e hapur për hyrje të të dhënave) dhe *seekp* (për fajllin e hapur për daljen-ruajtjen e të dhënave).

Prototipi i këtyre funksioneve është si vijon:

```
ostream& seekg( streamoff off, ios::seek_dir dir );
ostream& seekp( streamoff off, ios::seek_dir dir );
```

Tipi i parametrut të parë *streamoff* është *typedef* i tipit *long* (emërtim i tipit *long*), kurse *ios::seek_dir* është tip i numruar (*enum*).

Parametri i dytë mund të ketë tri vlera:

- *ios::beg* Tregon se pozita relative duhet të llogaritet nga fillimi i fajllit.
- *ios::cur* Tregon se pozita relative duhet të llogaritet nga pozita e tanishme e fajllit.
- *ios::end* Tregon se pozita relative duhet të llogaritet nga mbarimi i fajllit.

Pra funksionet *seekg* dhe *seekp* marrin si parametër pozitën relative dhe udhëzimin se nga cila pozitë duhet të llogaritet pozita relative.

Nëse parametri i parë e ka vlerën pozitive, atëherë pozita e pikës në fajll zhvendoset përpara, kurse nëse vlera e parametrut të parë është negative, atëherë pozita zhvendoset mbrapa.

Si shembull se si mund të zhvendosim pozitën e pikës në fajll-shihni kodin që vijon:

```
seekg(11L, ios::beg);    // zhvendose pozitën në shkronjën
                        // e 12 të fajllit të hapur për
                        // leximin e të dhënave

seekp(11L, ios::beg);    // zhvendose pozitën në shkronjën
                        // e 12 të fajllit të hapur për
                        // ruajtjen e të dhënave

seekg(11L, ios::cur);    // zhvendose pozitën relativisht 12
                        // shkronja përpara në fajllin e
                        // hapur për hyrjen e të dhënave

seekp(11L, ios::cur);    // zhvendose pozitën relativisht 12
                        // shkronja përpara në fajllin e
                        // hapur për dalje-ruajtjen e të
```

```

// dhënavë

seekg(-11L, ios::cur); // zhvendose pozitën relativisht 12
                        // shkronja mbrapa në fajllin e hapur
                        // për hyrjen e të dhënave

seekp(-11L, ios::cur); // zhvendose pozitën relativisht 12
                        // shkronja mbrapa në fajllin e hapur
                        // për dalje-ruajtjen e të dhënave

seekg(0L, ios::beg);    // shko në fillim të fajllit
seekp(0L, ios::beg);    // shko në fillim të fajllit

seekg(0L, ios::end);    // shko në mbarim të fajllit
seekp(0L, ios::end);    // shko në mbarim të fajllit

```

Pasi që fillojmë të numërojmë nga 0, atëherë vlera e parametrit të parë X do të thotë ta zhvendosim pozitën për X+1.

Libraritë standarde përmbajnë funksione për gjetjen e madhësisë së fajllit, mirëpo këtu, për ta ilustruar funksionin seekg (apo seekp për fajll për ruajtjen e të dhënave) e kemi përdorë këtë funksion për gjetjen e madhësisë së fajllit, p.sh.:

```

#include <fstream.h>

int main (int argc, char* argv[])
{
    if (argc < 2)
    {
        cout << "Në këtë program duhet ta emërtoni fajllin."
              << "për ta gjetur madhësinë." ;
        return 1;
    }

    fstream fRezultati(argv[1], ios::in);

    // shko në fund të fajllit
    fRezultati.seekg(0L, ios::end);

    // shih pozitën
    long lMadhesia = fRezultati.tellg();

    cout << "Madhësia e fajllit " << argv[1]
          << " është " << lMadhesia << " bajts. ";

    return 0;
}

```



```

        gjatesia_e_kolones_se_pemeve);

    if (strcmp("mollat      ", szKolona) == 0 )
    {
        // + gjatesia e kolones ne te djathte
        lPozitaECmimit = fPemet.tellg() + 10;
    }

    if (strcmp("bananet      ", szKolona) == 0 )
    {
        // shko në kolonën e çmimeve të bananeve
        fPemet.seekg(fPemet.tellg() + 10, ios::beg);
        fPemet.getline(szKolona, 7);
        // shndërro numrin e ruajtur prej tekstit në
        // integer
        iCmimiIBananeve = atoi(szKolona);
        break;
    }

    lPozita += gjatesia_e_rreshtit + 1;
}

fPemet.close();

// hape fajllin për të ruajtur të dhënat
fPemet.open("c:\\\\Pemet.dat", ios::out|ios::ate);

// shko në pozitën e ruajtur më parë
fPemet.seekp(lPozitaECmimit-1, ios::beg);

sprintf(szKolona, "%4d Lekë", iCmimiIBananeve * 2);

fPemet.write(szKolona, 10);

return 0;
}

```

Në programin e mësipërm së pari e kemi hapur fajllin për leximin e të dhënave dhe kërkimin e pozitës për kolonën e mollave. Pas gjetjes së pozitës së mollave e kemi ruajtur pozitën e kolonës për çmimin e mollave (në variablën lPozitaECmimit) për ta përdorë më vonë. Pastaj kemi kërkuar pozitën e bananeve dhe e kemi lexuar çmimin e tyre. Në këtë rast e kemi përdorë funksionin ndihmës atoi për shndërrimin e tekstit në integjer. Me këtë rast kemi dalë nga vorbulla përsëritëse dhe e kemi mbyllur fajllin e hapur për lexim.

Hapi i dytë në këtë program është hapi e fajllit për ruajtjen e të dhënave të lexuara më parë. Keni parasysh se këtu kemi hapur fajllin me parametrin ios::out|ios::ate për t'i ikur problemit për zëvendësimin e fajllit ekzistues

(pra përdorimin e parametrës `ios::out`). Pas hapjes së fajllit për ruajtjen e të dhënave, e kemi zhvendosur pozitën e pikës në pozitën e ruajtur më parë gjatë leximit të fajllit dhe me këtë rast e kemi ndërruar çmimin e mollave në dyfishin e çmimit të bananeve.

Duhet ta keni parasysh se ky program është vetëm për të ilustruar përdorimin e fajllit pa renditje dhe nuk është shembull i mirë për kërkimin e të dhënave në fajll apo për zëvendësimin e të dhënave.

10.1.4 Manipulime me fajll në metodën binare

Të dhënat në fajll ruhen në dy formate:

- ♦ Formati i formatuar për lexim (tekst), dhe
- ♦ Formati binar

Deri më tani e kemi përdorë formatin *tekst* dhe për këtë arsye të dhënat e ruajtura në fajll mund të lexohen edhe me programet e tjera për leximin e fajllave tekst (p.sh. me anë të programit Notepad).

Fajllat me format *binar* zakonisht përdoren për programe ekzekutuese (p.sh. për fajllat me prapashtesën *.exe*), për libraritë e sistemit operativ si dhe fajll të tjerë në të cilat përmbajtja e të dhënave në fajll në formë teksti nuk ka kuptim. Me përdorimin e formatit *tekst*, të dhënat gjatë ruajtjes mund ta ndërrojnë përmbajtjen e shkronjave speciale të mvarura nga sistemi operativ, kurse përmbajtja e të dhënave në format *binar* nuk mvaret nga sistemi operativ.

Për hapjen e fajllit në format *binar* duhet ta përdorni këtë konstantë:

```
ios::binary
```

p.sh.:

```
fstream fdata("c:\\te_dhena.dat", ios::out|ios::binary);
```

Fajllat e ruajtura në format binar duhet të hapen dhe të lexohen në të njëjtin format. Përndryshe, nëse e përdorni një format për ruajtjen e të dhënave dhe një format tjetër për leximin e të dhënave, atëherë përmbajtja e të dhënave mund të jetë e ndryshme (jo e rregullt).

10.2 Hierarkia e klasëve ios

Manipulimet me fajll në gjuhët C dhe C++ bëhen të mundshme nëpërmjet librariëve standarde. Në gjuhën C++ libreria standarde për manipulim me fajll është libreria *iostream* që përmban hierarkinë e klasëve për manipulim me fajll. Të gjitha klasët për manipulim me fajll (shtypjen dhe leximin e të dhënave) kanë prejardhje direkte apo indirekte nga klasa *ios* (*ios* është akronim i *Input Output Stream*). Edhe pse klasa *ios* nuk është klasë abstrakte, nuk është e arsyeshme ta deklaroni ndonjë objekt të tipit *ios* apo të krijoni ndonjë klasë të trashëguar direkt nga klasa *ios*. Mirëpo, klasët e tjera në nivele më të ulta të hierarkisë mund të përdoren për shtypjen e të dhënave apo leximin e të dhënave nga objekte të ndryshme. Për këtë arsye, libraritë standarde për gjuhën C++ përmbajnë klasë të ndryshme për shtypjen dhe leximin e të dhënave në objekte të ndryshme. T'ju kujtojmë objektet e ndryshme të tipit *iostream* siç janë: *cin* i cili përdoret për leximin e të dhënave nga hyrja e të dhënave standarde (tastatura); *cout* i cili përdoret për shtypjen e të dhënave në daljen e të dhënave standarde (monitor) dhe *cerr* për shtypjen e të dhënave në daljen standarde të gabimeve (zakonisht monitori).

Edhe pse klasa *ios* nuk përdoret direkt, shumë funksione anëtare apo variabla anëtare të trashëguara nga kjo klasë përdoren nëpërmjet klasëve të tjera më specifike. Siç kemi përmendur më parë, për hapjen e fajllit kemi përdorur variablat statike anëtare të klasës *ios*, p.sh. parametri *ios::in*:

```
fstream fPemet("c:\\Pemet.dat", ios::in);
```

Të gjitha klasët në libarinë *iostream* (në hierarkinë e klasës *ios*) janë të mvarura nga klasa *streambuf* për procesin e hyrjes dhe daljes së të dhënave. Kjo klasë është klasë abstrakte, mirëpo libreria *iostream* përmban klasë të trashëguara nga klasa *streambuf* për t'u përdorë për manipulim me fajll, p.sh.:

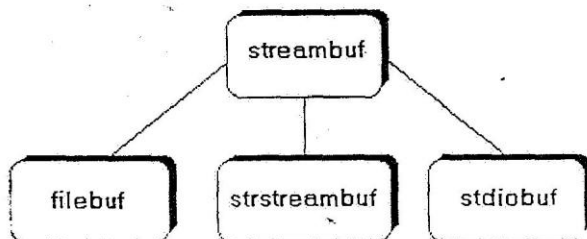
- *filebuf* – klasa për memorizimin e të dhënave hyrëse dhe dalëse për fajllat në disk.
- *strstreambuf* – klasa për memorizimin e të dhënave në bajt *array*.
- *stdiobuf* – klasa për manipulimin e të dhënave të memorizuara nga nivelet më të ulta për hyrjen dhe daljen e të dhënave në fajll.

Klasët në hierarkinë e klasës *ios* furnizojnë me funksione anëtare për manipulimin e të dhënave në fajll dhe formatimin e këtyre të dhënave, mirëpo

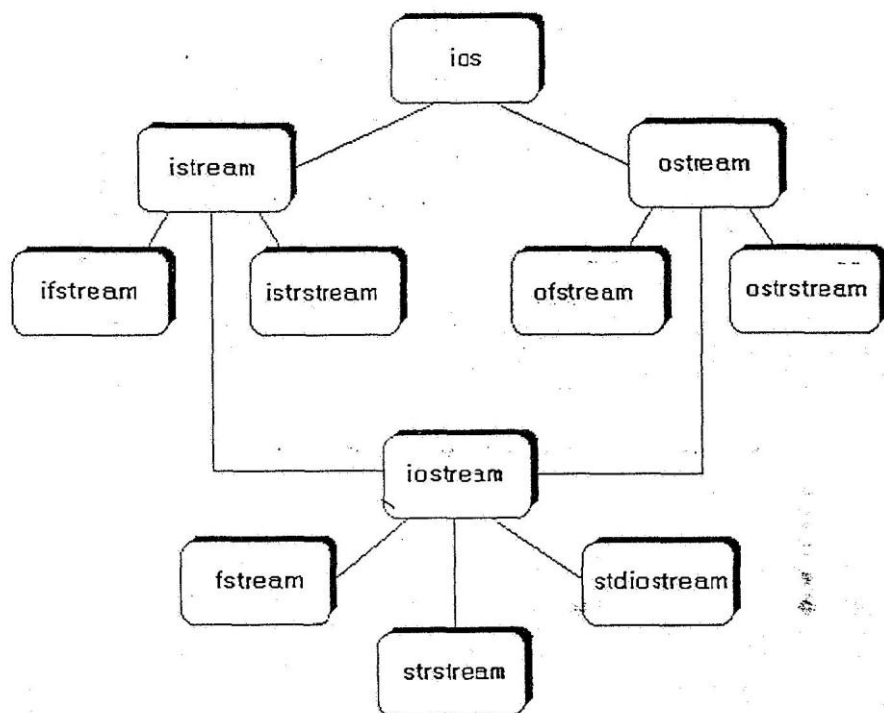
klasët e trashëguara nga klasa `streambuf` kryejnë funksionin kryesor. Funksionet e klasëve `ios` i thërrasin funksionet `virtual` të klasës `streambuf` prej të cilave disa kanë implementimin bazë.

Për të përdorë pjesë të caktuara të memories fizike, apo *window* (sikurse të sistemit operativ *Windows*), ose ndonjë aparat të ri, do të duhet të trashëgoni klasë nga klasa `streambuf` dhe t'ia përshtatni sistemit përkatës. Mirëpo, kjo është vetëm për nivele të ulta të programimit dhe në të shumtën e rasteve nuk do të keni nevojë të manipuloni me klasën `streambuf`.

Për kompletim kemi paraqitur hierarkinë e klasëve të trashëguara nga klasa `streambuf` në librarinë standarde `iostream`:



Kurse, sa i përket hierarkisë së klasëve të trashëguara nga klasa `ios`, shikoni figurën që vijon:



Hierarkia e klasës `ios`.

Ushtrime

1. Shkruajeni programin për numrimin e zanoreve në fajllin e dhënë. Shikojeni programin në këtë kapitull për leximin e të gjitha shkronjave të alfabetit anglez.
2. Shkruajeni programin për numrimin e bashkëtingëlloreve në një fajll të dhënë.
3. Shkruajeni programin për kopjimin e një fajll ekzistues në një fajll tjetër, duke përdorë klasët *ifstream* dhe *ofstream*. Shikoni programin e shkruar me tipin *FILE* dhe funksionet ndihmëse. Krahasoni programin e shkruar me klasët *ifstream* dhe *ofstream* me atë të shkruar me tipin *FILE*.
4. Shkruajeni një program që lexon të dhënat e shtypura nga përdoruesi në hyrjen standarde (tastaturë) dhe ruani këto të dhëna në një fajll. Këto të dhëna të lexohen derisa përdoruesi të shtypë fjalën "fund".
5. Shkruajeni programin për gjetjen e gjatësisë së fajllit për daljen-ruajtjen e të dhënave. Përdorni funksionet anëtare të cilat kanë të bëjnë me fajllin për daljen e të dhënave.
6. Shkruajeni programin që lexon çdo të pestën shkronjë në fajllin e dhënë si parametër në program. Sgërtytëzoni metodën e përdorimit të fajllit pa renditje.
7. Ndryshojeni programin në 10.1.3, i cili ndryshon çmimin e mollave në fajllin "pemet.dat". Ndryshoni çmimin e dredhëzave, ashtu që të jetë tri herë më i lartë sesa çmimi i bananeve.
8. Testoni hapjen e fajllit me metodën normale dhe binare. Përdorëni programin e shkruar në ushtrimin 3 dhe tentoni t'i kopjoni fajllat ekzekutuese (fajllat me prapashitesën *exe*). Pastaj ndërrojeni programin e ushtrimit 3 që ta hapë fajllin me metodën binare. Tentoni të ekzekutoni fajllat e kopjuara me të dy metodat e hapjes së fajllave.

Përmbledhje

Në të shumtën e rasteve, të dhënat e përdorura në program duhet të ruhen në memorie të qëndrueshme. Nëse sasia e të dhënave është relativisht e vogël apo struktura e të dhënave është e thjeshtë, atëherë këto të dhëna mund të ruhen në fajll, përndryshe duhen përdorë programet e quajtura *database*.

Në gjuhën C++ manipulimi me fajll bëhet në metodën e përdorur në gjuhën C (pra me përdorimin e tipit `FILE`) apo me anë të klasëve të furnizuara nga libraria standarde `iostream`. Klasa `ifstream` përdoret për leximin e të dhënave nga fajlli, klasa `ofstream` përdoret për ruajtjen e të dhënave në fajll, kurse klasa `fstream` mund të përdoret për leximin apo ruajtjen e të dhënave në fajll.


```

}

int minimumi(long a, long b)
{
    return a < b ? a : b;
}

float minimumi(float a, float b)
{
    return a < b ? a : b;
}
...

```

Me përdorimin e shabllonit, kodi i mësipërm do të shkruhej kështu:

```

<template class T>

T minimumi(T a, T b)
{
    return a < b ? a : b;
}

```

Pra siç e vëreni në kodin e mësipërm, shabllonet krijohen me përdorimin e udhëzimit `template`, që rrjedh nga gjuha angleze `template` (lexo: `templlejt` = shabllon).

Funksioni shabllon `minimumi` mund të përdoret për të gjitha tipet, të cilat përmbajnë operatorin `<`. D.m.th. përpos tipeve që përdorëm në funksionet ngarkuese më sipër, mund të përdoret edhe për tipet `char`, `double` etj. si dhe tipet e krijuara nga përdoruesit që përmbajnë operatorin `<`.

Me përdorimin e shablloneve nuk keni nevojë ta përsëritni të njëjtin kod për tipe të ndryshme, kështu që i ikni edhe gabimeve të mundshme gjatë përsëritjes së kodit. Kjo mundëson të koncentrohemi në një pjesë më të vogël të kodit, prandaj të bëni më pak gabime dhe më lehtë t'i gjeni gabimet gjatë ekzekutimit të programit.

Përpos përparësisë së krijimit të funksioneve gjenerike, shabllonet mundësojnë krijimin e tipeve abstrakte gjenerike për të cilat do të flasim në temën që vijon.

11.1 Tipet abstrakte gjenerike

Deri më tani kemi përdorur tipin *array* për të mbajtur më shumë se një vlerë. Problemi me këtë tip është se duhet ditur numrin e elementeve të përdorura qysh gjatë shkrimit të kodit, një gjë që shpeshherë është pothuajse e pamundshme. Numri i elementeve të përdorura shpeshherë dihet vetëm gjatë ekzekutimit të programit. Në disa zgjidhje, përdoruesit kanë përdorë tipin *array* me numër shumë të madh të elementeve, kështu që kodi ka qenë "i përshtatshëm" edhe në raste kur numri i elementeve të ruajtura në *array* është rritur. Po themi "i përshtatshëm" pasi që kjo zgjidhje nuk është ideale për arsye se në rastet kur kemi numër të vogël të elementeve të ruajtura në *array* do të kemi memorie të rezervuar pa nevojë për tipin *array*. D.m.th programi do të përdorë sasi të madhe të memories pa nevojë. Problemi tjetër është se edhe pse kemi deklaruar tipin *array* me numër të madh të elementeve, shpeshherë nuk është e mundur ta parashikojmë numrin maksimal të elementeve për t'u ruajtur në *array*, d.m.th. prapë e limitojmë zgjidhjen e problemit.

Në kapitujt e mëparshëm kemi përmendur listën lidhëse për zgjidhjen e problemeve, kur numri i elementeve për t'u ruajtur (dhe manipuluar) nuk dihet, përderisa programi të mos ekzekutohet. Me përdorimin e listës lidhëse do të duhet të shkruhej i njëjti kod për të gjitha tipet e nevojshme për t'u ruajtur në bashkësi, për shembull për tipin *char*, *int*, *float* etj.

P.sh. pasi që kemi shkruar listën lidhëse për tipin *int* dhe e kemi testuar se punon si duhet, do të ishte shumë lehtë të shkruhej lista lidhëse për tipin *char* apo ndonjë tip tjetër, duke kopjuar kodin për tipin *int* dhe duke zëvendësuar tipin *int* me tipin *char*. Mirëpo problemi me kopjimin e kodit qëndron se nëse keni nevojë ta ndryshoni një pjesë të kodit p.sh. për shkak të gabimeve ose për shkak të optimizimit të ekzekutimit të kodit etj., atëherë do të duhej të ndryshohej kodi i përsëritur në tërë projektin. Në të shumtën e rasteve do të harronit ta bëni përmirësimin e kodit për tipet e tjera dhe kjo do të ndikonte në prishjen e qëndrueshmërisë së kodit si tërësi.

Zgjidhja e këtij problemi do të ishte përdorimi i shablloneve në krijimin e tipeve abstrakte. Në këtë kapitull do ta krijojmë tipin vektor që do ta simulojë tipin *array*, pra do ta mundësojë mbajtjen e më shumë se një elementi të tipeve primitive apo tipeve të shkruara nga vetë përdoruesit.

11.1.1 Tipi Vektori

Përparësia e tipit vektori ndaj tipit *array* është se tipi vektori mundëson ruajtjen e numrit të ndryshëm të elementeve gjatë ekzekutimit të programit. Tipi vektori që do ta krijojmë në këtë kapitull, do ta ruajë numrin e elementeve të ruajtura në objekt si dhe vetë të dhënat për çdo element. Për shembull, nëse tipi vektori ruan të dhënat e tipit *int*, atëherë do të duhej ta rezervojmë memorien për ruajtjen e elementeve *integer* si dhe deklarimin e një variable anëtare të klasës vektori, për t'i numëruar elementet *integer* të ruajtura në objektin e tipit vektori.

Në tipin vektori do të mundësojmë që në mënyrë dinamike të shtohen anëtarë të rinj (përkundër tipit *array* ku numri i anëtarëve është fiks). Sa për ilustrim anëtarët e objektit vektori do të ruhen në listë lidhëse. Sa herë që shtojmë ndonjë anëtar të ri në objekt të tipit vektori, do ta rezervojmë memorien e duhur për atë anëtar.

Sa për fillim, klasën vektori do ta definojmë me disa funksione elementare për manipulimin e elementeve në memorien e rezervuar, kthimin e numrit të anëtarëve në klasën vektori etj.:

```
class Vektori
{
public:
    Vektori();
    virtual ~Vektori();

public:
    void Shto(int iAnetari);
    void Fshije(int iIndeksi);
    int NumriIAnetareve();

private:
    struct elementi
    {
        int iElementi;
        struct elementi* tjetri;
    } *m_pElementet;

    int m_iAnetaret;
};
```

Siç e vëreni, brenda në klasën vektori kemi deklaruar një tip të ri (*elementi*) për t'i ruajtur të dhënat e një elementi, si dhe treguesin në elementin tjetër në listën lidhëse. Duhet ta kuptoni se kemi përdorë këtë metodë për të ilustruar shabllonet dhe nuk rekomandohet të krijoni një klasë si p.sh. klasa vektori me anë të listës lidhëse. Kjo metodë nuk është aspak efikase.

Zakonisht kompajlerët modernë ofrojnë klasë sikurse vektori që janë mjaft efikase, duke përdorë metoda të ndryshme për ruajtjen e elementeve në bashkësi. Shikoni klasët `vector`, `map`, `list`, `set` etj.

Funksioni `Shto`, anëtar i klasës vektori, merr si parametër vlerën për t'u ruajtur në objekt. Ky funksion rezervon memorien e duhur për element dhe ia bashkangjet listës së elementeve të tjera në objekt. Nëse elementi i shtuar është elementi i parë, atëherë ky element do të jetë në fillim të listës së elementeve në objekt dhe treguesi `*tjetri` do të tregojë në `NULL`. Implementimi i funksionit `Shto` do të jetë kështu:

```
void Vektori::Shto(int iAnetari)
{
    // nëse në fillim të listës
    if (m_pElementet == NULL)
    {
        m_pElementet = new elementi;

        m_pElementet->iElementi    = iAnetari;
        m_pElementet->tjetri        = NULL;

        m_iAnetaret                = 1;
    }
    else
    {
        // shto elementin e ri ne fund te listes

        elementi* p = m_pElementet;
        while (p->tjetri != NULL)
            p = p->tjetri;

        // rezervu memorien e duhur për element të ri
        elementi* pIri = new elementi;

        pIri->iElementi    = iAnetari;
        pIri->tjetri        = NULL;

        // bashkangjite elementin e ri në fund të listës
        p->tjetri          = pIri;
        m_iAnetaret++;
    }
}
```

Siç përmendëm më parë, kjo metodë është joefikase për arsye se kur ta shtojmë ndonjë element të ri, duhet kërkuar fundin e listës për t'ia bashkangjitur atë element. Paramendoni nëse lista përmban numër shumë të madh të elementeve, sa i ngadalshëm do të ishte ky operacion. Në anën tjetër, përparësia e klasës vektori ndaj klasës `array` do të ishte se klasa vektori mund të mbajë numër të ndryshëm të elementeve dhe të përdorë memorie

(edhe pse përdor më shumë memorie për element në krahasim me tipin *array*) vetëm atëherë që është e nevojitet gjatë ekzekutimit të programit.

Funksioni *Fshije* ka për detyrë të bëjë fshirjen e elementit në objekt me indeks, sikurse parametri i këtij funksioni (pra numri i elementit në renditje të listës). Nëse përdoruesi i klasës vektori furnizon parametër të gabuar në këtë funksion anëtar (për shembull objekti përmban 5 elemente, kurse përdoruesi tenton ta fshijë elementin e 10-të), atëherë duhet raportuar gabimin apo përfunduar programin me metodën e kompajlimit *debug*.

Në funksionin *Fshije* kemi përdorë funksionin *assert* (i definuar në fajllin *<assert.h>*) që shtyp informata diagnostike të programit dhe e përfundon programin nëse vlera e shprehjes së parametrut është e barabartë me zero. Nëse vlera e parametrut nuk është e barabartë me zero, funksioni *assert* pason ekzekutimin e kodit në rreshtin tjetër, pa e shtypur ndonjë informatë. Informatat e shtypura nga ky funksion mvaren nga kompajlerët e ndryshëm, mirëpo në të shumtën e rasteve e kemi ndonjërin nga informatat në vazhdim: emrin e fajllit të kodit, rreshtin në të cilin funksioni *assert* është thirrur etj. Duhet ta keni parasysh se funksioni *assert* është shumë i vlefshëm në metodën e kompajlimit *debug* (pra në fazat e implementimit dhe testimit të programit). Zakonisht funksioni *assert* në metodën e kompajlimit *release* nuk ndërmerr asgjë (nuk shtyp ndonjë informatë, mirëpo edhe pse parametri është zero, nuk e terminon programin). Për këtë arsye, ky funksion nuk mund të përdoret (dhe të panifikoni ta përdorni) për të shtypur ndonjë informatë në produktin final të programit.

Në kodin që vijon kemi implementuar funksionin *Fshije*, i cili merr si parametër numrin e renditjes së elementit për t'u fshirë:

```
void Vektori::Fshije(int iIndeksi)
{
    // indeksi nuk mundet të jetë më i madh se numri
    // i anëtarëve (elementeve) apo më i vogël se zero
    assert(iIndeksi >= 0 && iIndeksi < m_iAnetaret);

    elementi* p          = m_pElementet;

    // nëse elementi i parë
    if (iIndeksi == 0)
    {
        m_pElementet = m_pElementet->tjetri;

        delete p;
    }
    else
```

```

{
    // trego në një element para elementit për t'u fshirë
    while (--iIndeksi)
    {
        p = p->tjetri;

        elementi* pTmp      = p->tjetri;
        p->tjetri             = pTmp->tjetri;

        delete pTmp;
    }

    m_iAnetaret--;
}

```

Funksioni Fshije merr parasysh dy raste për të fshirë elementin në objektin Vektori:

- ♦ elementin e parë në listë
- ♦ çdo element tjetër

Nëse tentojmë ta fshijmë elementin e parë në listë, atëherë treguesi tjetri i elementit të parë tregon në elementin e ardhshëm në listë (apo NULL nëse lista përmban vetëm një element) dhe pastaj e lirojmë memorien e elementit të parë.

Në rastin tjetër, nëse tentojmë ta fshijmë elementin tjetër në listë (pra përpos elementit të parë), kemi përdorur vorbullën përsëritëse për të gjetur një element para elementit të kërkuar në listë. Në këtë rast, treguesi i elementit para elementit të kërkuar do të tregojë në objektin e treguar nga treguesi tjetri i elementit të kërkuar dhe pastaj e lirojmë memorien e elementit të kërkuar. Për ta kuptuar më lehtë këtë, shikoni figurat që vijojnë:

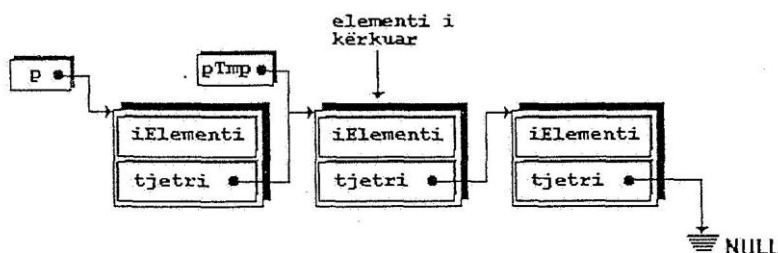
Pas ekzekutimit të kodit

```

while (--iIndeksi)
{
    p = p->tjetri;
}
elementi* pTmp = p->tjetri;

```

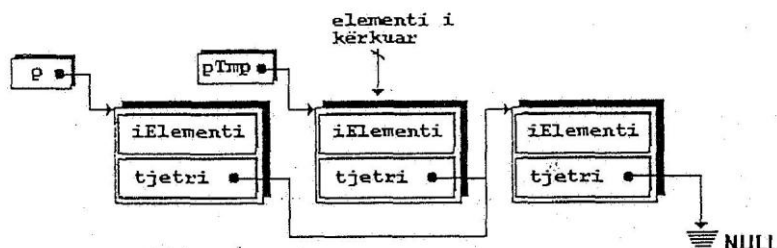
do të kemi këtë pozitë në listën e elementeve:



Pas ekzekutimit të kodit:

```
p->tjetri = pTmp->tjetri;
```

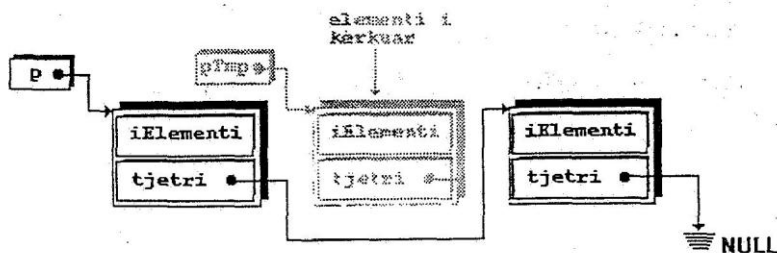
do të kemi këtë pozitë në listën e elementeve:



Dhe në fund, pas ekzekutimit të kodit:

```
delete pTmp;
```

do ta kemi këtë pozitë:



Funksioni anëtar NumriIAnetareve ka për detyrë kthimin e vlerës së variablës anëtare m_iAnetaret. Pra implementimi i këtij funksioni bëhet kështu:

```
int Vektori::NumriIAnetareve()
{
    return m_iAnetaret;
}
```

Pasi që klasa vektori rezervon në mënyrë dinamike memorien për listën e elementeve, atëherë kjo klasë duhet të kujdeset për lirimin e memories. Zakonisht në klasë, lirimi i memories bëhet në destruktore dhe për këtë arsye kemi implementuar destruktoren kështu:

```
Vektori::~Vektori()
{
    // liro memorien e rezervuar për çdo element në objekt
    if (m_pElementet != NULL)
    {
        elementi* pTmp      = NULL;
        elementi* p         = m_pElementet;

        while (p != NULL)
        {
            // trego në elementin për ta liruar memorien
            pTmp = p;

            // para se ta lirojmë memorien duhet
            // të tregojmë në objektin tjetër.
            p = p->tjetri;
            delete pTmp;
        }
    }
}
```

Një gjë që vlen të përmendet është se gjatë lirimit të memories së objektit në listë duhet pasur kujdes që të mos e përdorim memorien e objektit pas lirimit të objektit. P.sh. shihni vorbullën while në destruktoren e klasës vektori:

```
while (p != NULL)
{
    // trego në elementin për ta liruar memorien
    pTmp = p;

    // para se ta lirojmë memorien duhet
    // të tregojmë në objektin tjetër.
    p = p->tjetri;

    delete pTmp;
}
```

}

Shumica e programerëve do të mendojnë se kodi që vijon është plotësisht i vlefshëm dhe i ngjashëm me atë më sipër:

```
while (p != NULL)
{
    // trego në elementin për ta liruar memorien
    pTmp = p;

    // liroje memorien e objektit
    delete pTmp;

    // trego në objektin tjetër.
    p = p->tjetri;           // gabim
}
```

Në kodin e mësipërm kemi liruar memorien e treguar nga treguesi pTmp, mirëpo e njëjta memorie është e treguar nga treguesi p. D.m.th. e kemi liruar memorien e treguar nga vetë treguesi p dhe prandaj rreshti:

```
p = p->tjetri; // gabim
```

nuk është i vlefshëm, për arsye se memoria e treguesit p->tjetri është liruar dhe nuk vlen për procesin ekzekutues.

E njëjta gjë vlen edhe për funksionin anëtar Fshije të klasës Vektori, pra duhet pasur kujdes në renditjen e kodit për lirimin e memories.

Për kompletim të klasës Vektori kemi implementuar edhe konstruktorin që inicon vlerat anëtare të objektit:

```
Vektori::Vektori()
{
    m_iAnetaret = 0;
    m_pElementet = NULL;
}
```

Për testimin e klasës Vektori të implementuar, deri tani kemi përdorë kodin e mëposhtëm:

```
void main ()
{
    Vektori v;

    v.Shto(1);
    v.Shto(2);
    v.Shto(3);

    cout << "Numri i anëtarëve është "
          << v.NumriIAnetareve() << endl;
```



```

v.Fshije(1);

cout << "Numri i anëtarëve është "
      << v.NumriIANetareve();
}

```

Nëse e analizoni klasën vektori dhe funksionin e kësaj klase, do të vëreni se kjo klasë nuk është aspak e përdorshme ashtu siç është implementuar deri më tani. Arsyeja është se në këtë klasë mundemi të ruajmë vlera të ndryshme dhe t'i fshijmë vlerat e ruajtura më parë, mirëpo gjëra kryesore që mungojnë janë metoda e leximit të vlerave të ruajtura në objektin e tipit vektori si dhe ndryshimi i vlerave të elementeve të ruajtura.

Për këtë arsye duhet zgjeruar funksionin e klasës vektori, duke mundësuar leximin dhe ndryshimin e vlerave të ruajtura në objekt. Këtë do të mund ta bënim me shtimin e dy funksioneve anëtare, p.sh. për ndërrimin e vlerës në renditjen iIndeksi:

```
void NderroVleren(int iIndeksi, int ivlera);
```

si dhe për leximin e vlerës në renditjen iIndeksi:

```
int LexoVleren(int iIndeksi);
```

Pasi që klasa vektori simulon tipin array do të ishte më mirë ta implementonim operatorin [] për leximin dhe ndërrimin e vlerave. Operatori [] do ta marrë si parametër renditjen e elementit në listë.

```

class Vektori
{
public:
    Vektori();
    ~Vektori();

public:
    void Shto(int iAnetari);
    void Fshije(int iIndeksi);
    int NumriIANetareve();

    int operator[] (int iIndeksi);

private:
    struct elementi
    {
        int iElementi;
        struct elementi* tjetri;
    } *m_pElementet;

    int m_iAnetaret;
};

```

Implementimi i operatorit [] do të jetë i ngjashëm me funksionin Fshije, mirëpo në vend se ta fshijmë memorien e rezervuar për element, në këtë operator e kthijmë vlerën e elementit të kërkuar:

```
int Vektori::operator[] (int iIndeksi)
{
    // indeksi nuk mundet të jetë më i madh se numri
    // i anëtarëve (elementeve) apo më i vogël se zero
    assert(iIndeksi >= 0 && iIndeksi < m_iAnetaret);

    elementi* p = m_pElementet;

    // nëse elementi i parë --
    if (iIndeksi == 0)
    {
        return m_pElementet->iElementi;
    }
    else
    {
        // trego në elementin me renditje iIndeksi
        while (iIndeksi--)
        {
            p = p->tjetri;
        }

        return p->iElementi;
    }
}
```

Në operatorin [] nuk kemi nevojë të kërkojmë për treguesin e elementit në renditje para elementit për të cilin kërkojmë vlerën. Për këtë arsye e kemi përdorë auto-operatorin – pas leximit të vlerës së parametrin iIndeksi:

```
while (iIndeksi--)
```

që shkakton gjetjen e treguesit të vetë elementit të kërkuar.

Në funksionin Fshije e kemi përdorë auto-operatorin – para leximit të vlerës së parametrin iIndeksi:

```
while (--iIndeksi)
```

që shkakton gjetjen e treguesit të elementit para elementit të kërkuar.

Tani, pasi që e kemi implementuar operatorin [] do të mund t'i përdorim objektet e tipit vektori sikurse objektet e tipit array, p.sh.:

```
Vektori v;
```

```

v.Shto(1);
v.Shto(2);
v.Shto(3);

int i = 0;
cout << v[i++] << " ";
cout << v[i++] << " ";
cout << v[i] ;

```

do të shtypë 1 2 3.

Nëse tentoni ta ndërroni vlerën e elementit në objekt të tipit Vektori, p.sh.:

```
Vektori v;
```

```

v.Shto(1);
v.Shto(2);
v.Shto(3);

```

```
v[1] = 5;
```

disa kompajlerë do ta paraqesin si gabim, kurse në disa kompajlerë vlera e elementit në renditje me indeks 1 nuk ndryshohet fare. Arsyeja është mënyra e implementimit të operatorit []. Nëse e shikoni prototipin e operatorit {}:

```
int operator[] (int iIndeksi);
```

do të vëreni se ky operator kthen kopjen e vlerës së elementit të kërkuar. Kjo do të ishte plotësisht e vlefshme nëse vetëm lexojmë vlerat e elementeve në objektin vektori. Mirëpo, nëse tentojmë t'i ndërrojmë vlerat e elementeve në objektin vektori, kjo nuk është e vlefshme, sepse do të tentonim ta ndërronim vlerën e kopjes së elementit.

Ky problem zgjidhet nëse në vend të kopjes e kthejmë referencën e vlerës së elementit, p.sh.:

```
int& operator[] (int iIndeksi);
```

Implementimi i operatorit [] është i njëjtë si më parë, përveçqë definimi i funksionit do ta kthejë referencën në int:

```
int& Vektori::operator[] (int iIndeksi)
```

Tani kodi për ndërrimin e vlerës së elementit në objektin Vektori është plotësisht i vlefshëm:

```

Vektori v;

v.Shto(1);

```

```

v.Shto(2);
v.Shto(3);

v[1] = 5;

cout << v[1] ; // do të shtypë 5

```

Objekti i tipit vektori simulon objektin e tipit *array*, mirëpo përparësia e objektit të tipit vektori është se ai mund të rritet në mënyrë dinamike. Nëse e krahasoni me objektin e tipit *array* do të vëreni se objekti vektori mund të përdoret me çfarëdo tipi, pa marrë parasysh a është tip i thjeshtë apo tip i definuar nga vetë përdoruesit.

Në rastin tonë, objekti i tipit vektori momentalisht mund të përdoret vetëm me tipin *int*. Njëra prej mënyrave për të mundësuar që objekti i tipit vektori të përdoret edhe me tipe të tjera, do të ishte të implementonim klasë për çdo tip. Pasi që e kemi implementuar klasën vektori për tipin *int*, do të ishte lehtë ta kopjonim këtë kod dhe ta përdornim për tipe të tjera.

P.sh. nëse do ta përdornim objektin e tipit të llojit vektori me tipin *char*, do ta deklaronim klasën *vektoriChar* kështu:

```

class VektoriChar
{
public:
    VektoriChar();
    ~VektoriChar();

public:
    void Shto(char cAnetari);
    void Fshije(int iIndeksi);
    int NumriAnetareve();

    char& operator[] (int iIndeksi);

private:
    struct elementi
    {
        char cElementi;
        struct elementi* tjetri;
    } *m_pElementet;

    int m_iAnetaret;
};

```

Kjo metodë e kopjimit të kodit nuk është aspak e dëshirueshme. Nëse p.sh. e kemi gjetur ndonjë gabim në implementimin e kodit të klasës vektori, atëherë do të duhej të shikonim për të njëjtin gabim në të gjitha klasët e tjera të implementuara për tipe të tjera.

Për këtë arsye përdoren shabllonet, të cilat mundësojnë që e njëjta klasë të përdoret për çfarëdo tipi të thjeshtë apo të definuar nga përdoruesi. Në këtë mënyrë gabimet mund të përmirësohen përnjëherë për të gjitha tipet.

Metoda më e rekomanduar për krijimin e shabllove do të ishte metoda e përdorur në këtë libër, pra për krijimin e klasëve së pari për tip të thjeshtë e pastaj shndërrimin e klasës në shabllon. Në këtë mënyrë, klasa e krijuar për tip të thjeshtë mund të testohet si duhet dhe pasi të jeni të kënaqur me rezultatet e klasës së thjeshtë, mund ta shndërroni atë në shabllon. Duhet kuptuar se për disa tipe të krijuara nga përdoruesit ndonjëherë duhet zgjeruar funksionin e tyre për t'iu përshtatur shabllove, p.sh. disa tipeve u mungon operatori = apo ndonjë operator tjetër. Kjo do të thotë se tipet e përdorura në shabllo ne duhen të kenë disa nga funksionet e duhura të cilat mvaren nga funksioni i klasës shabllon.

Në kodin që vijon e kemi shndërruar klasën vektori, të implementuar më sipër për tipin int, në shabllon. Implementimi i klasës vektori nuk përdor asgjë specifike për tipin e përdorur, kështu që kjo klasë mund të përdoret pothuajse për çdo tip.

Nëse përkujtoni klasën string, definimin e saj e kemi bërë në fajllin header, kurse implementimin në fajllin me prapashtesë cpp. Për klasët shabllon nuk është e mundur të bëhet ndarja e definimit të klasës dhe e implementimit të klasës në dy fajlla. Definimi i klasës shabllon dhe implementimi i saj duhen të jenë në një fajll, pra në fajllin header. Kjo është për arsye të kompajlerit të gjuhës C++ i cili në mënyrë dinamike tenton të krijojë klasë për çdo tip të përdorur me shabllon.

Pra definimin dhe implementimin e klasës vektori e kemi bërë në fajllin vektori.h.

```
#ifndef Vektori_h
#define Vektori_h

#include <assert.h>

template <class T>
class Vektori
{
public:
    Vektori();
    ~Vektori();

public:
    void Shto(T tAnetari);
    void Fshije(int iIndeksi);
    int NumriAnetareve();
```

```

        T&    operator[] (int iIndeksi);
private:
    struct elementi
    {
        T        tElementi;
        struct elementi* tjetri;
    } *m_pElementet;

    int m_iAnetaret;
};

////////////////////////////////////
// implementimi i klasës Vektori
////////////////////////////////////
template <class T>
Vektori<T>::Vektori()
{
    m_iAnetaret = 0;
    m_pElementet = NULL;
}

template <class T>
Vektori<T>::~Vektori()
{
    // liro memorien e rezervuar për çdo element në objekt
    if (m_pElementet != NULL)
    {
        elementi* pTmp      = NULL;
        elementi* p          = m_pElementet;

        while (p != NULL)
        {
            // trego në elementin për ta liruar memorien
            pTmp = p;

            // para se ta lirojmë memorien duhet
            // të tregojmë në objektin tjetër.
            p = p->tjetri;

            delete pTmp;
        }
    }
}

template <class T>
int Vektori<T>::NumriAnetareve()
{
    return m_iAnetaret;
}

template <class T>

```

```

void Vektori<T>::Shto(T tAnetari)
{
    // nëse në fillim të listës, d.m.th. elementi i parë
    if (m_pElementet == NULL)
    {
        m_pElementet = new elementi;

        m_pElementet->tElementi    = tAnetari;
        m_pElementet->tjetri        = NULL;

        m_iAnetaret                = 1;
    }
    else
    {
        // shto elementin e ri në fund të listës

        elementi* p = m_pElementet;
        while (p->tjetri != NULL)
            p = p->tjetri;

        // rezervoj memorien e duhur për element të ri
        elementi* pIri = new elementi;

        pIri->tElementi    = tAnetari;
        pIri->tjetri        = NULL;

        // bashkangjite elementin e ri në fund të listës
        p->tjetri          = pIri;
        m_iAnetaret++;
    }
}

template <class T>
T& Vektori<T>::operator[] (int iIndeksi)
{
    // indeksi nuk mundet të jetë më i madh se numri
    // i anëtareve (elementeve) apo më i vogël se zero
    assert(iIndeksi >= 0 && iIndeksi < m_iAnetaret);

    elementi* p          = m_pElementet;

    // nëse elementi i parë
    if (iIndeksi == 0)
    {
        return m_pElementet->tElementi;
    }
    else
    {
        // tregoj në elementin në renditje iIndeksi

```

```

        while (iIndeksi--)
        {
            p = p->tjetri;
        }

        return p->tElementi;
    }
}

template <class T>
void Vektori<T>::Fshije(int iIndeksi)
{
    // indeks i nuk mund të jetë më i madh se numri
    // i anëtarëve (elementeve) apo më i vogël se zero
    assert(iIndeksi >= 0 && iIndeksi < m_iAnetaret);

    elementi* p          = m_pElementet;

    // nëse elementi i parë
    if (iIndeksi == 0)
    {
        m_pElementet = m_pElementet->tjetri;

        delete p;
    }
    else
    {
        // trego në një element para atij për t'u fshirë
        while (--iIndeksi)
        {
            p = p->tjetri;
        }

        elementi* pTmp      = p->tjetri;
        p->tjetri            = pTmp->tjetri;

        delete pTmp;
    }

    m_iAnetaret--;
}

#endif

```

Shndërrimi i klasës origjinale vektori në shabllon, që shumë i lehtë duke zëvendësuar tipin e anëtarit të strukturës elementi në T (tip i mvarur në parametrin e objektit të definuar), si dhe zëvendësimin e parametrin të funksionit shtoi dhe tipin kthyes të operatorit [].

Për testimin e shabllonit vektori në kodin në vijim kemi përdorë tipin int, char dhe float:

```
void main ()
{
    Vektori<int> vInt;
    Vektori<char> vChar;
    Vektori<float> vFloat;

    vInt.Shto(1);
    vInt.Shto(2);
    vInt.Shto(3);

    vChar.Shto('a');
    vChar.Shto('b');
    vChar.Shto('c');

    vFloat.Shto(1.0);
    vFloat.Shto(2.0);
    vFloat.Shto(3.0);

    cout << vInt[0] << " " << vInt[1] << " " << vInt[2]
         << endl;

    cout << vChar[0] << " " << vChar[1] << " " << vChar[2]
         << endl;

    cout << vFloat[0] << " " << vFloat[1] << " " << vFloat[2]
         << endl;
}
```

Sig e vëreni më sipër, të njëjtën klasë e kemi përdorë për tipe të ndryshme. Përparësia e shablloneve qëndron në atë se i njëjti kod mund të përdoret për shumë tipe, duke kursyer orë të tëra në testimin dhe implementimin e kodit për çdo lloj të tipit të përdorur.

Duhet ta keni parasysh se shabllonet janë tipe kontrolluese të forta (veti shumë e vlefshme në programim), p.sh. nëse kemi dy variabla të ndryshme:

```
Vektori<int> vInt;
Vektori<string> vString;

vInt.Shto(15);
vString.Shto("Kosova");
```

nuk është e mundur ta iniconi variablën vInt me vString:

```
vInt[0] = vString[0]; // gabim
```

11.1.2 Tipi Rreshti

Tipi Rreshti është tip mjaft i përdorshëm në problemet e jetës së përditshme. Për shpjegimin e tipit Rreshti do ta marrim problemin e një shitoreje, e cila i shet produktet vetëm nëpërmjet telefonit. Kjo shitore i merr porositë nëpërmjet telefonit dhe pastaj punëtorët i paketojnë produktet e porositura, të cilat i dërgojnë me postë te klienti.

Ta krijojmë programin për t'ia lehtësuar punën punëtorëve në shitore për marrjen e porositë dhe dërgimin e tyre.

Porositë e marra nga blerësit në telefon duhet të regjistrohen në program. Pastaj këto porosi duhen të dërgohen me renditjen e ardhur. Pra, porosia e blerësit të parë duhet të shërbehet e para, pastaj porosia e blerësit të dytë e kështu me radhë. Është logjike që për implementimin e programit për shitore të krijojmë tipin e ri Rreshti që do t'i ruajë porositë e blerësve. Pasi që tipi Rreshti mund të përdoret për tipe të ndryshme, do t'i përdorim shabllonet për ta bërë tipin Rreshti tip të përgjithshëm. Nëse e shqyrtojmë problemin rreth rreshtit të blerësve, do t'i vërejmë funksionet e duhura për klasën Rreshti. P.sh. blerësit thirrën me telefon për ta porositur produktin, ndërsa punëtori në shitore e regjistron porosinë e marrë në telefon, pra e shiton porosinë e marrë në listën e porositë të tjera. Pas një kohe, shitësi në shitore vendos që të gjitha porositë e marra në telefon t'i shërbejë – ti paketojë dhe t'i bëjë gati për postim. Pas konsiderimit të problemit, kemi marrë përfundimet e mëposhtme për definimin e klasës Rreshti:

```
template <class T>
class Rreshti
{
public:
    Rreshti();
    ~Rreshti();

public:
    void Shto(const T& tPorosia);
    T Sherbe();
    bool RreshtiIZbrazet();

private:
    struct elementi
    {
        T tElementi;
        struct elementi* tjetri;
    };

    elementi* m_pFillimiIRreshtit;
    elementi* m_pFundiIRreshtit;
};
```

Funksioni anëtar shtoi do të përdoret për regjistrimin e porosive, pra për shtimin e porosive në listë, kurse funksioni shërbe do të përdoret për shërbimin e porosive. Për ta filluar paketimin e produkteve, duhet që së paku ta kemi ndonjë porosi në rresht, prandaj kemi krijuar funksionin RreshtiIzbrazet për të shikuar nëse rreshti është i zbrazët.

Brenda klasës Rreshti e kemi definuar tipin elementi për t'i ruajtur të dhënat e një elementi në rresht (në rastin tonë për t'i ruajtur porosinë). Pasi që porosinë duhet të shërbehen me rend, pra prej fillimit të rreshtit, atëherë kemi deklaruar variabël anëtare m_pFillimiIRreshtit për të treguar në fillim të rreshtit. Shtimi i porosive bëhet në fund të rreshtit, prandaj kemi deklaruar variablën m_pFundiIRreshtit për të treguar në fund të rreshtit. Nëse e shikoni implementimin e tipit vektori do ta vëreni se e kemi përdorë vetëm një tregues për të treguar në fillim të listës së elementeve. Të njëjtën teknikë mund ta përdorim edhe për tipin Rreshti, mirëpo atëherë do të duhej ta kërkonim çdo herë fundin e listës për shtimin e porosisë. Kjo nuk është aspak efikase, prandaj kemi deklaruar dy tregues, ku njëri tregon në fillim të listës (rreshtit) së elementeve, kurse tjetri tregon në fund të listës.

Implementimin e klasës Rreshti e kemi paraqitur në kodin që vijon:

```

////////////////////////////////////////
// implementimi i klasës Rreshti
////////////////////////////////////////
template <class T>
Rreshti<T>::Rreshti()
{
    m_pFillimiIRreshtit = NULL;
    m_pFundiIRreshtit   = NULL;
}

template <class T>
Rreshti<T>::~Rreshti()
{
    // liro memorien e rezervuar per çdo element ne rresht
    if (m_pFillimiIRreshtit != NULL)
    {
        elementi* pTmp      = NULL;
        elementi* p         = m_pFillimiIRreshtit;

        while (p != NULL)
        {
            // trego ne elementin per te liruar memorien
            pTmp = p;

            // para se ta lirojmë memorien duhet
            // te tregojmë ne objektin tjetër.
            p     = p->tjetri;
        }
    }
}

```

```

        delete pTmp;
    }
}

template <class T>
void Rreshti<T>::Shto(const T& tAnetari)
{
    // nese rreshti i zbrazet
    if (RreshtiIZbrazet())
    {
        m_pFillimiIRreshtit = new elementi;

        m_pFillimiIRreshtit->tElementi = tAnetari;
        m_pFillimiIRreshtit->tjetri = NULL;
        m_pFundiIRreshtit = m_pFillimiIRreshtit;
    }
    else
    {
        // shto elementin e ri ne fund te rreshtit

        // rezervu memorien e duhur per element te ri
        elementi* pIri = new elementi;

        pIri->tElementi = tAnetari;
        pIri->tjetri = NULL;

        // bashkangjite elementin e ri ne fund te listes
        m_pFundiIRreshtit->tjetri = pIri;

        m_pFundiIRreshtit = m_pFundiIRreshtit->tjetri;
    }
}

template <class T>
bool Rreshti<T>::RreshtiIZbrazet()
{
    return m_pFillimiIRreshtit == NULL;
}

template <class T>
T Rreshti<T>::Sherbe()
{
    assert(!RreshtiIZbrazet());

    elementi* p = m_pFillimiIRreshtit;
    T tPorosia = m_pFillimiIRreshtit->tElementi;

    m_pFillimiIRreshtit = m_pFillimiIRreshtit->tjetri;
}

```

```
// liro memorien e rezervuar për elementin e parë
// ne rresht
delete p;
```

```
return tPorosia
```

```
}
```

Tani pasi që e kemi implementuar klasën Rreshti, krijimi i programit për shitore do të jetë shumë më i lehtë. Për t'ia lehtësuar punën punëtorëve në shitore do ta shtypim menynë me mundësitë e funksioneve në program. Pra përdoruesi i programit do ta zgjedhë në meny funksionin e dëshiruar. Në kodin që vijon e kemi implementuar programin, duke marrë parasysh se definimi dhe implementimi i klasës Rreshti, të diskutuar deri më tani është bërë në fajllin Rreshti.h.

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <assert.h>
#include "Rreshti.h"
```

```
using namespace std;
```

```
#define GJATESIA_E_POROSISE 255
#define KOMANDA_PER_PASTRIM "cls"
```

```
typedef string Porosia;
```

```
// prototipi i funksioneve
void ShtypeMenyne();
void PastroMonitorin();
void ShtoPorosine(Rreshti<string>& rreshti);
void SherbePorosite(Rreshti<string>& rreshti);
void TrajtoGabimin();
void PastroTeDhenatHyrese();
```

```
void main ()
```

```
{
    Rreshti<Porosia> rreshti;

    char chKomanda;

    do
    {
        PastroMonitorin();
        ShtypeMenyne();

        // lexo komandën
        cin.get(chKomanda);
```

```

        PastroTeDhenatHyrse();

        switch (chKomanda)
        {
        case '1':
            ShtoPorosin(rreshti);
            break;
        case '2':
            SherbePorosit(rreshti);
            break;
        case '3':
            // perfundo programin
            break;
        default:
            TrajtoGabimin();
        }

        cin.clear();
        // lexo germen enter
        //cin.get(chTemp);

    } while (chKomanda != '3');
}

// implementimi i funksioneve

void ShtypeMenyne()
{
    cout << "*****" << endl;
    cout << "*****          Shitorja KOSOVA          *****" << endl;
    cout << "*****" << endl;
    cout << endl;
    cout << endl;
    cout << "      1.  Shto porosinë" << endl;
    cout << "      2.  Shërbe porositë" << endl;
    cout << endl;
    cout << "      3.  Përfundo programin" << endl;
    cout << endl;
    cout << endl;
    cout << "*****" << endl;
    cout << "Shtype njërën prej komandave 1, 2, ose 3" << endl;
}

void PastroMonitorin()
{
    system(KOMANDA_PER_PASTRIM);
}

```

```
}

void ShtoPorosine(Rreshti<Porosia>& rreshti)
{
    char          caPorosia[GJATESIA_E_POROSISE + 1];
    Porosia       sPorosia;

    cout << "Shtype porosine : " ;
    cin.getline(caPorosia, GJATESIA_E_POROSISE);

    sPorosia = caPorosia;

    rreshti.Shto(sPorosia);
}

void SherbePorosite(Rreshti<Porosia>& rreshti)
{
    if (rreshti.ReshtiIZbrazet())
    {
        cout << "Nuk ka porosi te pashërbyer" << endl ;
    }
    else
    {
        while(!rreshti.RështiIZbrazet())
        {
            cout << "Shërbehet porosia: "
                << rreshti.Sherbe() << endl ;
        }
    }

    cout << "Shtype ENTER për të vazhduar" << endl ;

    PastroTeDhenatHyrese();
}

void TrajtoGabimin()
{
    cout << "Komanda e shtypur është komandë e gabuar "
        << endl;
    cout << "Shtyp ENTER per të vazhduar" << endl ;

    PastroTeDhenatHyrese();
}

void PastroTeDhenatHyrese()
{
    char ch;

    do
    {
```

```

        cin.get(ch);

    }while (ch != '\n');
}

```

Nodnëse emërtimi i funksioneve ndihmon mjaft për ta kuptuar veprimin e tyre, prapëseprapë do të flasim shkurtimisht për secilin funksion. Funksioni `shtypeMenyne` përdoret për ta shtypur menynë e programit (listën e komandave). Funksioni `PastroMonitorin` përdoret për pastrimin e monitorit. Implementimi i këtij funksioni është mjaft i thjeshtë. Ai përdor funksionin e librarisë standarde `system` që përdoret për ekzekutimin e komandave të sistemit operativ. Këtë program e kemi kompajluar dhe testuar në sistemin operativ *Microsoft Windows* prandaj kemi përdorur komandën `cls` për pastrimin e monitorit, pra:

```
system(KOMANDA_PER_PASTRIM);
```

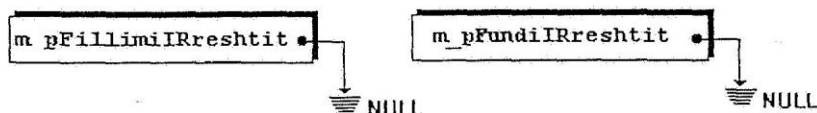
ku `KOMANDA_PER_PASTRIM` është definuar si `"cls"`. Nëse këtë program e kompajloni në sistemin operativ *UNIX* apo *Linux*, atëherë duhet ta ndërroni definimin e makros `KOMANDA_PER_PASTRIM` në `"clear"` (komanda `"clear"` në *UNIX* dhe *Linux* është e ngjashme sikurse komanda `"cls"` në *DOS*), pra:

```
#define KOMANDA_PER_PASTRIM    "clear"
```

Funksioni `shtoPorosine` përdoret për shtimin e porosive në listën e porosive të tjera nga shitësi, të cilat porosi ruhen përderisa nuk "shërbehen" me funksionin `sherbePorosite`. Funksioni `TrajtoGabimin` përdoret për ta informuar përdoruesin e programit nëse shtyp komandë të gabuar. Dhe në fund, funksioni `PastroTeDhenatHyrese` përdoret për pastrimin e të dhënave hyrëse. Nëse e shikoni implementimin e këtij funksioni, do të vëreni se ky funksion lexon shkronjat nga objekti `cin` (hyrja e të dhënave) përderisa përdoruesi nuk e shtyp shkronjën ENTER (rrëshiti i ri). Këtë e bëjmë për arsye se komanda përmban vetëm një shkronjë (shkronjën '1', '2' apo '3'). Për ta lexuar programi komandën, përdoruesi duhet ta shtypë shkronjën ENTER. Pasi që simbolet e vetme të vlefshme në këtë program janë '1', '2' dhe '3', duhet t'i lexojmë të gjitha simbolet e tjera të shtypura pas këtyre simboleve duke përfshirë edhe simbolin për ENTER, për të mos i paraqitur si gabim rastet kur përdoruesi shtyp p.sh. '1' dhe pastaj preklën ENTER. Natyrisht se kjo nuk është ideale, për arsye se programi i merr si komanda edhe simbolet e pavlefshme pas simboleve të vlefshme. Në rastin tonë programi merr parasysh vetëm simbolin e parë për komandë, kurse simbolet të tjera të shtypura në mes të simbolit të parë dhe të simbolit ENTER (duke përfshirë edhe simbolin ENTER) injorohen.

Për ta kuptuar më mirë implementimin e klasës Rreshti si dhe metodën se si i ruan ajo të dhënat, do ta paraqesim në mënyrë grafike çdo nivel të ruajtjes së të dhënave (me ç'rast të dhëna do të jenë porositë).

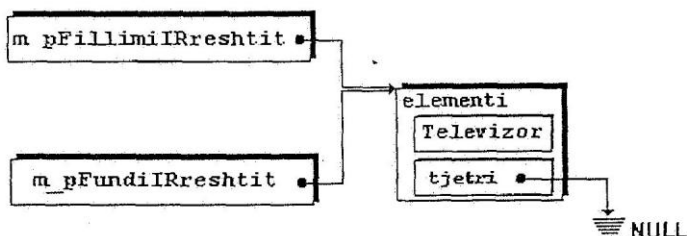
Në fillim, kur e konstruktojmë objektin e tipit Rreshti, lista e anëtarëve në rresht do të jetë e zbrazët, prandaj variablat anëtare të klasës Rreshti do të jenë të barabarta me NULL, pra:



Pas shtimit të një porosie, p.sh.:

```
Rreshti<Porosia> rreshti;
rreshti.Shto("Televizor");
```

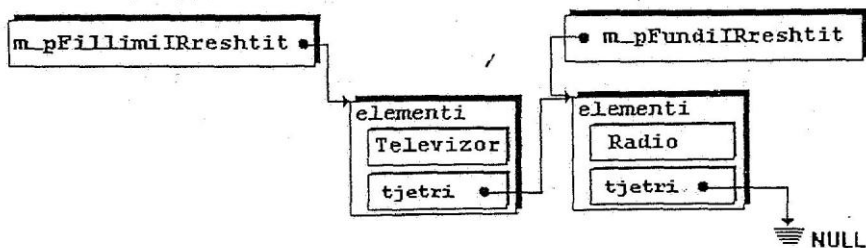
do ta kemi pozitën e paraqitur në figurën që vijon:



Nëse e shtojmë edhe një porosi në rresht, p.sh.:

```
rreshti.Shto("Radio");
```

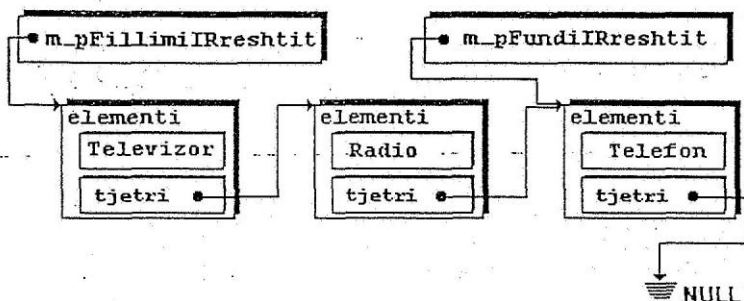
atëherë do ta kemi pozitën si në figurën që vijon:



Nëse shtojmë edhe një porosi në rresht, p.sh.:

```
rreshti.Shto("Telefon");
```

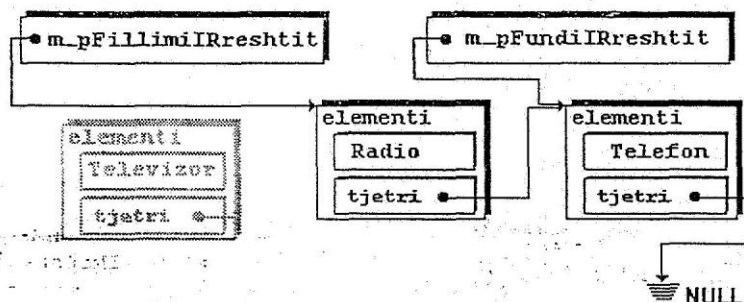
atëherë do ta kemi pozitën si në figurën që vijon:



Tani nëse e shërbejmë vetëm një porosi, p.sh.:

```
rreshti.Sherbe();
```

atëherë do ta kemi pozitën e paraqitur në figurën që vijon:



Siç e vëreni në figurë, porosia e parë në rresht është shërbyer dhe është fshirë nga lista e porosive dhe treguesi anëtar m_pFillimiIRreshtit tregon në porosinë e dytë, e cila tani në listë bëhet e para.

11.2 Përsëritësit

Klasët shabllone të ofruara nga libraritë standarde si vektori, map, list, set etj. njihen si klasë mbajtëse (angl. container classes), për arsye se këto klasë përdoren për mbajtjen e një apo më shumë objekteve të ndryshme. Secila prej këtyre klasëve i ka përparësitë dhe dobësitë e veta, p.sh. disa klasë mbajtëse janë më efikase për shtimin e anëtarëve, kurse disa janë më efikase për kërkimin e anëtarëve në objekt. Edhe pse implementimi i klasëve mbajtëse është i ndryshëm për secilën klasë, do të ishte mirë që këto klasë të kenë një metodë të njëjtë për të manipuluar me vlerat e anëtarëve në listë. Duke ofruar metodë të njëjtë për të manipuluar me vlerat e anëtarëve në listë, mundësojmë që kodi të jetë më portabil. Përparësia tjetër është se mënyra e përdorimit të këtyre klasëve mësohet dhe përdoret më lehtë nga përdoruesit pasi që kanë një *interfejs* të njëjtë.

Duke i marrë parasysh përparësitë e interfejsit të njëjtë, libraritë standarde i ofrojnë klasët mbajtëse me *përsëritës* të cilët mundësojnë aksesin e çdo anëtari në çfarëdo lloji të klasës mbajtëse.

Përsëritësit janë klasë në vete dhe punojnë në bashkësi me klasët mbajtëse. Përsëritësit përmbajnë funksione për të filluar aksesin e anëtarëve në klasë mbajtëse, vizitimin e anëtarit tjetër në listë të klasës mbajtëse, kthimin prapa në anëtarin e mëparshëm si dhe testimin nëse jemi në fund të listës së anëtarëve të klasës mbajtëse. Përsëritësit në gjuhën angleze njihen si *iterator*, term që përdoret edhe në libraritë standarde.

Për të ilustruar njërin prej metodave të implementimit të përsëritësit, kemi implementuar përsëritësin për klasën vektori në kodin që vijon:

```
template < class C >
```

```
class Perseritesi
```

```
{
```

```
public:
```

```
    Perseritesi(Vektori<C>&.v) : m_Vektorii(v)
```

```
{
```

```
    m_iPozita = 0;
```

```
}
```

```
C AnëtariITanishem()
```

```
{
```

```
    assert( m_iPozita < m_Vektorii.NumriAnetareve());
```

```
    return m_Vektori[m_iPozita];
```

```
}
```

```

bool FundiIListes()
{
    return m_iPozita == m_Vektori.NumriIANetareve();
}

void AnetariTjetër()
{
    m_iPozita++;
}

private:
    Vektor<C>& m_Vektori;
    int m_iPozita;
};

```

Nëse të gjitha klasët mbajtëse të implementuara nga ju, furnizojnë përsëritës që përmbajnë të njëjtat funksione sikurse përsëritësi i implementuar më sipër, atëherë aksesit i anëtarëve në klasët mbajtëse do të jetë i njëjtë pa marrë parasysh implementimin e brendshëm të tyre.

P.sh. aksesin e çdo anëtari të klasës vektor do ta bënim kështu:

```

Vektor<int> v;
v.Shto(1);
v.Shto(12);
v.Shto(23);

Perseritesit<int> p(v);

while(!p.FundiIListes())
{
    cout << p.AnetariITanishem() << " ";
    p.AnetariTjetër();
}

```

Ju me siguri do ta vëreni se anëtarët e klasës vektor mund të vizitohen edhe kështu:

```

int iNumriIANetareve = v.NumriIANetareve();

for (int i=0; i < iNumriIANetareve; i++)
{
    cout << v[i] << " ";
}

```

Atëherë shtrohet pyetja: "Pse ta krijojmë një klasë tjetër si përsëritësi kur ne mund ta vizitojmë (të manipulojmë me) çdo anëtar të klasës vektor pa përsëritës"?

Nëse e përdorim vetëm klasën vektor, natyrisht se nuk do të nevojitej klasa përsëritësi për manipulim me anëtarët e kësaj klase. Mirëpo nëse kemi disa klasë të ngjashme sikur klasa vektor dhe këto klasë nuk e implementojnë operatorin {}, atëherë metoda për manipulimin me anëtarë të këtyre klasëve mund të jetë e ndryshme për secilën klasë. Në këtë mënyrë, përdoruesit e këtyre klasëve do të duheshin ta mësonin metodën për ta përdorë secilën klasë që natyrisht lë mundësi për gabime.

Përsëritësit mundësojnë një metodë të njëjtë për manipulimin me anëtarët e klasëve mbajtëse, gjë që ka këto përparësi:

- ♦ qartësi në kod,
- ♦ lehtësi në përdorimin e klasëve mbajtëse dhe
- ♦ lehtësi në ndërrimin e kodit (p.sh. zëvendësimin e një klase mbajtëse me një klasë tjetër mbajtëse më efikase).

Përsëritësit zakonisht janë të definuar brenda klasës mbajtëse, ose është përdorë udhëzimi `typedef` brenda në klasë. Pra çdo klasë mbajtëse përmban përsëritës specifik që ka interfejs të njëjtë me përsëritësit e klasëve të tjera mbajtëse. P.sh. nëse e marrim klasën `vector` të definuar në librarinë standarde STL (Standard Template Library) në fajllin `<vector>` përsëritësi definohet kështu:

```
vector<int> v;

// shtoji numrat 10, 21, 22 në listën e anëtarëve të objektit v
v.push_back(10);
v.push_back(21);
v.push_back(22);

// deklarë përsëritësin e tipit vector<int>
vector<int>::iterator theIterator = v.begin();

// vizitoje çdo anëtar në objektin v
for ( ; theIterator != v.end(); theIterator++)
{
    cout << *theIterator;

    if (theIterator != v.end()-1)
        cout << " ";
}
```

Natyrisht, libraritë standarde përmbajnë klasë mbajtëse më të pasura me funksione anëtare sesa klasa jonë vektori e implementuar në këtë kapitull. Po ashtu inerte i klasëve përsëritëse në librarinë standarde është tjetër nga ai i paraqitur në klasën Perseritesi. Klasën Perseritesi e kemi definuar vetëm sa për ta ilustruar funksionalitetin dhe përparësitë e përsëritësve. Ju rekomandojmë që të ushtroni me klasët mbajtëse të librarive standarde, sepse këto klasë janë shumë të vlefshme dhe në të shumtën e rasteve të pazëvendësueshme.

Një gjë që vlen të përmendet është se nëse i përdorni klasët mbajtëse të librarive standarde, duhet patjetër ta përdorni udhëzimin

```
using namespace std;
```

para përdorimit të klasëve mbajtëse.

11.3 Udhëzimi *typename*

Udhëzimi *typename* është një udhëzim i cili është paraqitur vonë në gjuhën C++. Ky udhëzim përdoret për t'i ndihmuar kompajlerit në përcaktimin e tipeve në shabllone. Në definimin e variablave të tipeve të thjeshta apo të tipeve të krijuara nga përdoruesit, kompajleri është i "vetëdijshëm" për memorien e duhur për ruajtjen e objektit si dhe për tipin e objektit. Mirëpo, kur kemi të bëjmë me shabllone, kompajleri përcakton tipet në mënyrë të ndryshme nga tipet normale.

P.sh. le të jetë klasa *Vektori* klasë shabllon e definuar sikur më parë. Atëherë, nëse e definojmë një variabël:

```
Vektori<int> vInt;
```

kompajleri do ta krijojë një klasë që përdor tipin *int* për çdo tip *T* (shih klasën *Vektori*) të përdorur në shabllon.

Nëse kemi variabla të tipeve të ndryshme, p.sh.

```
Vektori<int> vInt;
Vektori<char> vChar;
Vektori<float> vFloat;
// ...
```

atëherë kompajleri do të krijojë klasë për çdo tip të përdorur me klasën shabllon *Vektori*. Në një mënyrë kompajleri vetëm e lehtëson punën për përdoruesit, që këta të mos krijojnë klasë për çdo tip dhe të humbin kohë me testimin dhe mirëmbajtjen e kësaj klase për çdo tip.

Për lehtësimin e përdorimit të klasëve mbajtëse, libraritë standarde janë implementuar duke i përdorë përsëritësit. Implementimi i përsëritësve është pothuajse i ndryshëm për çdo lloj të klasëve mbajtëse dhe për këtë arsye përsëritësit definojnë brenda klasëve mbajtëse. Pra, për të mos pasur nevojë që përdoruesit të mësojnë pa nevojë se cilët përsëritës punojnë me cilët klasë mbajtëse, përsëritësit definojnë në të shumtën e rasteve kështu:

```
emri_i_klases_template::iterator objekti;
```

P.sh. për klasën mbajtëse *vector* dhe në klasën mbajtëse *map* përsëritësi definohet kështu:

```
vector<int>::iterator perseritesi_per_vector;
```

```
map<int, int, less<int> >::iterator p_per_map;
```

Paramendoni nëse kemi një klasë shabllon që mban elemente të tipeve të ndryshme dhe bën disa kalkulime me anëtarët e saj. Kjo klasë do të mund ta përdorte klasën mbajtëse (p.sh. vector) për mbajtjen e elementeve, p.sh.:

```
#include <vector>

using namespace std;

template <class Tipi>

class Data
{
public:
    void Shto(Tipi tAnetari)
    {
        m_vec.push_back(tAnetari);
    }

    void Kalkulo()
    {
        vector<Tipi>::iterator iter;

        for (iter = m_vec.begin(); iter != m_vec.end();
              iter++)
        {
            // bëji disa kalkulime me secilin
            // element në vektori
        }
    }

private:
    vector<Tipi> m_vec;
};
```

Klasa Data përmban funksionin anëtar për shtimin e elementeve anëtare si dhe funksionin për të kalkuluar ndonjë algoritëm të caktuar me elementet anëtare. Kjo klasë pra mund të përdoret për shumë tipe, të cilat përmbajnë operator apo funksione të përdorura në funksionin Kalkulo. Definimin e një variable të tipit Data do ta bënim kështu:

```
Data<int> iData;

iData.Shto(1);
iData.Shto(2);
iData.Kalkulo();
```



```
{
  //...
};
```

11.4 Treguesit e mençur (Smart pointers)

Njëra prej vetive më të fuqishme të gjuhës C dhe C++ është pa dyshim mundësia për manipulim me memorie në nivel të ulët. Edhe pse gjuhët C dhe C++ njihen si gjuhë të nivelit të lartë, këto gjuhë mundësojnë zgjidhjen e problemeve të nivelit të ulët, pra në nivel harduerik. Në anën tjetër, kjo veti (mundësia për ta përdorë gjuhën për probleme të nivelit të ulët) për disa përdorues, si për shembull për përdoruesit e gjuhëve si Visual Basic, Java etj., lë mundësi për të bërë gabime fatale në programim.

Njëri prej problemeve të paraqitura me programimin në gjuhët sikur C dhe C++ është ai kur rezervohet memoria për nevoja të ndryshme gjatë ekzekutimit të programit dhe harrohet të bëhet lirimi i memories së rezervuar. Ky problem është i njohur si "rrjedhja e memories". Ky problem nuk është aq fatal për programe të vogla dhe për programet të ekzekutuara më rrallë. T'ju përkujtojmë se me fikjen e makinës llogaritëse, memoria e rezervuar lirohet edhe për programet që harrojnë ta lirojnë memorien gjatë ekzekutimit të tyre. Mirëpo problemi i memories rrjedhëse është fatal për programet që ekzekutohen pandërprerë (p.sh. serverë të ndryshëm si p.sh. serveri për WWW, etj.). Nëse programet e ekzekutuara në mënyrë konstante harrojnë ta lirojnë memorien e rezervuar për nevoja të ndryshme, atëherë makina llogaritëse do të bllokohet dhe duhet të fiket për t'u ndezur sërish.

Ky problem është fatal për serverë të cilët duhet të jenë të gatshëm të pranojnë transaksione në çdo moment dhe do të ishte e papranueshme të fiket makina llogaritëse që e ekzekuton këtë server.

Kjo nuk do të thotë se nuk duhet pasur kujdes për programe të vogla gjatë rezervimit të memories. Përkundrazi, çdo program e ka rëndësinë e vet, prandaj duhet krijuar veti të mirë punuese në programim në çfarëdo gjuhe programuese. Me një fjalë, u duhet përmbytur rregullave të gjuhës programuese.

Në praktikë është vërtetuar se në gjuhët e limituara bëhen më pak gabime në kod, mirëpo në këto gjuhë është vështirë apo e pamundshme të zgjidhen problemet komplekse. Njëra prej gjuhëve të paraqitura pas gjuhës C dhe C++ është gjuha Java. Kjo gjuhë është dizajnuar të jetë portabile në sisteme operative të ndryshme dhe të zgjidhë disa probleme të tjera të paraqitura në gjuhët e mëparshme, pra edhe problemën e memories. Në gjuhën Java memoria e rezervuar nuk duhet të lirohet pasi që gjuha Java krijon një proces që kujdeset për lirimin e memories. Kjo është veti shumë e mirë, mirëpo ka

edhe anën negative të saj që shkakton ngadalësimin e programeve ekzekutuese.

Edhe pse gjuha C++ nuk kujdeset në mënyrë automatike për lirimin e memories së rezervuar gjatë ekzekutimit të programit, ajo e mundëson këtë me anë të të ashtuquajturve "tregues të mençur". Treguesit e mençur janë klasë të cilat i "mbështjellin" treguesit e vërtetë dhe i implementojnë operatorët e treguesve. Pasi që treguesit e mençur kanë të njëjtin funksion për çfarëdo tip, atëherë do të ishte e udhës ta krijojmë një klasë shabllon që vlen për tregues të çfarëdo tipi.

Klasa që i impementon treguesit e mençur kujdeset për lirimin e memories së rezervuar në mënyrë dinamike.

Nëse i kujtoni treguesit e thjeshtë, do të vëreni se dy apo më shumë tregues mund të tregojnë në të njëjtën memorie. Nëse e lirojmë memorien e treguar nga njëri prej këtyre treguesve, atëherë treguesit e tjerë nuk do të mund ta përdorin memorien e treguar më parë. Pasi që vetë përdoruesit e këtyre treguesve vendosin për lirimin e memories, atëherë është edhe detyrë e vetë përdoruesve të treguesve të mos i përdorin treguesit që tregojnë në memorien e liruar më parë.

Me treguesit e mençur problemi qëndron ndryshe. Pasi që këta tregues kujdesen vetë për lirimin e memories së rezervuar shtrohet pyetja:

"Çka nëse dy apo më shumë tregues të mençur tregojnë në memorie të njëjtte? Cili tregues i mençur e liron memorien? Çka bëhet nëse të gjithë treguesit e mençur që tregojnë në të njëjtën memorie tentojnë ta lirojnë memorien?"

Së pari, nëse tentojmë ta lirojmë memorien e liruar më parë, natyrisht se programi do të përfundojë me gabim. Sa i përket asaj se cili tregues e liron memorien, do të ishte e udhës që treguesi i fundit të kujdeset për lirimin e memories. Kjo arrihet duke e mbajtur një numrues në klasën që e mbështjell treguesin. Ky numrues numron se sa tregues tregojnë në të njëjtën memorie. Kështu, sa herë një tregues i mençur tenton ta lirojë memorien e treguar, numruesi zbritet për një. Nëse numruesi është i barabartë me zero, atëherë kjo do të thotë se treguesi i fundit tenton ta lirojë memorien dhe në këtë rast do të mund ta lirojmë memorien.

Në kodin e mëposhtëm i kemi definuar dy klasë Numruesi dhe Treguesi. Klasa Numruesi përmban një integer për t'i numëruar treguesit e memories, kurse klasa Treguesi është klasë që i mbështjell treguesit e klasëve të prejardhura nga klasa Numruesi. Pra klasa Treguesi e krijon treguesin e mençur që kujdeset për lirimin e memories. Si shembull shihni më poshtë funksionin main dhe si nuk e lirojmë memorien e rezervuar për klasën Shembull.

```

////////////////////////////////////
// fajlli Treguesi
// implementimi i treguesit të mençur

#ifndef Treguesi_h
#define Treguesi_h

class Numruesi
{
public:
    Numruesi() { m_iNumriItreguesve = 0; }
    void Shto() { m_iNumriItreguesve++; }
    void Zvogelo()
    {
        if (--m_iNumriItreguesve == 0)
        {
            // treguesi i fundit, liro memorien
            delete this;
        }
    }

private:
    int m_iNumriItreguesve;
};

template < class T >
class Treguesi
{
public:
    Treguesi(T* p) : m_p(p)
    {
        m_p->Shto(); // shto numruesin për një
    }

    ~Treguesi()
    {
        m_p->Zvogelo(); // zvogelo numruesin për një
    }

    operator T*() { return m_p; }
    T& operator* () { return *m_p; }
    T* operator-> () { return m_p; }

    Treguesi& operator=(Treguesi<T>& p)
    {
        return operator=( (T*) p );
    }
}

```

```
Treguesi& operator=(T* p)
{
    // zvogelo numruesin e meparshem
    m_p->Zvogelo();

    m_p = p;

    // shto numrin e numruesit te ri
    m_p->Shto();

    return *this;
}
private:
    T* m_p;
};

#endif
```

```

////////////////////////////////////
// fajlli main.cpp

#include <iostream>
#include "Treguesi.h"

using namespace std;

class Shembull : public Numruesi
{
public:
    void ShtypTungjatjeta()
    {
        cout << "Tungjatjeta!" ;
    }
};

void main ()
{
    // rezervu memorien per treguesin p1
    Treguesi<Shembull> p1 = new Shembull;

    // tani numruesi i treguesit p1 është 1

    // rezervu memorien për treguesin p2
    Treguesi<Shembull> p2 = new Shembull;

    // tani numruesi i treguesit p2 është 1

    // rreshti që vijon thërret operatorin
    // operator=(T* p) të klasës Treguesi

    p1 = p2;           // numruesi i treguesit p1 është 0,
                       // prandaj memoria e rezervuar për objektin
                       // e treguar nga ky tregues lirohet.
                       // Pastaj treguesi p1 tregon në p2
                       // dhe e rrit numruesin, i cili behet 2
                       // për të dy treguesit p1 dhe p2.
                       // Pra të dy treguesit tregojnë në
                       // të njëjtën memorie.

    p1->ShtypTungjatjeta();

    // Në mbarim të funksionit main thirret destruktori
    // për treguesit e mençur p1 dhe p2 me ç'rast numruesi
    // bëhet zero dhe lirohet memoria e treguar nga të dy
    // treguesit.
}

```

Disa kompajler të gjuhës C++ furnizojnë klasën shabllon që prezanton treguesit e mençur për çfarëdo tipi, p.sh. klasa `auto_ptr`. Kjo klasë mund të përdoret për tipet e thjeshta (si p.sh. në kodin që vijon) si dhe për tipet e krijuara nga përdoruesit.

```

////////////////////////////////////
// fajlli main.cpp

#include <iostream>
#include <memory>    // për klasën auto_ptr

using namespace std;

void main (void)
{
    int* pInt = new int;

    *pInt = 5;

    {
        auto_ptr<int> pAutoInt(pInt);
    }

    delete pInt;    // gabim
}

```

Programi i mësipërm do të përfundojë me gabim, sepse kemi tentuar ta lirojmë memorien e liruar nga treguesi i mençur në bllokun:

```

{
    auto_ptr<int> pAutoInt(pInt);
}

```

Fasi që hapësira e veprimtarisë së variablës `pAutoInt` është brenda bllokut {}, destruktori i treguesit të mençur e liron memorien e rezervuar për treguesin `pInt`, prandaj nuk kemi nevojë të lirojmë memorien. Kodi që vijon është plotësisht i rregullt dhe nuk kemi nevojë për të liruar memorien e rezervuar në mënyrë dinamike:

```

#include <memory>    // për klasën auto_ptr

void main (void)
{
    int* pInt = new int;

    auto_ptr<int> pAutoInt(pInt);

    *pAutoInt = 5;
}

```

Ushtrime

1. Ndryshoni klasën shabllon vektori të implementuar në këtë kapitull ashtu që në vend të listës lidhëse, ta përdorni tipin *array*. Kuptohet që kjo klasë do të ketë kufizim në numrin e anëtarëve. Kjo klasë duhet ta përmbajë një variabël që ta numërojë numrin e anëtarëve. Nëse përdoruesi tenton të shtojë elemente më shumë se që është e mundur në *array*, praqiteni këtë si gabim.
2. Ndryshojeni klasën vektori për ta bërë më efikase duke rezervuar memorie në mënyrë dinamike për anëtarët e objektit. Këta anëtarë duhet të ruhen në *array*. Nëse memoria e rezervuar për *array* mbushet, atëherë duhet të rezervoni më shumë memorie për një *array* tjetër më të madhe. Kopjoni anëtarët e ruajtur në objektin *array* të vjetër, në objektin *array* të ri dhe pastaj lirojeni memorien e rezervuar për objektin *array* të vjetër. P.sh. le të jetë *m_pData* variabël anëtare e klasës vektori me tip tregues në *array*. Në fillim le të fillojmë me memorie të mjaftueshme për dy elemente. Nëse tentojmë ta shtojmë elementin e tretë, atëherë ju do ta implementoni funksionin anëtar shto sikurse në pseudokodin që vijon:

```
template <class T>
void Vektori<T>::Shto(T tAnetari)
{
    if (iNumriAnetareve == m_iSasiaEMemories)
    {
        // rrite memorien e rezervuar per 10% të
        // anëtarëve të objektit Vektori
        m_iSasiaEMemories = m_iNumriAnetareve +
            (m_iNumriAnetareve * 10/100);

        T* pData = new T[m_iSasiaEMemories];

        // kopjoi anëtarët e objektit të vjetër
        for (int i=0; i < m_iNumriAnetareve; i++)
        {
            pData[i] = m_pData[i];
        }

        pData[m_iNumriAnetareve] = tAnetari;

        // liroje memorien për objektin e vjetër
        delete m_pData;

        // trego në memorien e re
        m_pData = pData;
    }
    else
    {
```



```
m_pData[m_iNumriAnetareve] = tAnetari;  
iNumriAnetareve++;  
}
```

3. Krijoheni një klasë të re Perseritesi për klasën vektorit. Krahasoheni klasën Perseritesi të klasës së vjetër vektorit me këtë klasë.
4. Krijoheni një klasë shabllon mbajtëse që i mban anëtarët në renditje. D.m.th. sa herë që e shtoni një anëtar në listë, duhet t'i renditni anëtarët e tjerë dhe t'ia gjeni vendin e duhur anëtarit të ri. Përdoreni vetëm operatorin < për krahasimin e anëtarëve. Kjo do të thotë se klasa shabllon do të vlejë për të gjitha tipet që e implementojnë operatorin <.

Përmbledhje

Shabllonet mundësojnë krijimin e klasëve apo të funksioneve të përgjithshme të bazuara në tipet e furnizuara si parametra. Klasët apo funksionet shabllon operojnë në tipe të ndryshme, kështu që i shmangemi krijimit të klasëve apo funksioneve të njëjta për çdo tip të përdorur në program.

Gjuha C++ ofron, nëpërmjet librave standarde, klasët shabllon të njohura si klasë mbajtëse që e lehtësojnë zgjidhjen e shumë problemeve. Libraria që i përmban klasët shabllon njihet si libraria STL (akronim për Standard Template Library).

Operatorët cast

Në kapitullin e parë (1.4.4) kemi folë për operatorin cast. Ky operator, siç kemi përmendur edhe më parë, është shumë i përdorshëm në gjuhën C, për shndërrimin e një tipi në tip tjetër. Operatori cast ka edhe mangësitë e veta, p.sh. gjatë përdorimit të parregullt ky operator në të shumtën e rasteve shkakton përfundimin e programit pa dëshirë.

Operatori cast përdoret edhe në gjuhën C++, mirëpo gjuha C++ mundëson metoda të tjera më të sigurta që e eliminonin nevojën për operatorin cast, p.sh. me përdorimin e klasëve dhe funksioneve virtual në klasët trashëguese.

Shndërrimi i një tipi në tip tjetër në disa tipe elementare bëhet në mënyrë indirekte edhe nga vetë kompajleri, p.sh.:

```
float fData = 2.5;
int    iNumri = fData;
```

Variabla iNumri e merr vlerën 2, pra vlera e variablës fData shndërrohet në mënyrë indirekte në tip int duke humbur vlerën pas presjes dhjetore. Disa kompajlerë e paraqesin rreshtin

```
int    iNumri = fData;
```

si vërejtje (në angl. mbas kompajlmit kemi *warning*): "Mund të vijë te humbja e të dhënave gjatë shndërrimit të tipit float në int."

Kuptohet që vërejtjet nuk e ndalojnë kompajlimin e kodit, mirëpo janë shumë të vlefshme për gjetjen e gabimeve dhe nuk duhet të injorohen. Nëse jeni të sigurtë se dëshironi ta iniconi variablën iNumri të tipit int me variablën fData të tipit float atëherë përdoreni operatorin cast për mënjanimin e vërejtjes, p.sh.:

* Në gjuhën angleze (e edhe në programim) përdoret pika për ndarjen e pjesës decimale të numrave realë, kurse presja për lëktësimin e leximit të qindësive të numrat shumëshifrorë.

```
int iNumri = (int) fData;
```

Në gjuhën C++ mund ta përdorni operatorin cast edhe kështu:

```
int iNumri = int(fData);
```

ku `int (fData)` është sikurse përdorimi i konstruktorit shndërrues.

Në disa raste, shndërrimi i një tipi në tip tjetër bëhet në mënyrë indirekte nga kompajleri pa vërejtje (në disa kompajler) edhe pse ky shndërrim mund ta shkaktojë humbjen e të dhënave. P.sh. vështroni programin që vijon:

```
#include <stdio.h>

unsigned char ShnderroNe_char(int iVlera)
{
    unsigned char ch = iVlera;
    return ch;
}

void main ()
{
    printf("Vlera 1001 e shndërruar ne unsigned char: %d\n",
        ShnderroNe_char (1001));
}
```

Funksioni `ShnderroNe_char` e shndërron tipin `int` në tipin `unsigned char`. T'ju përkujtojmë se tipi `int` në sistemet operative 32 bit është 32 bit, kurse tipi `unsigned char` është 8 bit. Kuptohet që pasi sasia e memories për tipin `unsigned char` është më e vogël sesa sasia e memories për `int`, kompajleri e "shkurton" vlerën e tipit `int` për ta ruajtur në tip më të vogël (tipin `unsigned char`). P.sh. programi i mësipër në kompajlerin e VC++ shtyp këtë fjali:

```
"Vlera 1001 e shndërruar ne unsigned char: 233"
```

Pra vlera 1001 është zvogëluar në 233. Natyrisht se shndërrimi i tipeve të mëdha në tipe të tjera nuk është aspak i preferueshëm, pasi që rezultati i këtij shndërrimi është në të shumtën e rasteve i mvarur nga sistemi operativ apo nga vetë kompajleri. Duhet cekur se në kompajlerin që është përdorë për testim në programin e mësipërm, tipi `int` shndërrohet në `unsigned char` në mënyrë indirekte nga kompajleri pa ndonjë vërejtje. Kurse gjatë shndërrimit të tipit `float` në `int`, kompajleri ka dhënë vërejtje.

Pra, disa shndërrime të një tipi në tip tjetër bëhen në mënyrë indirekte nga kompajleri, me ç'rast në disa shndërrime kompajleri jep vërejtje.

Disa shndërrime të një tipi të thjeshtë në tip tjetër, kompajleri i paraqet si gabim dhe e përfundon kompajlimin e programit pa e krijuar programin ekzekutues, p.sh. në funksionin shnderro kompajleri i gjuhës C++ e përfundon kompajlimin me gabim:

```
int* Shnderro(void* pVoid)
{
    return pVoid;
}
```

Nëse e kompajloni funksionin e mësipërm me kompajler të gjuhës C, atëherë asnjë gabim nuk do të paraqitet. Për ta kompajluar fajllin me kompajler të gjuhës C mund ta përdorni kompajlerin e gjuhës C++, mirëpo fajlli duhet ruajtur me prapashtesën .c.

Për ta kompajluar funksionin shnderro me kompajler të gjuhës C++, patjetër duhet përdorë operatorin cast (int*) për t'i treguar kompajlerit se me të vërtetë dëshironi ta shndërroni tipin void* në int*, p.sh.:

```
int* Shnderro(void* pVoid)
{
    return (int*) pVoid;
}
```

Kjo vërteton se gjuha C++ është kontrolluese më e fortë se gjuha C dhe se disa shndërrime të tipeve të bëra në mënyrë indirekte në gjuhën C nuk janë të lejueshme në gjuhën C++.

Gjuha C++, në krahasim me gjuhën C, përmban klasët që mundësojnë krijimin e tipeve të reja të ngjashme me tipet e ofruara nga vetë kompajleri. Këto klasë mund të përmbajnë funksione virtual apo t'i trashëgojnë funksionet nga klasët e tjera. Për arsye se gjuha C++ është më e komplikuar se gjuha C, operatori cast i përdorur në gjuhën C, duhet përdorë me kujdes në gjuhën C++. Për ta lehtësuar shndërrimin e një tipi në tip tjetër, gjuha C++ ofron katër lloje të operatorit cast, të cilët përdoren në situata të ndryshme. Këta operatorë janë:

- ♦ static_cast
- ♦ dynamic_cast
- ♦ const_cast
- ♦ reinterpret_cast

Tani do ta shikojmë funksionin e secilit operator cast.

12.1 Operatori cast static_cast

Për ta shpjeguar operatorin `static_cast` kemi krijuar dy klasë, klasën `KlasaA` dhe klasën `KlasaB` e cila trashëgohet nga klasa `KlasaA`.

```
class KlasaA
{
public:
    void Funksioni_1();
};

class KlasaB : public KlasaA
{
public:
    void Funksioni_2();
};
```

Tani të supozojmë se i kemi dy funksione të cilat e përdorin njëri prej klasëve të definuara më parë:

```
void f(void* pVoid)
{
    //Ky funksion bën diçka me treguesin pVoid
}

void g()
{
    KlasaB b;
    f(&b);
}
```

Siç e vëreni, në funksionin `g()` e kemi krijuar një objekt të tipit `KlasaB` si dhe e kemi pasuar adresën e objektit `b` në funksionin `f` që e pranon tregues të tipit `void`. Me siguri ju do të pyetni "Si është e mundur ta pasojmë adresën e objektit të tipit `KlasaB`, kur funksioni `f` pret tip tregues të tipit `void`?". Më parë kemi përmendur se disa shndërrime të tipeve bëhen në mënyrë indirekte nga vetë kompajleri. Shndërrimi i adresës së çfarëdo objekti në tregues të tipit `void` bëhet në mënyrë indirekte në kompajlerin e gjuhës C dhe C++, pra asnjë gabim nuk është lajmëruar nga kompajleri C++ edhe pse nuk e kemi përdorë operatorin `cast`.

Nëse e kujtoni polimorfizmin në kapitullin 9, do ta vëreni se shndërrimi i tipit tregues të ndonjë klase në tip tregues të klasës bazë, po ashtu bëhet në mënyrë indirekte në gjuhën C++ pa pasur nevojë të përdoret operatori `cast`. P.sh. kodi që vijon kompajlohet pa ndonjë gabim apo vërejtje edhe pse në funksionin `f2`

e kemi pasuar adresën e objektit `KlasaB`, kurse funksioni `f2` pret tregues të tipit `KlasaA`:

```
void f2(KlasaA* pKlasaA)
{
    //ky funksion bën diçka me treguesin pKlasaA
}

void g2()
{
    KlasaB b;
    f2(&b);
}
```

Duhet cekur se ky shndërrim është i lejueshëm vetëm nëse trashëgimi është `public`. Nëse klasa `KlasaB` trashëgohet nga klasa `KlasaA` në njërin prej metodave të paraqitura më poshtë, atëherë kompajleri nuk do të mund ta shndërrojë në mënyrë indirekte apo direkte (me përdorimin e operatorit `cast`) adresën e objektit të tipit `KlasaB` në tregues të tipit `KlasaA`.

```
class KlasaB: private KlasaA
{ ...
};

// apo

class KlasaB : protected KlasaA
{ ...
};
```

Shndërrimi i adresës së ndonjë tipi në tregues të klasës bazë nuk është i lejueshëm as në raste kur ai tip ka prejardhje nga më shumë se një klasë në hierarki. Për këtë do të flasim më vonë.

Nëse trashëgimi i tipit `KlasaB` është `public` atëherë shndërrimi i treguesit të klasës `KlasaB` apo adresës së objektit të tipit `KlasaB` në tregues të tipit `KlasaA` konsiderohet si shndërrim i sigurtë, pasi që klasa `KlasaB` përmban të gjitha variablat apo funksionet anëtare të cilat i përmban `KlasaA`.

Shndërrimi i tipit në drejtim në kundërt, p.sh. i adresës së objektit të tipit `KlasaA` në tregues të tipit `KlasaB`, nuk është i lejueshëm. Po ashtu, as shndërrimi i treguesit të tipit `void` në tregues të ndonjë tipi tjetër nuk është i lejueshëm:

```
void f3(KlasaB* pKlasaB)
{
```

```

        // ky funksion bën diçka me treguesin pKlasaB
    }

    void g3(void* pVoid)
    {
        KlasaA a;
        f3(a);      // nuk eshte e lejueshme
        f3(pVoid);  // nuk eshte e lejueshme
    }

```

Arsyeja për këtë është sepse kompajleri nuk mund të garantojë se treguesi i tipit `void` ka ndonjë lidhshmëri me tip tjetër, apo treguesi i klasës bazë ka ndonjë lidhshmëri me tipin e trashëguar. D.m.th. kompajleri nuk mund të garantojë se tipi i klasës bazë i ka të gjitha funksionet anëtare, apo variablat anëtare sikur ato të tipit të klasës së trashëguar.

Mirëpo, nëse jeni të bindur se treguesi i klasës bazë tregon në tip të klasës së trashëguar apo në tregues të tipit `void`, atëherë mund t'i tregoni kompajlerit ta bëjë shndërrimin e tipit me anë të operatorit `static_cast`, p.sh.:

```

void f4(KlasaB* pKlasaB)
{
    // ky funksion bën diçka me treguesin pKlasaB
}

void g4()
{
    KlasaB      b;

    void*        pVoid      = &b;
    KlasaA*      pKlasaA    = &b;

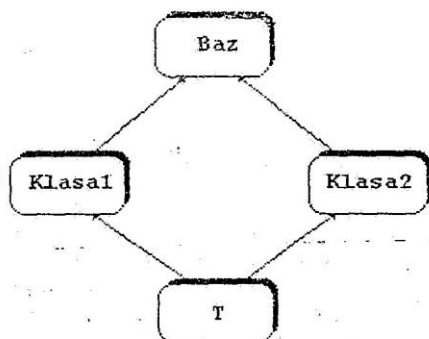
    f4(static_cast<KlasaB*>(pVoid));
    f4(static_cast<KlasaB*>(pKlasaA));
}

```

Pasi që në kodin e mësipërm kemi qenë të sigurtë se treguesit `pvoid` dhe `pKlasaA` tregojnë në tip të klasës `KlasaB`, e kemi përdorë operatorin `cast static_cast` për t'i treguar kompajlerit të mos e paraqesë ndonjë gabim në shndërrimin e tipit.

Operatori `cast static_cast` nuk mund të përdoret për shndërrimin e treguesit të ndonjë tipi të trashëguar me udhëzimin `virtual` në tregues të tipit bazë. Për këtë shndërrim duhet përdorë operatorin `cast dynamic_cast` (shih operatorin `dynamic_cast`).

Tani ta shikojmë një shndërrim më të komplikuar të tipeve në hierarki. Për ta ilustruar problemin, këto klasë i kemi definuar pa ndonjë funksion anëtar apo variabël anëtare.



Klasët Klasa1 dhe Klasa2 janë të trashëguara nga klasa Baze, kurse klasa T është e trashëguar nga të dy klasët Klasa1 dhe Klasa2.

```

class Baze[ ];

class Klasa1 : public Baze [ ];
class Klasa2 : public Baze [ ];

class T : public Klasa1, public Klasa2 [ ];

void main ()
{
    T          t;

    Klasa1*     pKlasa1 = &t; // pa problem
    Klasa2*     pKlasa2 = &t; // pa problem

    Baze*       pBaze = &t;   // gabim
}
  
```

Në kodin e mësipërm, kompajleri bën shndërrimin në mënyrë indirekte prej treguesit të tipit T (adresës së variablës t) në tregues të tipit Klasa1 dhe në tregues të tipit Klasa2. Mirëpo, kompajleri nuk mund ta shndërrojë treguesin e tipit T në tregues të tipit Baze për arsye se tipi T ka dy nënklasë të tipit të prejardhur nga tipi Baze. Gjuha C++ nuk lejon këtë shndërrim për arsye se ky shndërrim është i paqartë dhe kompajleri nuk mund të vendos si ta shndërrojë treguesin e tipit T në tregues të tipit Baze. Edhe po ta përdorni operatorin cast në mënyrë direkte për t'i treguar kompajlerit për shndërrim, p.sh.:


```
Baze* pBaze = (Baze*). &t; // gabim
```

prapë kompajleri do ta paraqesë si gabim.

Problemin e mësipërm mund ta zgjidhim me përdorimin e operatorit `static_cast`, p.sh.:

```
Baze* pBaze = static_cast<Klasa1*>(t);
```

```
// apo
```

```
Baze* pBaze = static_cast<Klasa2*>(t);
```

Ky shndërrim është i vlefshëm për arsye se klasët `Klasa1` dhe `Klasa2`, nënklasë të klasës `T` kanë vetëm një nënklasë `Baze`. Operatori `static_cast` i tregon kompajlerit që treguesin e tipit `T` ta shndërrojë në tregues të tipit `Klasa1` apo `Klasa2` dhe pastaj kompajleri në mënyrë indirekte e shndërron tipin e treguesit `Klasa1` apo `Klasa2` në tregues të tipit `Baze`.

Pra operatorin `static_cast` duhet ta përdorni për shndërrimin e treguesit të tipit të klasës bazë në tregues të tipit të klasës së trashëguar, kur jeni të sigurtë që treguesi i tipit bazë tregon në tip të klasës së trashëguar. Operatorin `static_cast` e përdorni edhe kur ta shndërroni treguesin e tipit që ka prejardhje prej më shumë se një klase, në tip të klasës bazë.

12.2 Operatori cast dynamic_cast

Përdorimi i operatorit static_cast bëhet me vetëdije dhe me rrezikim të vetë përdoruesit. D.m.th. kompajleri nuk garanton se shndërrimi i një tipi në tip destinues është i vlefshëm. Me përdorimin e operatorit static_cast kompajleri beson se programuesi është i vetëdijshëm për shndërrimin e tipit dhe nuk shikon për vlefshmërinë e këtij shndërrimi. Nëse nuk jeni të sigurtë që shndërrimi i një tipi në tip tjetër është i vlefshëm, atëherë duhet t'i shmangeni përdorimit të operatorit static_cast. Në krahasim me gjuhën C, operatori cast mund të zëvendësohet në disa mënyra në gjuhën C++, duke dizajnuar programin në mënyrë objekt-orientuese. Në raste kur operatori cast është i domosdoshëm, përdorimi i këtij operatori mund të lokalizohet.

Gjuha C++ ofron operatorin cast dynamic_cast, i cili e shikon tipin e objektit gjatë ekzekutimit të programit. Pra shikon tipin e objektit në mënyrë dinamike, në krahasim me operatorin cast static_cast i cili përdoret vetëm gjatë kompajlimit të programit.

Nëse operatori cast dynamic_cast nuk mund ta shndërrojë një tip në tipin destinues, atëherë ky operator kthen null. Për ilustrim shihni klasët e definuara në kodin që vijon:

```
class KlasaA
{
public:
    virtual void Funksioni() {}
};

class KlasaB : public KlasaA
{
};

class KlasaC : public KlasaA
{
};

KlasaB* Shnderro(KlasaA* pKlasaA)
{
    return dynamic_cast<KlasaB*>(pKlasaA);
}

void main ()
{
```

```

KlasaB          b;
KlasaC          c;

KlasaA*          pKlasaA;
KlasaC*          pKlasaC;

pKlasaA          = &b;
pKlasaC          = &c;

KlasaB*          p1 = Shnderro(pKlasaA);
KlasaB*          p2 = Shnderro(pKlasaC);

if (p1 == NULL)
{
    cout << "Treguesi p1 nuk është i vlefshëm" << endl;
}

if (p2 == NULL)
{
    cout << "Treguesi p2 nuk është i vlefshëm" << endl;
}

```

Para se ta shpjegojmë kodin e mësipërm, duhet cekur se operatori vlen vetëm për klasët që kanë së paku një funksion virtual, përndryshe ky operator nuk vlen apo kompajleri nuk e kompajlon programin. Në disa kompajlerë për ta kompajluar programin e mësipërm duhet ta mundësoni opcionin për informata dinamike për tipe të definuara (në angl: *"Enable Run-Time Type Information"*).

Tani t'i kthehemi programit të mësipërm. Në klasën KlasaA kemi definuar funksionin anëtar virtual Funksioni për ta mundësuar përdorimin e operatorit cast dynamic_cast. Për ta ilustruar problemin i kemi deklaruar dy variabla, njëra e tipit KlasaB (variabla b) dhe tjetra e tipit KlasaC (variabla c). Kemi deklaruar edhe tregues të tipit KlasaA (pKlasaA) që tregon në variablën b dhe tregues të tipit KlasaC (pKlasaC) që tregon në variablën c.

Tentimi për shndërrimin e treguesit të tipit KlasaA (mirëpo që tregon në tipin KlasaB) në tregues të tipit KlasaB është i suksesshëm, d. m. th. treguesi p1 është i ndryshëm prej NULL dhe fjalia *"Treguesi p1 nuk është i vlefshëm"* nuk shtypet në monitor. Mirëpo tentimi i shndërrimit të treguesit të tipit KlasaC në tregues të tipit KlasaB nuk është i suksesshëm për arsye se tipi KlasaC nuk ka prejardhje nga tipi KlasaB (d.m.th. nuk ka kurrfarë relacioni në mes të këtyre tipeve). Në këtë rast, operatori cast dynamic_cast kthen NULL dhe shprehja p2 == NULL rezulton e saktë, prandaj programi shtyp fjalinë *"Treguesi p2 nuk është i vlefshëm"*.

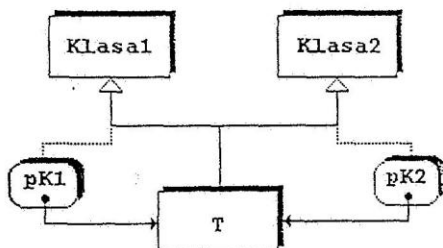
Operatori `cast dynamic_cast` mund të përdoret edhe për shndërrimin e referencës së një tipi në referencë të tipit tjetër. Mirëpo, pasi që në gjuhën C++ nuk ekziston referenca NULL, atëherë në rastet kur shndërrimi i referencës së një tipi në referencë të tipit tjetër nuk është i vlefshëm, operatori `dynamic_cast` krijon një përjashtim të tipit `std::bad_cast` (për përjashtimet do të flasim në kapitullin e ardhshëm). Kodi që vijon demonstroi shndërrimin e referencës së një tipi në referencë të tipit tjetër duke marrë parasysh klasët e definuara më parë.

```
void main ()
{
    KlasaB b;

    try
    {
        KlasaA& a = dynamic_cast<KlasaA&> (b);
    }
    catch(std::bad_cast)
    {
        cout << "Operatori dynamic_cast ka shkaktuar"
              << " përjashtim në shndërrimin e referencës "
              << " së tipit KlasaB në referencë të tipit "
              << " KlasaA " << endl;
    }

    try
    {
        KlasaC& c = dynamic_cast<KlasaC&> (b);
    }
    catch(std::bad_cast)
    {
        cout << "Operatori dynamic_cast ka shkaktuar"
              << " përjashtim në shndërrimin e referencës "
              << " së tipit KlasaB në referencë të tipit "
              << " KlasaC " << endl;
    }
    catch(...)
    {
        // kjo është e nevojshme për disa kompajlerë.
    }
}
```

Kur folëm për operatorin `cast static_cast` treguam se ky operator nuk mund të përdoret për shndërrimin e treguesit të tipit të klasës virtual bazë në tregues të tipit të trashëguar. Ky lloj shndërrimi mund të bëhet vetëm në mënyrë dinamike (jo gjatë kompajlimit), pra me anë të operatorit `cast dynamic_cast`.



```
pK2 = dynamic_cast<Klasa2*> (pK1)
```

Pra siç e vërejmë, treguesi pK1 është në degë tjetër të hierarkisë ndaj treguesit pK2, mirëpo operatori `dynamic_cast` arrin ta shndërrojë treguesin pK1 në tregues të tipit `Klasa2*` për arsye se treguesi pK1 tregon në objekt të tipit T. Nëse treguesi pK1 do të tregonte në objekt të tipit `Klasa1` atëherë:

```
dynamic_cast<Klasa2*> (pK1)
```

do të kthente `NULL`.

Operatori `dynamic_cast` mund të përdoret edhe për të treguar në fillim të objektit, duke e përdorur treguesin e tipit `void` (`void*`), pra `dynamic_cast<void*>`. P.sh. në kodin e mëposhtëm treguesi pVoid tregon në fillim të objektit t të tipit T.

```

Klasa1*   pK1 = &t;
Klasa2*   pK2 = dynamic_cast<Klasa2*> (pK1);

void*     pVoid = dynamic_cast<void*> (pK1);
  
```

Nëse kemi tregues të tipeve të ndryshme që tregojnë të gjithë në të njëjtin objekt në hierarki, atëherë mund ta përdorim operatorin `dynamic_cast` për të shikuar nëse treguesit tregojnë në të njëjtin objekt, p.sh.:

```

void main ()
{
    T      t;
    Klasa1* pK1 = &t;
    Klasa2* pK2 = dynamic_cast<Klasa2*> (pK1);

    void*  pVoid1 = dynamic_cast<void*> (pK1);
    void*  pVoid2 = dynamic_cast<void*> (pK2);
}
  
```

```

    if (pVoid1 == pVoid2)
    {
        cout << "Treguesi pVoid1 tregon në të "
              "njëjtin objekt sikurse treguesi "
              "pVoid2";
    }
}

```

Kodi i mësipërm demonstron se si dy tregues të tipeve të ndryshme mund të testohen nëse tregojnë apo jo në të njëjtin objekt. Kodi i mësipërm mund të shkruhet edhe pa deklarimin e treguesve void:

```

void main ()
{
    T      t;
    Klasa1* pK1 = &t;
    Klasa2* pK2 = dynamic_cast<Klasa2*> (pK1);

    if (dynamic_cast<void*> (pK1) == dynamic_cast<void*> (pK2))
    {
        cout << "Treguesi pK1 tregon në të njëjtin objekt "
              "sikurse treguesi pK2";
    }
}

```

Për të vërtetuar se me të vërtetë nevojitet la përdorim operatorin `dynamic_cast` për të shikuar se dy tregues të tipeve të ndryshme tregojnë në të njëjtin objekt, e kemi përdorë programin që vijon:

```

void main ()
{
    T      t;
    Klasa1* pK1 = &t;
    Klasa2* pK2 = dynamic_cast<Klasa2*> (pK1);

    void*   pVoid1 = dynamic_cast<void*> (pK1);
    void*   pVoid2 = dynamic_cast<void*> (pK2);

    if (pVoid1 == pVoid2)
    {
        cout << "Treguesi pVoid1 tregon në të "
              "njëjtin objekt si treguesi "
              "pVoid2";
    }

    if (pK1 == reinterpret_cast<Klasa1*> (pK2))
    {
        cout << "Treguesi pK1 tregon në të njëjtin "

```

```
"objekt si treguesi pK2";
```

Programi i mësipërm shtyp vetëm fjalinë:

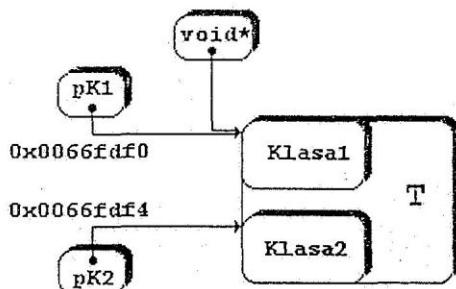
Treguesi pVoid1 tregon në të njëjtin objekt si treguesi pVoid2

Ju me siguri do të pyetni :

"Pse programi nuk shtyp fjalinë e dytë?" apo "Pse treguesit pK1 dhe pK2 nuk janë të barabartë kur të dy treguesit tregojnë në të njëjtin objekt?"

Nga programi i mësipërm vërejmë se treguesit pK1 dhe pK2 tregojnë në objekt të tipit T, me ç'rast të dy treguesit tregojnë në të njëjtin objekt. Pasi tipi T është i prejardhur nga tipi Klasa1 dhe Klasa2 të dy treguesit pK1 dhe pK2 janë të vlefshëm, mirëpo secili tregues tregon në adresë të ndryshme të objektit t. Kurse, siç përmendëm më parë, treguesi i shndërruar me operatorin `dynamic_cast<void*>` tregon në fillim të adresës së objektit, prandaj mund të përdoret për testim në tregues të tipeve të ndryshme.

Për ta kuptuar problemin, shihni figurën që vijon si dhe kodin që shtyp adresat e çdo treguesi:



```

void main ()
{
    T      t;
    Klasa1* pK1 = &t;
    Klasa2* pK2 = dynamic_cast<Klasa2*> (pK1);

    void*   pVoid1 = dynamic_cast<void*> (pK1);
    void*   pVoid2 = dynamic_cast<void*> (pK2);
  
```



```
cout << "Adresa e treguesit pVoid1 është: "
      << pVoid1 << endl;

cout << "Adresa e treguesit pVoid2 është: "
      << pVoid2 << endl << endl;

cout << "Adresa e treguesit pK1 është: " << pK1 << endl;

cout << "Adresa e treguesit pK2 është: " << pK2 << endl;
}
```

Ky program shtyp këtë rezultat:

Adresa e treguesit pVoid1 është: 006AFDF0

Adresa e treguesit pVoid2 është: 006AFDF0

Adresa e treguesit pK1 është: 006AFDF0

Adresa e treguesit pK2 është: 006AFDF4

Pra adresa e treguar nga treguesit pK1 dhe pK2 nuk është e njëjtë, prandaj në programin e mëparshëm udhëzimi

```
pK1 == reinterpret_cast<Klasa1*>(pK2)
```

nuk është e saktë.

Në fund të përsërisim edhe një herë se operatori `dynamic_cast` mund të përdoret për shndërrimin e tipeve gjatë ekzekutimit të programit dhe për ta parë vlefshmërinë e shndërrimit gjatë shndërrimit të treguesit të klasës virtual bazë në tregues të klasës së prejardhur, si dhe në gjetjen e fillimit të adresës së ndonjë objekti.

12.3 Operatori cast const_cast

Operatori cast `const_cast` përdoret për t'ia hequr vetinë `const` një tipi apo treguesi të tipit të caktuar. Ky operator i tregon kompajlerit ta pranojë shndërrimin e tipit konstant në tip jokonstant. Shihni kodin në vazhdim:

```
int          iNumri = 5;
const int*   pKonstantInt = &iNumri;
```

Nëse tentoni të deklaroni tregues të tipit `int`, jokonstant dhe ta iniconi me treguesin konstant të tipit `int`, atëherë kompajleri do ta paraqesë si gabim, p.sh.:

```
int*   p = pKonstantInt; // gabim
```

Për t'iu shmangur këtij gabimi mund ta përdorim operatorin `cast const_cast` i cili i tregon kompajlerit se me të vërtetë kemi menduar ta shndërrojmë këtë tip dhe se jemi të gatshëm ta marrim përgjegjësinë për vlefshmërinë e këtij shndërrimi, p.sh.:

```
int*   p = const_cast<int*>(pKonstantInt);
```

Operatori cast `const_cast` duhet të përdoret me shumë kujdes sepse në të shumtën e rasteve efekti i programit merr tjetër kalje dhe pasojat janë të paparashikueshme. P.sh. ta marrim rastin e librarive standarde, të cilat publikojnë interfejsin e funksioneve (prototipin e funksioneve) dhe të klasëve të ndryshme, kurse kodi implementues është i fshehur (i kompajluar në kod binar). Ta zëmë se libreria standarde ofron një funksion për llogaritjen e katrorit të një numri me këtë prototip:

```
int KalkuloKatrorin(const int*);
```

Në shikim të parë të funksionit të mësipërm, do të jemi të sigurtë se ky funksion nuk e ndryshon vlerën e vet hyrëse. Mirëpo, çka nëse implementuesi i funksionit `KalkuloKatrorin` e ka përdorë operatorin `cast const_cast` brenda në këtë funksion dhe e ka ndryshuar vlerën e treguar nga treguesi? Ta zëmë se ky funksion është implementuar në librarë standarde sikurse kodi në vijim, kurse ne si përdorues të librarisë standarde e kemi në dispozicion vetëm prototipin e funksionit.

```
int KalkuloKatrorin(const int* pInt)
{
```

```

int* p = const_cast<int*> (pInt);
*p      = 5;

return *pInt * *pInt;
}

```

Atëherë, programi në vijim jep rezultat të papritur edhe pse jemi të sigurtë se kodi në funksionin main është i rregullt.

```

void main ()
{
    int          iNumri      = 3;
    int          iNumri2    = 2;

    cout << KalkuloKatrorin(&iNumri) << endl;
    cout << KalkuloKatrorin(&iNumri2) << endl;

    // ...
    // disa kalkulime të tjera të cilat i përdorin variablat
    // iNumri dhe iNumri2

    cout << iNumri + iNumri2;

    // ...
}

```

Programi i mësipërm shtyp numrin 25 dy herë, në vend se t'i shtypë numrat 9 dhe 4. As kalkulimet pas kalkulimit të katrorit nuk e japin rezultatin e pritur.

Çili është problemi?

Gjatë gjetjes së gabimeve do të vërejmë se dy rreshtat :

```

cout << KalkuloKatrorin(&iNumri) << endl;
cout << KalkuloKatrorin(&iNumri2) << endl;

```

shtypin numrin 25 dhe 25. Ky rezultat na bën të dyshojmë se funksioni KalkuloKatrorin nuk e llogarit katrorin e numrit si duhet. Mirëpo, si është e mundur që rreshti:

```
cout << iNumri + iNumri2;
```

ta shtypë numrin 10, kur $3 + 2$ bëjnë 5. Nëse e shikojmë prototipin e funksionit KalkuloKatrorin do të jemi të bindur se ky funksion nuk ndërron vlerën e treguar nga treguesi i pasuar në funksion, pasi që merr si parametër tregues const.

Atëherë si do të mund ta gjejmë problemin?

Problemin e saktë nuk mund ta gjejmë nëse nuk kemi akses në implementimin e klasëve apo funksioneve të ndryshme, mirëpo problemin mund ta izolojmë

duke e përcjellë ecurinë e programit dhe ndryshimin e vlerave. P.sh. në këtë rast, nëse do t'i komentonim rreshtat:

```
// cout << KalkuloKatrroren(siNumri) << endl;  
// cout << KalkuloKatrroren(siNumri2) << endl;
```

atëherë rezultati i kalkulimeve në kodin vijues do të ishte në rregull dhe do të ishim në gjendje të konkludojmë se problemi qëndron në funksionin KalkuloKatrroren.

Problemi i paraqitur më sipër është problem artificial, i thjeshtësuar për ta ilustruar rrezikun e shndërrimit të tipeve konstanta në jokonstanta.

Në jetë të përditshme problemet janë më të komplikuar, projektet janë të mëdha dhe në to punojnë disa programerë. Nëse nuk i përmbahemi rregullave të programimit, atëherë sistemi do të ketë shumë gabime dhe gjetja e këtyre gabimeve do të ishte shumë e vështirë. Disiplina për përmbajtjen e rregullave në kodim shkakton që kodi të jetë më i lexueshëm dhe me më pak gabime. Njëra prej këtyre rregullave është edhe mosndryshimi i vlerave të variablave konstanta.

Pra operatori cast `const_cast` duhet të përdoret vetëm në rastet kur jeni të sigurtë që nuk tentohet të ndryshohet vlera e variablës konstante. Në disa kompajlerë të sistemeve operative, nëse tentojmë ta ndryshojmë vlerën e variablave konstante programi do të përfundojë me gabim fatal.

12.4 Operatori cast reinterpret_cast

Operatori cast reinterpret_cast përdoret për shndërrimin e një tipi në tregues të atij tipi apo tregues të tipit tjetër që nuk ka kurrfarë relacioni me tipin e parë. Prej të gjithë operatorëve të përmendur deri më tani, ky operator është më i rrezikshmi për shndërrimin e tipit dhe nuk rekomandohet të përdoret. Operatori reinterpret_cast nuk është kompatibil në kompajlerë të ndryshëm dhe nuk e garanton shndërrimin e një tipi në tip tjetër.

Operatori reinterpret_cast mund të përdoret kështu:

```
int* pInt = reinterpret_cast<int*> (0x800f)
```

Edhe pse programi që e përdor kodin e mësipërm kompajlohet pa problem, në këtë rast kompajleri nuk është në gjendje të shikojë për vlefshmërinë e adresës 0x800f. Për ta përdorur operatorin cast reinterpret_cast, duheni të jeni tejet të sigurtë se adresa apo tipi i shndërruar është i vlefshëm. D.m.th. programerët e kodit (e jo kompajleri) që përdorin reinterpret_cast janë përgjegjës për vlefshmërinë e shndërrimit të tipit.

Edhe një herë duhet cekur se nëse mendoni ta përdorni operatorin cast ndaluni e mendoni mirë a është i domosdoshëm përdorimi i tij. Tentoni ta dizajnoni programin pa pasur nevojë për operatorët cast, e nëse patjetër janë të nevojshëm këta operatorë, lokalizojeni përdorimin e tyre në funksione globale ose funksione anëtare të klasëve të caktuara.

Ushtrime

1. Gjeni disa shembuj kur mund t'i përdorni operatorët cast. Krahasoni operatorët cast në mes vete dhe mundohuni ta vëreni dallimin e këtyre operatorëve me njëri-tjetrin.
2. Shikoni shembujt e gjetur për ushtrimin 1 dhe mundohuni të gjeni zgjidhje tjetër pa përdorimin e operatorëve cast.
3. Tregoni pse operatorët cast nuk preferohen të përdoren shpesh në kodimin e programeve.
4. Shihni klasët e definuara në kodin që vijon:

```
class A {
};

class B: public A {
};
```

A është i nevojshëm përdorimi i operatorit cast për shndërrimin e treguesit të tipit B në tregues të tipit A?

P.sh.:

```
B* pB;
...

A* pA = pB;
```

5. Tregoni pse nuk nevojitet operatori cast në ushtrimin 4.
6. Duke i marrë parasysh klasët e definuara në ushtrimin 4, a është i vlefshëm shndërrimi i tipit të mëposhtëm?

```
A* pA;
...

B* pB = pA;
```

7. Tregoni arsyen pse shndërrimi i tipit në ushtrimin 6 nuk është i vlefshëm.
8. Tentoni t'i numroni rastet kur kompajleri bën shndërrimin e një tipi në tip tjetër në mënyrë indirekte, d.m.th. pa pasur nevojë ta përdorë operatorin cast.

Përmbledhje

Operatorët cast janë shumë të përdorshëm në gjuhën C për shndërrimin e përkohshëm të një tipi në tip tjetër. Këta operatorë janë të përdorshëm edhe në gjuhën C++, mirëpo gjuha C++ mundëson metoda të tjera për t'iu shmangur përdorimit të operatorëve cast, si p.sh. nëpërmjet të polimorfizmit. Pasi që përdorimi i pakujdes i operatorëve cast shkakton gabime fatale në program, këta operatorë rekomandohet të përdoren me kujdes dhe të lokalizohet përdorimi i tyre.

Gjuha C++ është gjuhë tip-kontrolluese më e fortë se gjuha C, prandaj kjo gjuhë ofron 4 lloje të operatorëve cast që përdoren për situata të ndryshme. Këta operatorë janë:

- ♦ `static_cast`
- ♦ `dynamic_cast`
- ♦ `const_cast`
- ♦ `reinterpret_cast`

Përsëritja

Në këtë kapitull do t'ju njohim me funksionet përsëritëse. Shkurt, funksionet përsëritëse janë ato funksione që e thërrasin vetveten.

13.1 Konceptet e përsëritjes

Në programim kemi dy lloj përsëritjesh, përsëritja direkte dhe përsëritja indirekte. Përsëritja direkte është kur funksionet e thirrën vetveten para përfundimit. Funksioni faktorial() në Programin 13.1 është shembull i përsëritjes direkte. Përsëritja indirekte është kur funksioni p.sh. A e thirr funksionin B i cili funksion thirr funksionin A.

Funksionet përsëritse janë shumë të përdorshme. Ato janë kompakte në krahasim me funksionet jo-përsëritse. Mirëpo funksionet përsëritse e kanë dopsin e tyre sepse iu nevojitet më shumë memorie në stak si dhe janë më të ngadalshme në ekzekutim.

13.2 Metoda përçaj-e-sundo dhe përsëritja

Për zgjidhjen e problemit në funksionet përsëritse është e njohur metoda përçaj e sundo. Principet e metodës përçaj-e-sundo janë në ndarjen e problemit në probleme më të vogla (nën-probleme) derisa këto nën-probleme janë të thjeshta për zgjidhje. Çdo ndarje e nënproblemit është thirrje përsëritse , dhe përsëritja vazhdon derisa kushti për përfundim arihet - dhe zgjidhja e nën-problemit.

Në zgjidhjen e problemit me anë të funksioneve përsëritse, gjithmonë duhet kërkuar kushtin për përfundimin e përsëritjes sepse përsëritja mund të jetë e pakufishme. Problemi pushtohet (zgjidhet) kur të gjitha thirrjet e nën-

problemeve përfundojnë me sukses). Për ilustrim të metodës përçaj e sundo dhe funksioneve përsëritëse shih problemin e kullës së Hanoit në shtojcën A.

13.3 Funksionet përsëritëse dhe jo-përsëritëse në C++

Le të konsiderojmë funksionin përsëritës në zgjidhjen e problemit të numrit faktorial

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

Hapi i parë në zgjidhje është gjetja e kushtit për përfundim.

Pasi që

$$0! = 1 \quad \text{dhe} \quad 1! = 1$$

atëherë këto janë kushte për përfundim.

Pastaj konsidero rastet më të thjeshta p.sh..

$$\begin{aligned} n = 2, & \quad 2! = 2 * 1 = 2 * 1! \\ n = 3, & \quad 3! = 3 * 2 * 1 = 3 * (2 * 1) = 3 * 2! \\ n = m, & \quad m! = m * (m-1) * \dots * 2 * 1 \end{aligned}$$

Pra definimi i $n!$ është:

$$\begin{aligned} 1 & \quad \text{për } n = 0 \text{ ose } 1 \text{ (kushti për përfundim)} \\ n! &= n * (n-1)! \quad \text{për } n = 2, 3, \dots \end{aligned}$$

Hapat kryesor për zgjidhjen e funksioneve përsëritëse janë:

Hapi 1. Konsidero rastin më të thjeshtë të problemit.

Hapi 2. Identifiko kushtin për përfundimin e thirrjeve përsëritëse. Nëse kushti për përfundim nuk dihet, funksioni përsëritës nuk mund të pëdoret. Përndryshe vazhdo në hapin 3.

Hapi 3. Konsidero rastin e nivelit më të lart të problemit dhe merr parasysh vlerën e kthyer nga nën-problemin i mëparshëm.

Hapi 4 Zgjidhja specifikon funksionin përsëritës

Konsidero funksionin përsëritës faktorial()

```
int faktorial (int n)
```

```

{
    if ( n < 0 )
    {
        cout << "Numër i pavlefshëm n = " << n ;
        exit(0);
    }

    if ( n == 1 )
        return (1);          // rasti terminues
    return ( n * faktorial( n - 1 ) );
}

```

Programi 13.1

I njëjti funksion mirëpo jo-përsëritës mund të paraqitet kështu

```

int faktorial (int n)
{
    int fak = 1;

    if ( n < 0 ) {
        cout << "Numër i pavlefshëm n = " << n ;
        exit(0);
    }

    if ( n == 1 )
        return (1);

    while( n > 1)
        fak *= n-- ;        // fak = fak * n--

    return (fak);
}

```

Programi 13.2

Në funksionin përsëritës ekzekutimi i funksionit përfundon kur $n = 0$ ose $n = 1$. Përcjellja e ekzekutimit të funksionit faktorial me $n = 5$ është:

faktorial(5)	thirrja 1
= 5 * faktorial(4)	thirrja 2
= 5 * (4 * faktorial(3))	thirrja 3
= 5 * (4 * (3 * faktorial(2)))	thirrja 4
= 5 * (4 * (3 * (2 * faktorial(1))))	thirrja 5
= 5 * (4 * (3 * (2 * 1)))	^
= 5 * (4 * (3 * 2))	kushti për përfundim
= 5 * (4 * 6)	
= 5 * 24	
= 120	

Në thirrjen e pestë paraqitet kushti për përfundim dhe vlera 1 kthehet në thirrjen e mëparshme të funksionit faktorial. Vlera e kthyer 1 përdoret për shumëzim me 2 dhe prodhimi kthehet në thirrjen e mëparshme. Vlerat e funksionit përsëritës kthehen në funksion të nivelit më të lartë përderisa nuk vjen te thirrja e parë e funksionit përsëritës.

Gjatë ekzekutimit të funksioneve përsëritëse duhet që adresa, ose vlera e kthyer, të ruhet gjatë çdo etape. Kjo bëhet me të ashtuquajturën *stak*. Stak-u është pjesë e memories së sistemit, e rezervuar posaçërisht për ruajtjen e variablave automatike, parametrave të funksionit, vlerave të kthyer nga funksioni dhe adresat e kthyer.

Në shembullin e dytë shpjegojmë funksionin përsëritës për gjetjen e numrit x në fuqinë y , ku çdo numër x në fuqinë y , (ku y nuk është numër negativ) shkruhet x^y dhe definohet si:

e padefinuar	nëse $x=0$ dhe $y = 0$
0	nëse $x=0$ dhe y jo i barabartë me 0
$\text{fuqia}(x, y) = 1$	nëse x jo baras me 0 dhe $y = 0$
x^y	nëse x jo 0 dhe y jo zero

Nga definimi, dy kushte për përfundim janë identifikuar ($x = 0$, dhe y jo baras me 0) dhe (x jo baras me 0 dhe $y = 0$). Njëra prej këtyre kushteve do ta përfundojë ekzekutimin e funksionit.

$$\begin{aligned}\text{fuqia}(x, y) &= x^y \\ &= x^1 * x^{y-1} \\ &= x * \text{fuqia}(x, (y-1))\end{aligned}$$

Më poshtë shihni renditjen e thirrjeve përsëritëse të funksionit *fuqia* me parametra $x=3$ dhe $y=4$:

<code>fuqia(3 , 4)</code>	thirrja 1
<code>= 3 * fuqia(3 , 4 - 1)</code>	thirrja 2
<code>= 3 * (3 * fuqia(3, 3-1))</code>	thirrja 3
<code>= 3 * (3 * (3 * fuqia(3, 2-1)))</code>	thirrja 4
<code>= 3 * (3 * (3 * (3 * fuqia(3 , 0))))</code>	thirrja 5
<code>= 3 * (3 * (3 * (3 * 1)))</code>	^
<code>= 3 * (3 * (3 * 3))</code>	kushti për terminim
<code>= 3 * (3 * 9)</code>	
<code>= 3 * 27</code>	
<code>= 81</code>	

Shembullin e funksionit përsëritës dhe jopërsëritës e kemi paraqitur në kodin e mëposhtëm:

```
#include <iostream.h>
```

```

#include <iomanip.h>
#include <stdlib.h>

// versioni përsëritës
int fuqia(int x, int y)
{
    if ((x == 0) && (y == 0)) {
        cout << "\n Vlera e padefinuar 0^0 \n";
        exit (0);
    }
    if (x == 0)
        return (0);
    if (y == 0)
        return (1);
    if (x > 0)
        return (x * fuqia (x, y - 1));
}

// versioni jopërsëritës
int fuqiaJoPërsëritese (int a, int b)
{
    int prodhimi = 1;

    if ((a == 0) && (b == 0)) {
        cout << "\n Vlera e padefinuar 0^0 \n";
        exit (0);
    }
    if (a == 0)
        return (0);
    if (b == 0)
        return (1);
    while (b > 0) {
        prodhimi *= a;
        b--;
    }
}

void main (void)
{
    int x = 3, y = 4;

    cout << "\n SHEMBULLI I FUNKSIONIT PËRSËRITËS "
        << "DHE JOPËRSËRITËS \n"
    cout << x << " ^ " << y << " = " << fuqia(x,y)
        << " Rezultati i funksionit përsëritës"
        << endl;
    cout << x << " ^ " << y << " = " << fuqiaJoPërsëritese(x,y)
        << " Rezultati i funksionit jopërsëritës"
        << endl;
}

```


13.4 Përsëritja dhe përcjellja e memories stak

Për ta kuptuar më mirë përsëritjen e funksionit, duhet mësuar se si vlerat ruhen dhe si lexohen nga memoria stak.

Gjatë ekzekutimit të funksioneve përsëritëse, gjuha C++ automatikisht i ruan në memorien e quajtur *stak* vlerat e ndryshme gjatë kalkulimeve, adresat e thirrjeve të funksioneve, etj. Pasi që çdo thirrje e funksionit përsëritës shkaktën ruajtjen e variablave në stak, memoria stak mund të rritet shumë nëse ka shumë thirrje të funksionit përsëritës, gjë që mund ta shkaktojë përfundimin e programit. Kur kushti për përfundim arrihet, atëherë vlerat dhe adresat e kthyer lexohen nga memoria stak, përderisa lexohet vlera e parë e ruajtur në memorie. Ruajtja dhe leximi i vlerave nga memoria stak natyrisht se merr kohë të caktuar, kështu që programi që përdor funksionet përsëritëse është më i ngadalshëm sesa ai që përdor funksione jopërsëritëse.

Si shembull se si ruhen vlerat në memorien stak vëreni funksionin përsëritës fuqia me parametra $x = 3$ dhe $y = 2$. Në funksionin `main()`, funksioni `fuqia(3,2)` është thirrja e parë. Adresa e saj le të jetë z . Para fillimit të programit memoria stak është e zbrazët:



memoria stak e zbrazët

Në thirrjen e parë të funksionit `fuqia`, memoria stak përmban vlerën 3 për variablën x dhe vlerën 2 për variablën y , adresën kthyesë të funksionit z dhe vlerën e kthyer të funksionit (momentalisht të panjohur):

thirrja e	z	adresa kthyesë
parë	2	'y'
	3	'x'
	?	vlera kthyesë

pas thirrjes së funksionit `fuqia(3,2)`

Gjendja e memories stak në program në C++ pas thirrjes së dytë `fuqia(3,1)`

thirrja e	z	adresa kthyesë
dytë	1	'y'
	3	'x'
	?	vlera kthyesë

thirrja e	Z	adresa kthyesë
parë	2	'y'
	3	'x'
	?	vlera kthyesë

pas thirrjes së funksionit fuqia(3,2)

thirrja përsëritëse e funksionit fuqia(3,0) arrin në kushtin për përfundim dhe përsëritja e funksionit ndërprehet:

thirrja e	Z	adresa kthyesë
tretë	0	'y'
	3	'x'
	?	vlera kthyesë
thirrja e	Z	adresa kthyesë
dytë	1	'y'
	3	'x'
	?	vlera kthyesë
thirrja e	Z	adresa kthyesë
parë	1	'y'
	3	'x'
	?	vlera kthyesë

Pasi plotësohet kushti për përfundim dhe më nuk ka thirrje të funksionit përsëritës, bëhet leximi i vlerave dhe i adresave kthyesë, përderisa memoria stak zbrazet tërësisht.

thirrja e	Z	adresa
dytë	1	kthyesë
	3	'y'
	3	'x'
		vlera kthyesë
thirrja e	Z	adresa
parë	2	kthyesë
	3	'y'
	?	'x'
		vlera kthyesë

thirrja e	Z
parë	2
	3
	9



Pas thirrjes të

Pas kthimit të

Memoria

funksionit stak
fuqia (3, 0)

funksionit
fuqia (3, 1)

e zbrazët pas
kthimit të
funksionit
fuqia(3, 2)

Pasi memoria stak zbrazet, vlera e fundit e kthyer në funksionin main () është 9. Vini re: nëse funksioni fuqia thirret me parametrat x dhe y me vlera më të mëdha, atëherë memoria stak duhet të ruajë më shumë vlera.

Ushtrime

1. Tentoni ta gjeni ndonjë problem ku mund ta përdorni funksionin përsëritës për zgjidhjen e problemit.
2. Pse zgjidhjet e problemeve me funksionet jopërsëritëse janë më të preferueshme?

Përmbledhje

Në këtë kapitull kemi folur për funksionet përsëritëse, të cilat janë ato funksione që thërrasin vetveten në mënyrë direkte ose indirekte. Funksionet përsëritëse përdorin memorien stak për ruajtjen e vlerave gjatë përsëritjes, prandaj përdorimi pa kujdes i funksioneve përsëritëse shkakton përfundimin e programit pa dëshirë.

Disa probleme zgjidhen më lehtë me anë të funksioneve përsëritëse për shkak të natyrës së problemit, prandaj funksionet përsëritëse janë shumë të vlefshme për zgjidhjen e problemeve të ndryshme.

Gabimet dhe trajtimi i gabimeve

Shembujt e kodit të përdorur në këtë libër i kemi shkruar më qëllim të ilustrimit të pjesëve specifike të gjuhës C ose gjuhës C++. Në të shumtën e rasteve këta shembuj artificialë të kodit nisen nga presupozimi se përdoruesi i programeve i furnizon të dhënat e pritura në program. Mirëpo në jetën e përditshme përdoruesit e programeve në të shumtën e rasteve furnizojnë të dhënat e papritura në programe. Shumica e bëjnë këtë jo me qëllim për të shkaktuar ndonjë problem në program, por prej mosdijes se cilat të dhëna janë të vlefshme (cilat të dhëna priten nga programi i ekzekutuar).

Programet e përdorura në problemet e jetës së përditshme duhen të marrin parasysh se të dhënat hyrëse të furnizuara nga përdoruesi ose nga programet e tjera (p.sh. në sistemet distributive) mund të jenë të dhëna të pavlefshme. Prandaj këto programe duhet të dizajnohen dhe të kodohen që të reagojnë në mënyrë të parashikueshme ndaj të dhënave hyrëse të pavlefshme. Reagimet e paramenduara mundën me qenë p.sh. shtypja e porosisë në monitor duke shpjeguar problemin, përfundimi i programit për arsye se programi nuk është në gjendje të vazhdojë me ekzekutim (programi nuk mund të shërohet nga problemi i shkaktuar) etj.

Secila gjuhë programuese furnizon një apo më shumë metoda për trajtimin e problemeve të shkaktuara gjatë ekzekutimit të programit. Mirëpo, qëllimi kryesor jo vetëm në trajtimin e gabimeve, mirëpo edhe në kodimin e algoritmeve ose të problemeve të tjera, është që kodi të jetë sa më i lexueshëm dhe më i lehtë për t'u modifikuar në të ardhmen. Prandaj duhet zgjedhur metodën më të përshtatshme për trajtimin e gabimeve, metodën e cila lejon zgjerimin më të lehtë të kodit dhe është më e lexueshme. Duhet të kuptohet se gabimet mund trajtohen në disa mënyra dhe se secila mënyrë ka përparësitë e saj ndaj problemeve specifike. Sistemet e suksesshme përdorin më shumë se një metodë, p.sh. metodën e përshtatshme për nivel të ulët të programimit bashkë me metodat e përshtatshme për nivelet e larta të programimit.

Qëllimi i trajtimit të gabimeve është që një pjesë e programit ta lajmërojë pjesën tjetër të programit ose përdoruesit e programit për raste të jashtëzakonshme në ekzekutimin e programit. Zakonisht programet duhet t'i lajmërojnë përdoruesit për raste të jashtëzakonshme rreze përdoruesit e programit fusin të dhëna hyrëse të pavlefshme ose në një mënyrë tjetër janë shkaktarë të gabimeve.

Nëse rastet e jashtëzakonshme, kur gabimet janë shkaktuar nga të dhënat hyrëse nga ndonjë klasë a funksion në program, atëherë këto klasë a funksione duhen të njoftohen për problemin e shkaktuar.

14.1 Trajtimi i gabimeve në gjuhën C

Në gjuhën C trajtimi i gabimeve tradicionalisht është bërë duke deklaruar variablën globale të tipit `int` si dhe `array` të tipit `char*` për përshkrimin e gabimeve. Variabla globale e tipit `int` është përdorë si indeks në `array` për ta përshkruar gabimin.

Në kodin në vijim do ta ilustrojmë metodën tradicionale për trajtimin e gabimeve në gjuhën C.

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

```
#define      NUMRI_ME_I_VOGEL          13
#define      NUMRI_ME_I_MADH           63

#define      E_NUMRI_I_VOGEL           1
#define      E_NUMRI_I_MADH            2
#define      E_JO_NUMER                 3
```

```
typedef enum { SUKSES = 0, GABIM = 1 } statusi;
```

```
/* variabla globale për ta ruajtur numrin unik të gabimit */
int numri_i_gabimit;
```

```
/* variabla globale që përshkruan gabimin duke përdorë variablën
   numri_i_gabimit si indeks */
static const char* pershkrimi[] =
{
```

```
    "",
    "Numri shumë i vogël",
    "Numri shumë i madh",
    "Të dhënat hyrëse nuk janë numër",
```

```

        };

void ShtypeGabimin()
{
    fprintf(stderr, "%s\n", pershkrimi[numri_i_gabimit]);
}

int KontrolloTedhenatHyrese(int iNumri)
{
    if (iNumri < NUMRI_ME_I_VOGEL)
    {
        numri_i_gabimit = E_NUMRI_I_VOGEL;
        return GABIM;
    }
    else if (iNumri > NUMRI_ME_I_MADH)
    {
        numri_i_gabimit = E_NUMRI_I_MADH;
        return GABIM;
    }

    return SUKSES;
}

int LexojiTedhenatHyrese(void)
{
    int    ivjet;
    char   szVjet[100], *pVjet;

    printf("Shtype numrin e vjetëve:");

    pVjet = gets(szVjet);

    // shih se a janë numër të dhënat hyrëse
    for ( ; (pVjet != NULL) && (isdigit(*pVjet)
        || *pVjet == '-') ; pVjet++)
        ;

    if (*pVjet != NULL)
    {
        numri_i_gabimit = E_JO_NUMER;
        return GABIM;
    }

    ivjet = atoi(szVjet);

    return KontrolloTedhenatHyrese(ivjet);
}

void main (void)

```

```

{
    int rezultati, pergjigjja;

    do
    {
        rezultati = LexojTedhenatHyrese();

        if (rezultati == GABIM)
            ShtypeGabimin();

        printf("A dëshironi të vazhdoni?\n");
        printf("Shtyp j për JO ose ndonjë shkronjë ");
        printf("tjetër për PO?\n");

        pergjigjja = getchar();

        /* fshije shkronjen ENTER nga memoria */
        getchar();
    }while(pergjigjja != 'j');
}

```

Programi i mësipërm i lexon të dhënat hyrëse nga përdoruesit dhe i shndërron ato në numër. Nëse numri i shtypur përmban ndonjë shkronjë, atëherë variabla globale `numri_i_gabimit` inicohet me `E_JO_NUMBER`. Nëse numri i shtypur është më i vogël se 18 atëherë variabla `numri_i_gabimit` inicohet me `E_NUMRI_I_VOGEL` si dhe nëse numri i shtypur është më i madh se 65 variabla `numri_i_gabimit` inicohet me `E_NUMRI_I_MADH`. Nëse njëri prej kushteve të përmendura më parë është i saktë, atëherë funksioni `LexojTedhenatHyrese()` kthen vlerën e barabartë me `GABIM` dhe në funksionin `main` thërrasim funksionin `ShtypeGabimin()` që shtyp fjalinë e ruajtur në variablën globale `array` pershkrimi. Funksioni `ShtypeGabimin` përdor variablën globale `numri_i_gabimit` për ta shtypur fjalinë e duhur të renditur në variablën pershkrimi. Në gjuhën C tradicionaisht numri zero do të thotë se asnjë gabim nuk është paraqitur në funksionet e ekzekutuara më parë, prandaj fjalia e parë (natyrisht me indeks zero) në variablën pershkrimi është e zbrazët. Kjo është për arsye se nëse initojmë variablën globale `numri_i_gabimit` me zero, atëherë funksioni `ShtypeGabimin` nuk do të shtypë asgjë. Pra nëse asnjë gabim nuk paraqitet në program, thirrja e funksionit `ShtypeGabimin` nuk do të shkaktojë (shtyp) asgjë.

Kjo metodë është përdorë në libraritë standarde të gjuhës C si dhe në implementimin e sistemeve operative si UNIX dhe Windows. Këto sisteme përdorin variablën globale të tipit `int` `errno` (sikurse variabla `numri_i_gabimit`) si dhe funksionet `perror` dhe `strerror` për ta shtypur pershkrimin e gabimit të fundit në funksionet e sistemit operativ (sikurse funksioni `ShtypeGabimin`).

Përparësitë e metodës së trajtimit të gabimeve të përmendura më parë janë:

- ♦ Përshkrimet e gabimeve janë të grumbulluara në një vend dhe shtimi apo ndryshimi i ndonjë përshkrimi të gabimit është më i lehtë.
- ♦ Trajtimi i gabimeve bëhet me një rregull të caktuar që e bën programin më të lexueshëm dhe më të lehtë të kuptohet nga programuesit e tjerë, të cilët nuk kanë marrë pjesë në krijimin e programit.

Natyrisht se kjo metodë ka edhe mangësitë e saj, p.sh. përdoruesi i ndonjë librerie të shkruar në gjuhën C që i trajton gabimet në metodën e përmendur më parë mund të jetë i njohur se si t'i trajtoj gabimet, mirëpo nuk mund t'i zbulojë në mënyrë të thjeshtë këto gabime. P.sh. përdoruesi i librisë do të duhej të shikonte gjithnjë nëse variabla globale errno është ndryshuar, me ç'rast do të duhej ta ekzekutonte kodin e caktuar apo të lajmëronte për gabimin e paraqitur. Kjo do të sillte deri te kodi i palexueshëm.

Edhe metoda e trajtimit të gabimeve është e papërshtatshme për ndryshimin e përshkrimeve të gabimeve për përdoruesit që nuk kanë akses në kodin e librisë standarde. P.sh. çka nëse përdoruesi i librisë standarde dëshiron ta shtypë ndonjë përshkrim tjetër për gabimin e paraqitur, apo çka nëse përdoruesi dëshiron ta shtypë përshkrimin e gabimit në gjuhë tjetër.

Gjuha C++ tejkalon mangësitë e gjuhës C në shumë pikëpamje, e po ashtu edhe në trajtimin e gabimeve. Gjuha C++ ofron metodën e trajtimit të gabimeve që është e ndërtuar në vetë gjuhën C++, e njohur si "*trajtimi i përjashtimeve*" (angl. "exception handling") në të cilën do të përqendrohemi në këtë kapitull.

Duhet kuptuar se trajtimi i përjashtimeve në gjuhën C++ nuk e zëvendëson trajtimin e gabimeve të gjuhës C, mirëpo zgjeron mundësinë për trajtimin e gabimeve.

14.2 Trajtimi i përjashtimeve

Siç përmendëm më pare, gjuha C++ e zgjeron mundësinë e trajtimit të gabimeve duke ofruar metodën *“trajtimi i përjashtimeve”*. Kjo metodë zgjedh problemin e rasteve kur funksionet që kanë shkaktuar gabimin nuk janë në gjendje të vendosin si të vazhdojnë me ekzekutimin e programit. Trajtimi i përjashtimeve mundëson që gabimet të trajtohen në nivele më të larta të programimit. P.sh. përdoruesit e librarive standarde mund të vendosin çka të bëjnë (p.sh. çfarë përshkrimi të gabimit të shtypin etj.) për gabimet e shkaktuara në funksionet e librarive standarde, në vend që këto funksione të librarive standarde të vendosin për ecurinë e ekzekutimit të programit, nëse është paraqitur ndonjë problem (përjashtim).

Gjuha C++ vë në zbatim tri shprehje të reja për trajtimin e përjashtimeve. Këto shprehje janë `try`, `catch` dhe `throw`. Në raste kur funksioni nuk është në gjendje të vazhdojë së ekzekutuari për shkak të gabimit të paraqitur, atëherë ky funksion mund të shkaktojë përjashtim me anë të shprehjes `throw`. Udhëzimi `throw` e “hedh” përjashtimin, përderisa nuk gjendet kodi i shkruar në nivel më të lartë, i cili e zë përjashtimin me anë të shprehjes `catch`. Kodi i shkruar në nivel të lartë duhet të jetë i gatshëm ta zërë përjashtimin në bllokun e kodit `try`. Pra blloku i kodit `try` përdoret për t'i rrethuar funksionet apo klasët, të cilat mund të shkaktojnë përjashtime gjatë ekzekutimit, p.sh.:

```
try
{
    FunksioniQeShkaktonPerjashtim();
}
```

Blloku `try` përcillet me bllokun `catch` i cili i “zë” përjashtimet dhe vendos ç’të bëjë me këto përjashtime, p.sh.:

```
try
{
    FunksioniQeShkaktonPerjashtim();
}
catch(char)
{
    // ekzekuto ndonjë kod specifik në bazë të
    // përjashtimit të shkaktuar.
}
```

Blloku `try` mund të përcillet me më shumë se një bllok të shprehjes `catch`, ku secili bllok i shprehjes `catch` pranon përjashtime të tipeve të ndryshme, p.sh.:

```

try
{
    Funkcioni_1( );
    Funkcioni_2( );
}
catch(char* szP)
{
    // ekzekuto ndonjë kod specifik në bazë të
    // përjashtimit të shkaktuar.
}
catch(int& iP)
{
    // ekzekuto ndonjë kod specifik në bazë të
    // përjashtimit të shkaktuar.
}

```

Për arsye se udhëzimi `throw` mund ta hedhë vetëm një përjashtim (pra vetëm një tip të përjashtimit), atëherë kodi në bllokun `catch` ekzekutohet vetëm për tipin e përjashtimit të hedhur.

Udhëzimi `throw` mund ta hedhë (ta shkaktojë) çfarëdo përjashtimi i cili mund të prezantohet me tipet e thjeshta të furnizuara nga kompajleri apo me tipet e krijuara nga vetë përdoruesi. Përparësia e përjashtimeve është se përdoruesit mund të krijojnë tipe për ta shpjeguar më detajisht problemin e shkaktuar. Në kodin që vijon kemi definuar tipin `Perjashtimi` që përmban përshkrimin e përjashtimit si dhe numrin identifikues për përjashtim.

```

#ifndef Perjashtimi_h
#define Perjashtimi_h
#include <string>

class Perjashtimi
{
public:
    Perjashtimi(const std::string& sPershkrimi,
                int iNumriIGabimit = 0)
        : m_sPershkrimi(sPershkrimi),
          m_iNumriIGabimit(iNumriIGabimit)
    {
    }

    Perjashtimi& operator=(const std::string& sPershkrimi)
    {
        m_sPershkrimi = sPershkrimi;
    }
}

```



```

        std::string Pershkrimi() { return m_sPershkrimi; }
        int NumriIGabimit() { return m_iNumriIGabimit; }
private:
        std::string m_sPershkrimi;
        int m_iNumriIGabimit;
};
#endif

```

Konstruktori i klasës Perjashtimi merr si parametër tipin string për përshkrimin e problemit (përjashtimit) si dhe tipin int për ta identifikuar përjashtimin në mënyrë të përgjithshme pa u varur nga përshkrimi i problemit. Nëse tipi Perjashtimi përdoret vetëm për ta përshkruar përjashtimin, atëherë nuk jemi të detyruar ta ofrojmë numrin identifikues të përjashtimit, pasi që konstruktori i klasës Perjashtimi përdor vlerën ngarkuese të barabartë me zero.

Vështroni kodin në vijim të shihni se si kemi shkaktuar përjashtim në funksionin Testo.

```

#include <iostream>

// fajlli ku kemi definuar klasën Perjashtimi
#include "Perjashtimi.h"

using namespace std;

void Testo()
{
    throw Perjashtimi("Kjo fjali përshkruan përjashtimin.");

    cout << "Kjo fjali nuk do të shtypet fare." ;
}

void main (void)
{
    try
    {
        Testo();
    }
    catch(Perjashtimi& p)
    {
        cout << p.Pershkrimi();
    }
}

```

Pas ekzekutimit të programit të mësipërm, do të vëreni se programi shtyp vetëm fjalinë "Kjo fjali përshkruan përjashtimin.", kurse rreshti pas shprehjes

throw nuk ekzekutohet fare. Pra, pas hedhjes së përjashtimit, kodi do të përfundojë së ekzekutuari, përderisa nuk pritet nga udhëzimi catch.

Duhet të keni parasysh se udhëzimi catch pret vetëm përjashtimet e përcaktuara në kllapa. Nëse në bllokun try funksionet apo objektet e deklaruara hedhin tipe të përjashtimeve të papritura në bllokun catch, atëherë programi do të përfundojë në mënyrë të parregullt. P.sh. në kodin që vijon kemi hedhur përjashtimin e tipit char* i cili nuk pritet në funksionin main, prandaj programi do të përfundojë në mënyrë të parregullt.

```
#include <iostream>
#include <string>
#include "Perjashtimi.h"

using namespace std;

void Testo(int iNum)
{
    if (i == 0)
    {
        throw "variabla iNum është e barabartë me zero";
    }
    else
    {
        throw Perjashtimi("variabla iNum nuk është"
                           " e barabartë me zero");
    }
}

void main (void)
{
    try
    {
        Testo(0);
    }
    catch(Perjashtimi& p)
    {
        cout << p.Pershkrimi();
    }
}
```

Për ta zgjidhur problemin e mësipërm, duhet përdorë shprehjen catch për tipin char*, p.sh.:

```
void main (void)
{
    try
    {
```

```

        Testo(0);
    }
    catch(Perjashtimi& p)
    {
        cout << p.Pershkrimi();
    }
    catch(char* psz)
    {
        cout << psz;
    }
}

```

Rekomandohet që udhëzimi `catch` të përdoret për të gjitha tipet e përjashtimeve potenciale brenda bllokut `try`.

Me siguri do të pyetni *"Si është e mundur të dini se çfarë përjashtimesh potenciale mund të hedhin funksionet e librarive standarde kur nuk kemi akses në implementimin e këtyre funksioneve?"*. Zakonisht funksionet të cilat mund të hedhin përjashtime duhen të dokumentohen duke shpjeguar listën e përjashtimeve potenciale. Prandaj lexoni dokumentacionet e librarive standarde dhe gjeni se cilat funksione hedhin përjashtime. Nëse funksionet nuk hedhin përjashtime, atëherë nuk ka nevojë të përdorni shprehjet `try` dhe `catch` për thirrjet e këtyre funksioneve.

Në rast kur lista e përjashtimeve të hedhura në program në bllokun `try` është shumë e madhe dhe nuk jeni të interesuar për përshkrimin e përjashtimeve ose dëshironi ta ekzekutoni ndonjë kod pa marrë parasysh tipin e përjashtimit, atëherë mund ta përdorni udhëzimin `catch(...)`. Udhëzimi `catch` me trepikëshin e lajmëron kompajlerin që t'i zërë të gjitha përjashtimet (të papritura më parë) pa marrë parasysh tipin e përjashtimit. Dobësia e përdorimit të shprehjes `catch` me trepikësh është se nuk do të jemi në gjendje ta shohim ndonjë informatë për ta kuptuar çfarë ka shkaktuar përjashtimin, p.sh.:

```

void main (void)
{
    try
    {
        Testo(12);
    }
    catch(...)
    {
        // këtu mund p.sh.. të shtypin ndonjë porosi
        // të përgjithshme për arsye se nuk kemi
        // informata për përjashtimin e hedhur
    }
}

```

Nëse kemi më shumë se një bllok `catch` dhe nëse ndodh ndonjë përjashtim në bllokun `try`, atëherë programi do të fillojë të ekzekutohet në bllokun `catch` të parë që pret tipin përkatës të përjashtimit të hedhur. Prandaj duhet pasur kujdes që nëse e përdorni më shumë se një bllok `catch` dhe njëri prej këtyre blloqeve është blloku `catch(...)`, atëherë blloku `catch(...)` duhet patjetër të jetë blloku i fundit. P.sh. në kodin e mëposhtëm supozojmë që funksioni `Testo` hedh përjashtim të tipit `char*`:

```
void main (void)
{
    try
    {
        Testo(0);
    }
    catch(...)
    {
        cout << "Programi ka shkaktuar përjashtim";
    }
    catch(char* psz)
    {
        cout << psz;
    }
}
```

Pasi që blloku `catch(...)` është blloku i parë (i cili i pranon të gjitha përjashtimet), kodi në bllokun:

```
catch(char* psz)
{
    cout << psz;
}
```

nuk ekzekutohet fare edhe pse tipi i përjashtimit është `char*`. Pra keni kujdes që blloku `catch(...)` të jetë blloku i fundit në listën e blloqeve `catch`, p.sh.:

```
void main (void)
{
    try
    {
        Testo(0);
    }
    catch(char* psz)
    {
        // Kodi në këtë bllok ekzekutohet së pari
        // nëse përjashtimi është i tipit char*
        cout << psz;
    }
    catch(...)
```

```
{  
    // Kodi në këtë bllok nuk ekzekutohet fare  
    // nëse përjashtimi është i tipit char*,  
    // për arsye se është pritur në bllokun  
    // e mësipërm. Përndryshe kodi në këtë bllok  
    // ekzekutohet për të gjitha përjashtimet  
    // e tipeve të tjera.  
    cout << "Programi ka shkaktuar përjashtim";  
}
```

14.2.1 Përfundimet e papritura

Më parë përmendëm se përjashtimet e papritura shkaktojnë që programi të përfundojë në mënyrë të parregullt. Në të vërtetë, përfundimin e programit e shkakton funksioni global `terminate` i furnizuar nga kompajleri, i cili e lajmëron përdoruesin për përjashtimin e papritur dhe thërret funksionin `abort` për ta përfunduar programin.

Nëse dëshironi që funksioni `terminate` ta thërrasë ndonjë funksion tjetër të furnizuar nga përdoruesi, atëherë përdoreni funksionin `set_terminate`, i cili i tregon kompajlerit cilin funksion duhet thirrur në raste kur nuk priten përjashtimet. P.sh. vështroni programin e mëposhtëm ku e kemi përdorë funksionin `perfuno` për t'u ekzekutuar në rastet kur përjashtimet e tipeve të caktuara nuk janë pritur:

```
#include <iostream>

// fajlli ku është definuar prototipi i funksioni set_terminate
#include <eh.h>

// fajlli ku kemi definuar klasën Perjashtimi
#include "Perjashtimi.h"

void Testo(int iNum)
{
    if (i == 0)
    {
        throw "variabla iNum është e barabartë me zero";
    }
    else
    {
        throw Perjashtimi("variabla iNum nuk është "
                           "e barabartë me zero");
    }
}

void perfuno()
{
    cout << "Programi ka shkaktuar përjashtim i cili nuk "
           "është pritur në njërin prej shprehjeve catch. "
           << endl;
    exit(-1);
}

void main (void)
```

```

{
    try
    {
        set_terminate(perfundo);

        Testo(0);
    }
    catch(Perjashtimi& p)
    {
        cout << p.Pershkrimi();
    }
}

```

Me anë të funksionit `set_terminate` kemi caktuar funksionin `perfundo` për t'u thirrur në raste të përjashimeve të papritura. Pasi që funksioni `Testo(0)` krijon përjashtimin e tipit `char*` i cili nuk pritet fare në funksionin `main`, programi do ta thirrë funksionin `perfundo`, i cili e shtyp fjalinë dhe e përfundon programin.

14.3 Kompletimi i klasës Vektori

Në kapitullin për shabllone kemi krijuar klasën shabllon `vektori`, e cila mund të përdoret për ruajtjen e elementeve të tipeve të ndryshme. Klasa `vektori` përmban numër të ndryshëm të elementeve gjatë ekzekutimit të programit. Secili element ka renditjen e caktuar në listën e elementeve dhe vlera e këtyre elementeve mund të lexohet ose të ndryshohet me përdorimin e indeksit (numrit rendor në listën e elementeve), p.sh.:

```

Vektori v;

v.Shto(11);
v.Shto(23);
...

v[0] = 15; // ndërro vlerën e elementit të parë

```

Çka nëse e përdorim indeksin e gabuar (për elementin joekzistues)?

Në kodin e mësipërm kemi vetëm dy elemente në vektor dhe nëse përdorim indeksin e barabartë me 3, p.sh.:

```
v[3] = 15;
```

programi i kompajluar në metodën debug do të na lajmërojë për problem pasi që e kemi përdorë funksionin assert:

```
assert(iIndeksi >= 0 && iIndeksi < m_iAnetaret);
```

Mirëpo nëse programi është kompajluar në metodën release, funksioni assert nuk paralajmëron asgjë dhe programi do të përfundojë në mënyrë të parregullt. Funksioni assert është shumë i vlefshëm gjatë kodimit dhe testimit të programit, mirëpo nuk ka kurrfarë efekti në produktin final të programit. Zakonisht gabimet në programe gjenden gjatë përdorimit të programeve në mjediset për të cilat është krijuar programi dhe nga përdoruesit e thjeshtë të cilët nuk kanë njohuri të thellë në program. Prandaj programet duhen të dizajnohen dhe të kodohen në atë mënyrë që të jenë sa më informuese për problemet e shkaktuara. P.sh. vëreni rastin e mëparshëm për indeks të elementeve në vektor. Çka nëse përdoruesi i thjeshtë i programit furnizon indeksin e elementit në vektor? Këtyre përdoruesve nuk u është i njohur implementimi i programit dhe përfundimi i programit në mënyrë të parregullt vetëm pse përdoruesi ka furnizuar ta zëmë indeksin 3 të elementit në vektor është i papranueshëm. Për t'i ikur problemit të indeksit, do të duhej që sa herë që e lexojmë apo e ndryshojmë vlerën e ndonjë elementi në vektor, të shikojmë nëse është indeksi i vlefshëm, p.sh.:

```
int    indeksi;
Vektori v;

v.Shto(11);
v.Shto(23);

...

indeksi = 0;

if (indeksi < v.NumriIAnetareve())
{
    v[indeksi] = 15;
}
else
{
    cout << "Indeksi i gabuar për elementet në listë.";
}
```

Edhe pse kjo metodë do ta zgjidhë problemin e indeksit të elementeve në vektor, është mjaft e papërshtatshme dhe jo shumë e pëlqyeshme të përdoret shprehja if-else sa herë dëshirojmë të lexojmë ose ndryshojmë elementin në vektor.

Problemi i përdorimit të shprehjeve if-else në shumë vende do të zgjidhet nëse do ta shikonim vlefshmërinë e indeksit në implementimin e operatorit {} të klasës vektori, p.sh.:

```
template <class T>
T& Vektori<T>::operator[] (int iIndeksi)
{
    // indeksi nuk mundet me qenë më i madh se numri
    // i anëtarëve (elementeve) apo më i vogël se zero
    if (iIndeksi >= NumriAnetareve())
    {
        cout << "Indeksi i gabuar për elementet në listë.";

        T t;          // gabim: nuk preferohet të kthejmë
        return t;      // vlerë të përcaktuar
    }

    elementi* p      = m_pElementet;

    // nese elementi i pare
    if (iIndeksi == 0)
    {
        return m_pElementet->tElementi;
    }
    else
    {
        // trego në elementin në renditje iIndeksi
        while (iIndeksi--)
        {
            p = p->tjetri;
        }

        return p->tElementi;
    }
}
```

Edhe pse i kemi ikur problemit të përfundimit të programit në mënyrë të parregullt në rast se përdoruesi furnizon indeks të gabuar, si dhe problemit të përdorimit të shprehjes if-else sa herë që tentojmë ta lexojmë ose ta ndryshojmë vlerën e elementit në vektor, implementimi i operatorit {} për klasën vektori, prapë nuk është në rregull.

Nëse e përdorim indeksin e gabuar, operatori {} i klasës vektori e shtyp fjalinë duke lajmëruar për problem dhe kthen vlerën e objektit t (vlerë e cila do të ishte e papërcaktueshme për të gjitha tipet).

Këtu shtrohen pyetjet: A është në rregull ta kthejmë vlerën e objektit t? Cilën vlerë duhet kthyer?

Natyrisht se kthimi i vlerës së objektit t nuk do të ishte në rregull. Objekti t nuk është inicuar, prandaj vlera e objektit t është e papërcaktueshme. Do të

ishte e pamundshme të vendosim me cilën vlerë mund ta iniconim objektin `t`, pasi që objekti `t` mund të përfaqësojë tipe të ndryshme. Edhe sikur ta gjenim ndonjë vlerë të caktuar që do të mund të përdorej për të gjitha tipet e përdorura në klasën `Vektori`, prapë problemi i operatorit `[]` nuk do të zhdukej.

Çka nëse shfrytëzuesit e klasës `Vektori` nuk janë të kënaqur me prorsinë e shtypur në përdorimin e indeksit të gabuar për elementet në vektor? Si do të mund ta shitypinim ndonjë porosi tjetër në rast të indeksit të gabuar?

Nëse përdorim implementimin ekzistues të klasës `Vektori`, atëherë porosia për indeks të gabuar do të mund të ndryshohej vetëm me ndryshimin e implementimit të klasës `Vektori`. Natyrisht se kjo nuk do të ishte aspak e dëshirueshme.

Të gjitha këto probleme do të zgjidheshin me përdorimin e përjashtimeve. Duhet përkujtuar se implemetimi i funksionit anëtar `فشije`, i klasës `Vektori` është i ngjashëm me implementimin e operatorit `[]`, prandaj edhe në implementimin e funksionit `فشije` kemi përdorë përjashtimet për t'i zgjidhur problemet e përmendura më parë. Në kodin që vijon do ta japim përkufizimin e plotë të klasës `Vektori` me ndryshimet e bëra për përdorimin e përjashtimeve. Ndryshimet janë paraqitur me shkronja të theksuara. Për të identifikuar përjashtimin, kemi përdorë klasën `Perjashtimi` të definuar në këtë kapitull.

```
#ifndef Vektori_h
#define Vektori_h

#include "Perjashtimi.h"

// numri i gabimeve eventuale ne klasen Vektori
#define G_MEMORIE_E_PAMJAFTUESHME 0
#define G_INDEKSI_I_GABUAR 1

#define GS_MEMORIE_E_PAMJAFTUESHME "Rezervimi i memories qe i \"\
"pasuksesshem për elementin në Vektor"
#define GS_INDEKSI_I_GABUAR "Indeksi i gabuar për elementin \"\
"në vektor."

template <class T>
class Vektori
{
public:
    Vektori();
    ~Vektori();

public:
```

```

        void Shto(T tAnetari);
        void Fshije(int iIndeksi);
        int NumriIANetareve();

    T& operator[] (int iIndeksi);

private:
    struct elementi
    {
        T tElementi;
        struct elementi* tjetri;
    } *m_pElementet;

    int m_iAnetaret;
};

template <class T>
Vektori<T>::Vektori()
{
    m_iAnetaret = 0;
    m_pElementet = NULL;
}

template <class T>
Vektori<T>::~~Vektori()
{
    // liro memorien e rezervuar për çdo element në objekt
    if (m_pElementet != NULL)
    {
        elementi* pTmp = NULL;
        elementi* p = m_pElementet;

        while (p != NULL)
        {
            // trego në elementin për të liruar memorien
            pTmp = p;

            // para se ta lirojmë memorien duhet të
            // tregojmë në objektin tjetër.
            p = p->tjetri;

            delete pTmp;
        }
    }
}

template <class T>
int Vektori<T>::NumriIANetareve()
{
    return m_iAnetaret;
}

```

```

template <class T>
void Vektori<T>::Shto(T tAnetari)
{
    // nëse në fillim të listës, d.m.th. elementi i parë
    if (m_pElementet == NULL)
    {
        m_pElementet = new elementi;

        if (m_pElementet == NULL)
        {
            throw Perjashtimi(GS_MEMORIE_E_PAMJAFTUESHME,
                               G_MEMORIE_E_PAMJAFTUESHME);
        }

        m_pElementet->tElementi = tAnetari;
        m_pElementet->tjetri = NULL;

        m_iAnetaret = 1;
    }
    else
    {
        // shto elementin e ri në fund të listës
        elementi* p = m_pElementet;
        while (p->tjetri != NULL)
            p = p->tjetri;

        // rezervoj memorien e duhur për element të ri
        elementi* pIri = new elementi;

        if (pIri == NULL)
        {
            // shkakto përjashtim
            throw Perjashtimi(GS_MEMORIE_E_PAMJAFTUESHME,
                               G_MEMORIE_E_PAMJAFTUESHME);
        }

        pIri->tElementi = tAnetari;
        pIri->tjetri = NULL;

        // bashkangjite elementin e ri në fund të listës
        p->tjetri = pIri;
        m_iAnetaret++;
    }
}

template <class T>
T& Vektori<T>::operator[] (int iIndeksi)
{
    // indeksi nuk mundet me qenë më i madh se numri
    // i anëtarëve (elementeve) apo më i vogël se zero

```

```

if (iIndeksi < 0 || iIndeksi >= m_iAnetaret)
{
    // shkakto përjashtim
    throw Perjashtimi(GS_INDEKSI_I_GABUAR,
                      G_INDEKSI_I_GABUAR);
}

elementi* p          = m_pElementet;

// nëse elementi i parë
if (iIndeksi == 0)
{
    return m_pElementet->tElementi;
}
else
{
    // trego në elementin në renditje iIndeksi
    while (iIndeksi--)
    {
        p = p->tjetri;
    }

    return p->tElementi;
}
}

template <class T>
void Vektori<T>::Fshije(int iIndeksi)
{
    // indeksi nuk mundet me qenë më i madh se numri
    // i anëtarëve (elementeve) apo më i vogël se zero
    if (iIndeksi < 0 || iIndeksi >= m_iAnetaret)
    {
        // shkakto përjashtim
        throw Perjashtimi(GS_INDEKSI_I_GABUAR,
                          G_INDEKSI_I_GABUAR);
    }

    elementi* p          = m_pElementet;

    // nese elementi i pare
    if (iIndeksi == 0)
    {
        m_pElementet = m_pElementet->tjetri;

        delete p;
    }
    else
    {
        // trego në një element para elementit për t'u fshirë
        while (--iIndeksi)

```

```

        {
            p = p->tjetri;
        }

        elementi* pTmp      = p->tjetri;
        p->tjetri            = pTmp->tjetri;

        delete pTmp;
    }

    m_iAnetaret--;
}

#endif

```

Për kompletimin e gabimeve eventuale në funksionin anëtar shto të klasës vektori kemi përdorë përjashtimet, në rast se rezervimi i memories për element nuk do të ishte i suksesshëm. Operatori new kthen NULL nëse rezervimi i memories nuk është i suksesshëm, përndryshe kthen tregues të tipit për të cilin tentojmë ta rezervojmë memorien.

Tani, pasi që kemi definuar klasën vektori me përdorimin e përjashtimeve, kemi një përparësi në mënyrën se si i trajtojmë gabimet. Të gjitha funksionet, të cilat mund të shkaktojnë përjashtime, duhen të përdoren në bllokun try, ku këto funksione në klasën vektori janë funksionet shto, Fshihe dhe operator [].

Si shembull për përdorimin e klasës vektori me përjashtime, vështrimi i kodin që vijon.

```

#include <iostream>
#include "Vektori.h"

using namespace std;

void main (void)
{
    Vektori<int> v;

    try
    {
        v.Shto(12);
        v.Shto(15);

        // ky rresht do të shkaktojë përjashtim
        v.Fshihe(3);
    }
    catch(Perjashtimi& p)

```

```

    {
        cout << p.Pershkrimi();
    }
}

```

Me përdorimin e përjashtimeve kodi është më i lexueshëm si dhe trajtimi i gabimeve përqendrohet në një vend. Blloku catch:

```

catch(Perjashtimi& p)
{
    cout << p.Pershkrimi();
}

```

pranon përjashtimet nëse është përdorë indeksi i gabuar, ose rezervimi i memories ka qenë i pasuksesshëm. Pra blloku catch në këtë rast është përdorë për trajtimin e dy gabimeve potenciale të ndryshme.

Zakonisht përshkrimet e përjashtimeve janë teknike dhe të pakuptueshme për përdoruesit e thjeshtë të programeve. Prandaj implementuesit e programeve duhet shpjeguar gabimet e paraqitura në programe duke marrë parasysh përdoruesit potencialë të programeve. Problemi qëndron edhe në raste ku i njëjti program duhet të shkruhet për disa gjuhë të ndryshme.

Përparësia e përjashtimeve është se nuk jemi të detyruar të përdorim përshkrimin e përjashtimit (gabimit) të furnizuar nga implementuesit e klasës që ka shkaktuar përjashtimin, pra jemi të lirë për sa i përket përshkrimit të gabimit. P.sh. nëse produkti final është për përdorues anglez, atëherë mund ta përdorim klasën vektori edhe pse i përshkruan gabimet në gjuhën shqipe. D.m.th. nuk jemi të detyruar ta shkruajmë klasën vektori (si dhe klasët e tjera) prej fillimit, vetëm për t'u përdorë në program për gjuhën angleze apo ndonjë gjuhë tjetër. Në këtë mënyrë i përmbahemi rregullave të inxhinjeringut të softuerit për përdorimin e kodit të shkruar më parë.

Në kodin e mëposhtëm kemi marrë një shembull artificial se si mund t'i përshkruajmë gabimet në gjuhë të tjera edhe pse përshkrimi i gabimeve është bërë origjinalisht në gjuhën shqipe, p.sh.:

```

#include <iostream>
#include "Vektori.h"

using namespace std;

void main (void)
{
    Vektori<int> v;

    try
    {
        v.Shto(12);
    }
}

```

```
v.Shto(15);  
v.Fshije(3);  
}  
catch(Perjashtimi& p)  
{  
    switch (p.NumriIGabimit())  
    {  
        case G_MEMORIE_E_PAMJAFTUESHME:  
            cout << "Could not allocate memory for a new "  
                  "item in the class Vektori.";  
            break;  
        case G_INDEKSI_I_GABUAR:  
            cout << "Wrong index used to delete the item -"  
                  "in the class Vektori.";  
            break;  
    }  
}  
}
```


Ushtrime

1. Krahasoni metodat e trajtimeve të gabimeve në gjuhën C dhe atë C++. Tentoni t'i identifikoni përparësitë dhe mangësitë e njëres metodë ndaj tjetrës.
2. Shkruajeni bllokun `catch` nëse funksioni `Funksioni_X()` është përdorë në bllokun `try` dhe mund të shkaktojë përjashtim të tipit `int`.
3. Shkruajini blloqet `catch` nëse funksioni `Funksioni_Y()` është përdorë në bllokun `try` dhe mund të shkaktojë përjashtime të tipit `int`, `char*` dhe `double`.
4. Shkruajeni vetëm një bllok `catch` për t'i trajtuar përjashtimet e hedhura të funksionit `Funksioni_Y()` në ushtrimin 3.

Përmbledhje

Përjashtimet mund të shkaktohen në disa vende në program, mirëpo këto përjashtime mund të zihen në vetëm një bllok `catch` të nivelit të lartë. Në këtë mënyrë, trajtimi i problemeve përqendrohet në një vend (bllok të kodit) të caktuar, që e bën programin më të lexueshëm dhe më të lehtë për t'u kuptuar. Përparësia e përjashtimeve është në atë se problemet mund të trajtohen në nivel të lartë të programimit, p.sh. përdoruesit e librarive standarde mund të vendosin mbi ecurinë e ekzekutimit të programit në raste kur funksionet ose klasët e librarive standarde shkaktojnë përjashtim. Përdoruesit e librarive standarde mund të vendosin për përshkrimet e gabimeve dhe ecurinë e ekzekutimit të programit, edhe pse nuk kanë akses në kodin e librarive standarde.

Shtojca A

Kulla e Hanoit

Problemi i lojës *Kulla e Hanoit* është problem i njohur për përsëritje, prandaj do ta paraqesim këtu për ta ilustruar edhe një herë përsëritjen e funksioneve.

Tregimi fillon kështu: tre personave në Tempullin e Brahmës i jepen tri gjilpëra dhe kërkohet prej tyre të bartin 64 disqe të arta prej një gjilpëre në tjetrën. Këta persona duhet ta bëjnë këtë duke bartur vetëm nga nga një disk dhe disku më i madh nuk mund të jetë mbi diskun më të vogël. Bota do të shkatërrohet nëse ata nuk e kryejnë këtë punë. Problemi përshkruhet më poshtë.

Problemi i Kullës së Hanoit

Le të jenë tri shtylla (gjilpërat). Kullën prej N disqesh duhet bartur prej një shtylle (shtylla fillestare) në tjetrën (shtylla përfunduese) me këto kushte:

- vetëm shtylla fillestare përmban të gjitha disqet
- vetëm nga një disk mund të bartet dhe
- asnjë disk nuk mund të vendoset mbi një tjetër më të vogël.

Në figurën A.1 shtylla 1 përmban 5 disqe, kurse shtyllat 2 dhe 3 janë të zbrazëta. Bartjet duhet të përsëriten, derisa të gjitha disqet të barten prej shtyllës fillestare (shtylla 1) në shtyllën përfunduese (shtylla 2) duke përdorë shtyllën 3 si shtyllë të përkohshme. Shih se disku më i vogël është në maje të shtyllës dhe disku më i madh është në fund. Disku i vogël është shumë e lehtë të bartet prej një shtylle në tjetrën. Mirëpo disku N (disku në fund të shtyllës) është shumë e vështirë të bartet, sepse duhet bartur $N-1$ disqet më të vogla duke iu përmbajtur kushteve të përmendura më sipër.

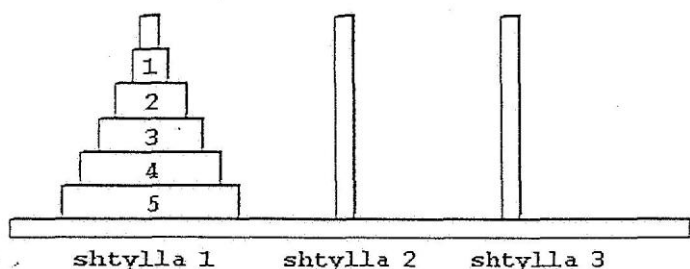


Figura A.1 Kulla e Hanoit me tri shtylla

Për zgjidhjen e këtij problemi do ta përdorim metodën *përçaj e sundo*. Këtë problem e ndajmë në tri nënprobleme të njohura si problemi me një disk, ai me dy disqe dhe ai me tri disqe. Duke e përdorë këtë metodë, do të arrijmë ta gjejmë kushtin për përfundimin e përsëritjes si dhe zgjidhjen e problemit me N-disqe.

Versioni i Kullës së Hanoit me një disk

Le të jenë tri shtylla dhe një disk në shtyllën 1. Barte diskun prej shtyllës fillestare (shtylla 1) në shtyllën përfundimtare (shtylla 2). Problemi u zgjidh.

Kushti për përfundimin e përsëritjes pra është versioni me një disk, i cili kërkon vetëm një lëvizje (bartje).

Versioni i Kullës së Hanoit me dy disqe:

Le të jenë tri shtylla dhe dy disqe në shtyllën 1. Barti të gjitha disqet prej shtyllës fillestare në shtyllën përfundimtare (shtylla 2) duke përdorë shtyllën 3 si shtyllë të përkohshme. Hapat për zgjidhjen e këtij problemi janë:

Hapi 1. Barte diskun e vogël (disku në maje) nga shtylla 1 në shtyllën 3.

Hapi 2. (Shtylla 1 përmban vetëm diskun e madh, dhe të gjitha disqet më të vogla janë bartur në shtyllën e përkohshme.) Barte diskun e madh prej shtyllës 1 në shtyllën përfundimtare (shtyllën 2).

Hapi 3. Barte diskun e vogël prej shtyllës 3 në shtyllën 2.

Hapi 4. Shtylla fillestare është e zbrazët dhe shtylla përfundimtare i përmban të gjitha disqet të renditura si duhet. Problemi u zgjidh.

Versioni i Kullës së Hanoit me tri disqe

Le të jenë tri shtylla dhe tri disqe në shtyllën 1. Barti të gjitha disqet prej shtyllës fillestare në shtyllën përfundimtare (shtylla 2) duke përdorë shtyllën 3 si shtyllë të përkohshme. Hapat për zgjidhjen e këtij problemi janë:

Hapi 1. Barte diskun 1 (diskun më të vogël) prej shtyllës 1 në shtyllën 2.

Hapi 2. Barte diskun numër 2 prej shtyllës 1 në shtyllën e përkohshme (shtyllën 3).

Hapi 3. Barte diskun numër 1 prej shtyllës 2 në shtyllën 3.

Hapi 4. Shtylla fillestare përmban diskun më të madh, dhe shtylla 3 (shtylla e përkohshme) përmban të gjitha disqet e tjera. Barte diskun më të madh (diskun 3) në shtyllën përfundimtare (shtylla 2).

Hapi 5. Barte diskun më të vogël (diskun 1) prej shtyllës 3 në shtyllën 1.

Hapi 6. Barte diskun numër 2 prej shtyllës 3 në shtyllën 2.

Hapi 7. Barte diskun më të vogël (diskun 1) prej shtyllës 1 në shtyllën 2.

Hapi 8. Shtylla fillestare (shtylla 1) është e zbrazët, kurse shtylla përfunduese (shtylla 2) i përmban të gjitha disqet, të renditura si duhet.

Problemi është zgjidhur.

Versioni i Kullës së Hanoit me tri disqe mund të zgjidhet duke u bazuar në zgjidhjen e versionit me dy disqe. Ndërsa, versioni i problemit me n disqe mund të bazohet në zgjidhjen e problemit me tri disqe.

P.sh. problemi me n disqe do të zgjidhej kështu:

- ♦ Barti $n-1$ disqe prej shtyllës 1 në shtyllën 3.
- ♦ Barte diskun më të madh (diskun n) në shtyllën 2; dhe pastaj
- ♦ Barti $n-1$ disqet në shtyllën 2.

Në këtë mënyrë, problemi me n disqe ndahet (zvogëlohet kompleksiteti në bazë të metodës *përçaj-e-sundo*) në problem të madhësisë $n-1$.

Duke analizuar zgjidhjen e versionit me një disk, dy disqe dhe tri disqe, shihet se të gjitha disqet, përveç diskut më të madh, duhet të barten në diskun e përkohshëm, para se disku më i madh të bartet në shtyllën përfundimtare.

Problemi i Kullës së Hanoit është paraqitur në kodin që vijon.

faqilli KullaEHanoit.h

```
#ifndef KullaEHanoit_h
#define KullaEHanoit_h

typedef struct ShtyllaMeDisqe
{
    int *DiskArray;
    int lartesiaUnazave;
    int numriIShtylles;
} *TRSHTYLLA;

class KullaEHanoit
{
public:
    KullaEHanoit(int iMaksDisqe);
    ~KullaEHanoit();

    void krijoShtyllat();
    void BarteDiskun (TRSHTYLLA prej, TRSHTYLLA ne);
    void zgjidhjaHanoit (int numriIDisqeve,
                        TRSHTYLLA shtyllaFillestare,
                        TRSHTYLLA shtyllaPërfunduese,
                        TRSHTYLLA shtyllaPërkohshme);

    void zgjidhjaEHanoit();
    int MaksDisqeq() { return m_iMaksDisqe; }

public:
    ShtyllaMeDisqe shtylla1; // shtylla fillestar
    ShtyllaMeDisqe shtylla2; // shtylla përfundimtare
    ShtyllaMeDisqe shtylla3; // shtylla e përkohshme
private:
    int m_iMaksDisqe;
    int m_iMaksLartesia;
    int m_iMaksDiskNo;

    TRSHTYLLA trShtylla1;
    TRSHTYLLA trShtylla2;
    TRSHTYLLA trShtylla3;
```

};

#endif

fajlli KullaEHanoit.cpp

#include <iostream>

#include "KullaEHanoit.h"

using namespace std;

KullaEHanoit::KullaEHanoit(int iMaksDisqe)

```
{
    m_iMaksDisqe      = iMaksDisqe;
    m_iMaksDiskNo     = iMaksDisqe;
    m_iMaksLartesia   = iMaksDisqe + 1;

    shtylla1.DiskArray = new int [m_iMaksLartesia];
    shtylla2.DiskArray = new int [m_iMaksLartesia];
    shtylla3.DiskArray = new int [m_iMaksLartesia];
}
```

KullaEHanoit::~KullaEHanoit()

```
{
    delete [] shtylla1.DiskArray;
    delete [] shtylla2.DiskArray;
    delete [] shtylla3.DiskArray;
}
```

void KullaEHanoit::krijoShtyllat()

```
{
    for (int i = 0; i < m_iMaksLartesia; i++)
    {
        shtylla1.DiskArray[i] = m_iMaksDisqe - i;
        shtylla2.DiskArray[i] = 0;
        shtylla3.DiskArray[i] = 0;
    }
}
```

// inicoj shtyllat

```
shtylla1.lartesiaUnazave = m_iMaksDisqe;
shtylla1.numriIShtylles  = 1;
shtylla2.lartesiaUnazave = 0;
shtylla2.numriIShtylles  = 2;
shtylla3.lartesiaUnazave = 0;
shtylla3.numriIShtylles  = 3;
```

```
trShtylla1 = &shtylla1;
trShtylla2 = &shtylla2;
trShtylla3 = &shtylla3;
```

}

```

void KullaEHanoit::BarteDiskun (TRSHTYLLA prejShtylla,
                                TRSHTYLLA neShtylla)
{
    int    diskNo =
        prejShtylla->DiskArray[prejShtylla->lartesiaUnazave];

    int    lartesia = prejShtylla->lartesiaUnazave;

    prejShtylla->DiskArray[lartesia] = 0;
    prejShtylla->lartesiaUnazave      = lartesia - 1;

    int    lartesiaERE = neShtylla->lartesiaUnazave + 1;

    neShtylla->lartesiaUnazave        = lartesiaERE;
    neShtylla->DiskArray[lartesiaERE] = diskNo;
}

void KullaEHanoit::zgjidhjalHanoit (int numriIDisqeve,
                                     TRSHTYLLA shtyllaFillestare,
                                     TRSHTYLLA shtyllaPerfunduese,
                                     TRSHTYLLA shtyllaEperkohshme)
{
    if (numriIDisqeve > 0)
    {
        zgjidhjalHanoit (numriIDisqeve - 1,
                        shtyllaFillestare,
                        shtyllaEperkohshme,
                        shtyllaPerfunduese);

        BarteDiskun (shtyllaFillestare,
                     shtyllaPerfunduese);

        cout << "\n Barte diskun " << numriIDisqeve
              << " prej shtyllës "
              << shtyllaFillestare->numriIShtyilles
              << " në shtyllën "
              << shtyllaPerfunduese->numriIShtyilles;

        zgjidhjalHanoit (numriIDisqeve - 1,
                        shtyllaEperkohshme,
                        shtyllaPerfunduese,
                        shtyllaFillestare);
    }
}

void KullaEHanoit::zgjidhjaEHanoit ()
{
    zgjidhjalHanoit(m_iMaksDisqe, trShtylla1,
                   trShtylla3, trShtylla2);
}

```


sa disqe përmban shtylla). Ky funksion thirret nga funksioni zgjidhjaHanoi().

Funksioni zgjidhjaHanoi() është funksion përsëritës, i cili përsëritet përderisa numri i disqeve është më i madh se zero. Gjatë përsëritjes, ky funksion thirr funksionin barteDiskun() dhe shtyp fjalinë për të informuar për bartjen e diskut.

Funksioni anëtar zgjidhjaEHanoi() i klasës KullaEHanoi thirret në funksionin main (). Ky funksion thërret funksionin zgjidhjaHanoi() dy herë; për t'i bartur disqet në shtyllën e tretë (shtylla e përkohshme) dhe pastaj për t'i bartur disqet prej shtyllës së tretë në shtyllën përfundimtare (shtylla 2).

P.sh. nëse e ekzekutojmë programin e mësipërm me tri disqe:

KullaEHanoi 3

rezultati do të jetë:

***** Zgjidhja e problemit të kullës së Hanoi*****

Barte diskun 1 prej shtyllës 1 në shtyllën 3
 Barte diskun 2 prej shtyllës 1 në shtyllën 2
 Barte diskun 1 prej shtyllës 3 në shtyllën 2
 Barte diskun 3 prej shtyllës 1 në shtyllën 3
 Barte diskun 1 prej shtyllës 2 në shtyllën 1
 Barte diskun 2 prej shtyllës 2 në shtyllën 3
 Barte diskun 1 prej shtyllës 1 në shtyllën 3
 Barte diskun 1 prej shtyllës 3 në shtyllën 2
 Barte diskun 2 prej shtyllës 3 në shtyllën 1
 Barte diskun 1 prej shtyllës 2 në shtyllën 1
 Barte diskun 3 prej shtyllës 3 në shtyllën 2
 Barte diskun 1 prej shtyllës 1 në shtyllën 3
 Barte diskun 2 prej shtyllës 1 në shtyllën 2
 Barte diskun 1 prej shtyllës 3 në shtyllën 2

Shtojca B

B.1 Fjalët e rezervuara në gjuhën C++

Gjuha C++ sikurse edhe gjuhët e tjera programuese, i ka fjalët që janë të rezervuara nga kompajleri dhe kanë kuptim special për kompajler. Gjatë shkrimit (kodimit) të programit duhet t'i shmangeri përdorimit të këtyre fjalëve për emërtimin e variablave, funksioneve etj.

Fjalët e rezervuara në C++ janë:

auto	for	return	volatile
bool	friend	short	wmain
break	goto	signed	while
case	if	sizeof	
catch	inline	static	
char	int	static_cast	
class	long	struct	
const	main	switch	
const_cast	__multiple_inheritance	template	
continue	__single_inheritance	this	
default	__virtual_inheritance	throw	
delete	mutable	true	
do	naked	try	
double	namespace	typedef	
dynamic_cast	new	typeid	
else	operator	typename	
enum	private	union	
explicit	protected	unsigned	
extern	public	using	
false	register	virtual	
float	reinterpret_cast	void	

Shtojca C

Alfabeti kompjuteristik ASCII :

(American Standard Code for Information Interchange -

- Kodi Standard Amerikan për Shkëmbim të Informacionit)

Kodi	Shkronja	Kodi	Shkronja	Kodi	Shkronja	Kodi	Shkronja
0	Null	32	hapësirë	64	@	96	'
1		33	!	65	A	97	A
2		34	"	66	B	98	B
3		35	#	67	C	99	C
4		36	\$	68	D	100	D
5		37	%	69	E	101	E
6		38	&	70	F	102	F
7	Zilja	39	'	71	G	103	G
8	hapësirë	40	(72	H	104	H
	mbrapa						
9	tab horiz.	41)	73	I	105	I
10		42	*	74	J	106	J
11	tab verti.	43	+	75	K	107	K
12	form feed	44	,	76	L	108	L
13	CR	45	-	77	M	109	M
14		46	.	78	N	110	N
15		47	/	79	O	111	O
16		48	0	80	P	112	P
17		49	1	81	Q	113	Q
18		50	2	82	R	114	R
19		51	3	83	S	115	S
20		52	4	84	T	116	T
21		53	5	85	U	117	U
22		54	6	86	V	118	V
23		55	7	87	W	119	W
24		56	8	88	X	120	X
25		57	9	89	Y	121	Y
26		58	:	90	Z	122	Z
27	ESCAPE	59	;	91	[123	{
28		60	<	92	\	124	
29		61	=	93]	125	}
30		62	>	94	^	126	~
31		63	?	95	_	127	DELeTe

Simbolet speciale

Simbolet speciale që përdoren në string (*array* të tipit *char*) janë shkronjat e paraqitura në tabelën që vijon. Këto simbole përfaqësohen me paraprirjen e simbolit `\` para një simboli tjetër, p.sh.. `'\b'` përfaqëson *hapësirë mbrapa*.

<code>\a</code>	zilha e kompjuterit	<code>\\</code>	simboli <code>'\'</code>
<code>\b</code>	hapësirë mbrapa	<code>\?</code>	pikëpyetja <code>'?'</code>
<code>\f</code>	fundi i faqes	<code>\'</code>	apostrofi <code>'</code>
<code>\n</code>	rresht i ri	<code>\"</code>	thonjëzat <code>"</code>
<code>\v</code>	kryerresht vertikal	<code>\t</code>	simboli për kryerresht
<code>\0</code>	null, përfundimi i stringut		

Shtojca D

Operacionet në bit

Zakonisht programet e shkruara në C dhe C++ manipulojnë me vlerat e variablaeve në bajt, p.sh. një shkronjë (char) merr 1 bajt të memories, int merr 2 ose 4 bajt (varësisht prej makinës dhe sistemit operativ) etj. Mirëpo ndonjëherë duhet të manipulojmë me vlerat e variablaeve në bit. Operacionet me bit janë paraqitur në tabelën që vijon:

Operatori	Përshkrimi
&	Operatori AND për bit
^	Operatori XOR për bit
	Operatori OR për bit
~	Komplimenti i njëshit
>>	Zhvendosja në të djathtë
<<	Zhvendosja në të majtë

Të gjithë operatorët e paraqitur në tabelë janë operatorë binarë, përpos operatorit ~ i cili është operator unar. Operatorët binarë manipulojnë me konstantat apo vlerat e variablaeve, duke marrë parasysh vlerën binare, ku çdo bit i njëres vlerë llogaritet me bitin në të njëjtën renditje të vlerës tjetër. Rezultati i operacioneve binare në bit është vlerë e përfituar nga operacionet në çdo bit.

Operatori AND

Operatori DHE (AND) jep vlerën e saktë (bitin 1₂) vetëm nëse të dy bitët e krahasuar janë të saktë (kanë vlerën 1₂). P.sh. nëse kryejmë operacionin DHE në vlerat që vijnë, do të kemi rezultatin e bazuar në çdo bit:

```

variabla 1      0 0 0 0 1 0 1 02   (e barabartë me 1010)
variabla 2      & 0 0 0 0 1 1 0 02   (e barabartë me 1210)

rezultati       0 0 0 0 1 0 0 02   (e barabartë me 810)
  
```

Pra në vlera decimale 10₁₀ & 12₁₀ japin rezultatin 8₁₀. Këtë mund ta vërtetoni edhe në program:

```
#include <iostream>

using namespace std;

void main ()
{
    int variabla1 = 10;
    int variabla2 = 12;

    int rezultati = variabla1 & variabla2;

    cout << rezultati;
}
```

Operatori OR

Operatori ose (OR) jep vlerën e saktë nëse njëri bit ose të dy bitat e vlerave janë 1, p.sh.:

variabla 1	0 0 0 0 1 0 1 0 ₂	(e barabartë me 10 ₁₀)
variabla 2	0 0 0 0 1 1 0 0 ₂	(e barabartë me 12 ₁₀)
rezultati	0 0 0 0 1 1 1 0 ₂	(e barabartë me 14 ₁₀)

Rezulta i variablave të njëjta sikurse në shembullin për operatorin DHE, në vlerë decimale është 14₁₀. Siç e vërejmë vetëm 0₂ | 0₂ jep rezultatin 0₂, kurse kombinimet e tjera japin rezultatin 1₂. Në C++ mund ta vërtetoni rezultatin kështu:

```
#include <iostream>

using namespace std;

void main ()
{
    int variabla1 = 10;
    int variabla2 = 12;
    int rezultati = variabla1 | variabla2;

    cout << rezultati;
}
```

Operatori XOR

Operatori XOR (eXclusive OR) në krahasimin e vlerave binare bit për bit jep bitin 1 vetëm nëse njëri bit prej vlerave është 1, përndryshe jep bitin 0, p.sh.:

variabla 1	0 0 0 0 1 0 1 0 ₂	(e barabartë me 10 ₁₀)
variabla 2	<u>~ 0 0 0 0 1 1 0 0₂</u>	(e barabartë me 12 ₁₀)
rezultati	0 0 0 0 0 1 1 0 ₂	(e barabartë me 6 ₁₀)

Pra $10_{10} \wedge 12_{10}$ jep rezultatin 6₁₀.

```
#include <iostream>

using namespace std;

void main ()
{
    int variabla1 = 10;
    int variabla2 = 12;
    int rezultati = variabla1 ^ variabla2;

    cout << rezultati;
}
```

Operatori Kompliment

Operatori kompliment (~) është operator unar i cili e ndërron vlerën e çdo biti të numrit binar prej 1₂ në 0₂ dhe prej 0₂ në 1₂.

variabla 1	~ <u>0 0 0 0 1 1 0 0₂</u>	(e barabartë me 12 ₁₀)
rezultati	1 1 1 1 0 0 1 1 ₂	(e barabartë me 243 ₁₀)

Pasi që operatori ~ është unar, kemi përdorë vetëm një vlerë (12₁₀) që jep rezultatin 243₁₀.

```
#include <iostream>

using namespace std;

void main ()
{
    int variabla1 = 12;
```

```
int rezultati = ~variabla1;
```

```
cout << rezultati;
```

```
}
```

Operatorët për zhvendosje

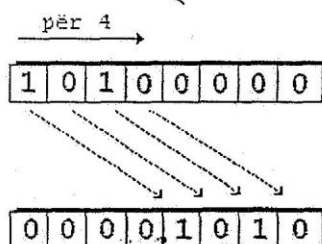
Operatorët për zhvendosje janë operatori për zhvendosje në të djathtë \gg dhe operatori për zhvendosje në të majtë \ll . Operatorët për zhvendosje janë operatorë binarë, pra operatorët që pranojnë dy vlera.

Operatori \gg zhvendos në të djathtë çdo bit të vlerës së parë për aq vende të barabarta me vlerën e dytë, p.sh.:

```
variabla 1    1 0 1 0 0 0 0 02 (e barabartë me 16010)
>> 0 0 0 0 0 0 1 02 (e barabartë me 410)
```

```
rezultati    0 0 0 0 1 0 1 02 (e barabartë me 1010)
```

Pra çdo bit i vlerës 160₁₀ zhvendoset në të djathtë për 4₁₀ vende dhe jep rezultatin 10₁₀ (shih figurën që vijon).



Në C++ zhvendosja në të djathtë bëhet kështu:

```
#include <iostream>
```

```
using namespace std;
```

```
void main ()
```

```
{
```

```
    int variabla1 = 160;
```

```
    int rezultati = variabla1 >> 4;
```

```
    cout << rezultati;
```

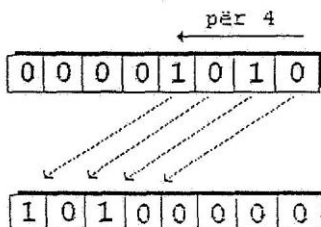
```
}
```


Kurse operatori \ll zhvendos në të majtë çdo bit të vlerës së parë për aq vende të barabarta me vlerën e dytë, p.sh.:

```
variabla 1      0 0 0 0 1 0 1 02 (e barabartë me 1010)
<< 0 0 0 0 0 1 0 02 (e barabartë me 410)

rezultati      1 0 1 0 0 0 0 02 (e barabartë me 16010)
```

Pasi zhvendosim çdo bit të vlerës 10₁₀ për 4₁₀ vende në të majtë, kemi vlerën 160₁₀. Pra zhvendosja në të majtë e rrit vlerën dhe bën të kundërtën e zhvendosjes në të djathtë, gjë e cila e zvogëlon vlerën e variablës së parë.



Në C++ zhvendosja në të majtë bëhet kështu:

```
#include <iostream>

using namespace std;

void main ()
{
    int variabla1 = 10;
    int rezultati = variabla1 << 4;

    cout << rezultati;
}
```

Shtojca E

Përparësia e operatorëve

Në tabelën që vijon kemi paraqitur operatorët në gjuhën C dhe C++. Operatorët e paraqitur në krye të tabelës kanë përparësi ndaj operatorëve në nëndarjet vijuese. Operatorët e paraqitur në të njëjtën nëndarje në tabelë i kanë përparësitë të njëjta.

Operatori	Përshkrimi	Llogaritja e përparësisë
()	thirrja e funksionit	Majtas - djathtas
[]	Elementi i tipit <i>array</i>	
->	Anëtari i treguesit	
.	Anëtari i objektit	
++	Rritje për një	Djathtas – majtas
-	Zvogëlim për një	
-	Minusi unar (p.sh. për numra negativë)	
!	Negimi logjik	
~	Komplimenti i njësht	
(tipi)	Ndërrimi i tipit me operatorin cast	
sizeof	Memoria e nevojitur për objekt	
&	Adresa e variablës	
*	Direferencimi i treguesit	Majtas – djathtas
*	Shumëzimi	
/	Pjesëtimi	
%	Modulus (mbetja)	Majtas – djathtas
+	Mbledhja	
-	Zbritja	Majtas – djathtas
<<	Zhvendosja në të majtë	
>>	Zhvendosja në të djathtë	Majtas – djathtas
<	Më i vogël	
<=	Më i vogël ose baras	
>	Më i madh	
>=	Më i madh ose baras	Majtas – djathtas
==	Baras	
!=	Jo baras	Majtas - djathtas
&&	Operatori AND për bit	
^	Operatori XOR për bit	
	Operatori OR për bit	

&&	Operatori logjik AND	Majtas - djathtas
Operatori	Përshkimi	Llogaritja e përparësis
	Operatori logjik OR	Majtas - djathtas
? dhe :	Operatori ternar	Djathtas - majtas
= += -= *= /= %= &= ^= = <<= >>=	Barazimi	Djathtas - majtas
,	Presja	Majtas - djathtas

Bibliografia

BRIAN, W. KERNIGHAN & DENNIS, M. RITCHIE [1978], The C Programming Language, Prentice Hall USA.

OUALLINE, STEVE [1992], C Elements of Style, M&T Publishing North America, co-published by Prentice Hall International (UK).

SAUMYENDRA, SENGUPTA & C. PHILLIP, KOROBKIN [1994], Object-Oriented Data Structure, Springer-Verlag.

JEAN ETTINGER [1994], Programming in C++, Macmillan Computer Science Series.

GARY, J. BRONSON [1995], A first book of C++ : from here to there, West Publishing Company.

BJARNE STROUSTRUP [1999], The C++ Programming Language, Third Edition, Addison Wesley.

ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES [1994], Design Patterns Element of Reusable Object-Oriented Software, Addison Wesley.

DAVID CHAPPELL [1996], Understanding ActiveX and Ole, A guide for developers & managers, Microsoft Press.

TENENBAUM A. and AUGENSTEIN M. [1990], Data Structures using C, Prentice-Hall.

STANDISH T. [1980], Data Structures Techniques, Addison Wesley

MARK PRIESTLEY [1996], Practical Object Oriented Design, The McGraw-Hill Companies

Horsepool, R.N. "Practical Fast Searching in Strings", Software-Practice and Experience, Vol 10.

Wirth, N, "Algorithms and Data Structures", Prentice-Hall

Steve Heller, "Optimizing C++", Prentice Hall PTR, 1999

Indeksi

A

abort, 318
 ADT, 115
 ANSI C, 8
 array, 41
 ASCII, 339
 assert, 236
 atoi, 224
 auto_ptr, 271
 auto-operatorët, 30

B

B, 8
 binarë, 12
 boolean, 65
 break, 54, 62

C

cast, 29, 30, 147, 274
 catch, 311
 char, 21
 cin, 120
 class, 128
 const_cast, 290
 continue, 64
 cout, 118
 ctype, 33

D

debug, 17
 default, 56
 define, 38
 Definimi i klasës, 130
 delete, 137
 deskonstruktorët, 133
 double, 21

do-while, 60
 dynamic_cast, 282

E

enum, 37

F

fajlli header, 33
 fclose, 211
 fgetc, 210
 fgets, 210
 fjalët e rezervuara, 338
 float, 21
 for, 26, 57, 348
 fputc, 210
 fputs, 210
 free, 96
 friend, 168
 fstream, 213
 funksionet, 69
 funksionet ngarkuese, 151
 funksionet përsëritëse, 296
 funksionet virtual, 193

G

get, 122
 getchar, 36
 getline, 217
 goto, 64

H

hierarkia, 171

I

if, 48

ifstream, 213
 include, 33
 int, 21
 ios, 226
 ostream, 118, 226
 isdigit, 33
 istream, 121
 iterator, 259

K

klasët, 128
 klasët abstrakte, 198
 klasët mbajtëse, 259
 kod objekt, 14
 kompajler, 12
 konstantat, 23
 konstruktor kopjues, 143
 konstruktorët, 133
 konstruktorët shndërrues, 138, 146
 konstruktori kopjues, 142
 kontot, 174
 kulla e Hanoi, 330

L

libraria standarde, 33
 libraritë standarde, 116
 lidhja dinamike, 189
 Linker, 14
 lista lidhëse, 97
 long, 21

M

main, 15, 18, 19, 20
 makefile, 15
 makros, 38
 malloc, 96
 metoda Debug, 17
 metoda Release, 17
 MFC, 172

N

new, 137
 null, 24

O

ofstream, 213
 operacionet aritmetike, 27
 operacionet në bit, 341
 operatorë logjikë, 52
 operatorët për zhvendosje, 344
 operatori DHE, 341
 operatori direferencues, 86
 operatori kompliment, 343
 operatori OSE, 342
 Operatori ternar, 51, 347
 operatori XOR, 343
 ostream, 118

P

parametrat, 73
 parametrat aktualë, 73
 parametrat formalë, 73
 parametrat me referencë, 74
 parametrat me vlerë, 74
 parametrat ngarkues, 155
 paraprocesori, 14, 38
 përjashtimet, 311
 përjashtimet e papritura, 318
 përparsia e operatorëve, 346
 përparsitë e operatorit, 28
 perror, 309
 përsëritësit, 259
 polimorfizmi, 189, 196
 printf, 34
 private, 158
 protected, 158
 prototipet, 73
 public, 158
 putchar, 35

R

register, 21
 reinterpret_cast, 293
 Release, 17
 renditja e elementeve, 106
 return, 45
 rreshti, 250

S

scanf, 35
 scope operator, 137
 seekg, 221
 seekp, 221
 set_terminate, 318
 setiosflags, 219
 setprecision, 119
 setw, 119, 219
 shabllonet, 231
 short, 21
 sizeof, 96
 stak, 299
 static, 80
 static_cast, 277
 std::bad_cast, 284
 stdlib, 96
 STL, 97
 streambuf, 226
 strerror, 309
 string, 7, 24
 String, 135
 stringu, 124
 strlen, 33
 struct, 37
 switch, 48, 53

T

tellg, 220
 tellp, 220
 template, 232
 terminate, 318
 this, 149
 throw, 311
 trajtimi i gabimeve, 306
 trashëgimi i shumfishtë, 183
 trashigimet - llojet, 187
 trashigimi, 171, 174
 treguesi në funksion, 102
 treguesi në tregues, 94
 treguesin në memorie, 95
 treguesit, 84
 treguesit e mençur, 267
 try, 311
 typedef, 37
 typename, 263

U

UML, 196
 unsigned, 21

V

Variable, 25
 variabla statike, 26
 variablat globale, 71
 variablat ngarkuese, 154
 vektori, 234
 vorbullla, 56

W

while, 60