LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ATMIŅAS OBJEKTU ATTĒLOŠANA PROGRAMMAS ATKĻŪDOŠANAS LAIKĀ

BAKALAURA DARBS

Autore: **Ilze Dzene-Vanaga**
Studenta apliecības Nr.: id09101
Darba vadītājs: Dr. dat. Guntis Arnicāns

RĪGA 2016

ANOTĀCIJA

Bakalaura darbā pētīts, kā programmas atkļūdošanas laikā attēlot lietotājam atmiņas objektus grafiskā veidā. Darbs turpina citu studentu iesāktos pētījumus šajā jomā. Darbā apkopotas idejas grafiska atkļūdotāja nodrošinātajai funkcionalitātei un iespējamajiem risinājumiem. Detalizētāk aplūkota C++ valodā rakstītu programmu atkļūdošana. Izveidots un aprakstīts šāda atkļūdotāja vienkāršots prototips, kurā realizēta atmiņas objektu datu iegūšana no atkļūdotāja GDB un to attēlošana, izmantojot rīku “graphviz”.

Atslēgvārdi: atkļūdošana, attēlošana, C++, datu struktūras.

ABSTRACT

Visualisation of memory objects during debugging

This paper describes, how to visualise memory objects during debugging. This continues other student's research on this subject. This work presents ideas of the necessary functions in such graphical debugger and possible implementations. Debugging of C++ language programmes is presented more detailed. Simplified prototype of such debugger is created and described. In this prototype, information about memory objects is gathered, using GDB debugger. For data display this prototype uses “graphviz” tool.

Keywords: debugging, visualisation, C++, data structures.

SATURS

Apzīmējumu saraksts....6
Ievads....7
1.Problēma un tās esošie risinājumi....10
1.1.Risināmā problēma un tās pamatojums....10
1.2.Esošie risinājumi....14
1.2.1.“DataDisplayDebugger”....14
1.2.2.Andreja Vihrova izstrādātā “memchart” bibliotēka....15
1.2.3.Citi risinājumi....15
2.Piedāvātā ideja un risinājuma varianti....16
2.1.Grafiska atkļūdotāja funkcionalitāte....16
2.1.1.Pamata funkcionalitāte....16
2.1.2.Papildus informācijas attēlošana....16
2.1.3.Programmas reversa atkļūdošana....17
2.2.Grafiska atkļūdotāja loģiskā arhitektūra....17
2.3.Datu iegūšana....18
2.3.1.Datu iegūšana no jau esoša atkļūdotāja....19
2.3.2.Atmiņas objektu “atcerēšanās” programmas izpildes laikā un to atribūtu iegūšana pēc adreses....20
2.3.3.Pieeju salīdzinājums....21
2.4.Datu analīze....22
2.4.1.Atmiņas noplūde....22
2.4.2.Kļūdainas norādes....23
2.4.3.Rīki atmiņas analīzei....23
2.5.Datu attēlošana....23
2.5.1.Vienkāršu datu attēlošana....24
2.5.2.Saliktu datu tipu attēlošana....24
2.5.3.Zināmāko datu struktūru attēlojums....25
2.5.4.Konfigurējama objektu attēlošana....26
2.5.5.Esošu grafu attēlošanas rīku lietojums....27
2.6.Datu krātuve....27
3.Izveidotā risinājuma detalizēts apraksts....29
3.1.Prototipa atbalstītā vide un arhitektūra....29
3.2.Komandrindas atkļūdotāja darbināšana valodā C++....30

3.2.1.Atkļūdotāja darbināšana Windows operētājsistēmā.....30
3.2.2.Atkļūdotāja darbināšana Linux operētājsistēmā.....33
3.3.Komandrindas atkļūdotāja darbināšana izmantojot Qt ietvaru.....35
3.3.1.Procesa izveide un sākšana.....35
3.3.2.Sloti un signāli.....36
3.3.3.Komandu nosūtīšana atkļūdotājam.....36
3.3.4.Atkļūdotāju atbilžu apstrāde.....37
3.3.5.Atmiņas objektu informācijas ieguve un saglabāšana.....39
3.4.Atmiņas objektu informācijas sagatavošana dot valodā.....42
3.5.“graphviz” darbināšana attēla ieguvei.....44
3.6.Iegūtā attēla parādīšana lietotājam.....45
Rezultāti.....46
Secinājumi.....47
Pateicības.....48
Izmantotā literatūra un avoti.....49
Pielikumi.....51
1. pielikums. Pirmkods C++ programmai Windows vidē saziņai ar komandrindas atkļūdotāju.....51
2. pielikums. Pirmkods C++ programmai Linux vidē saziņai ar komandrindas atkļūdotāju.....54
3. pielikums. Klases *GDBMIWriter* metožu atšifrējums.....56
4. pielikums. Klases *VDVariableList* pirmkods.....57

APZĪMĒJUMU SARAKSTS

Atkļūdošana – procedūra programmas sastādīšanas vai shēmas izstrādāšanas gaitā pieļauto kļūdu atrašanai, lokalizēšanai un novēršanai, ko parasti veic ar datora palīdzību [Term].

Atmiņas objekts – objekts atmiņā programmas izpildes laikā, kas satur informāciju par kādu programmas mainīgā vērtību un datu tipu.

Dinamisks mainīgais – mainīgais, kuram atmiņa tiek izdalīta programmas izpildes laikā.

Globāls mainīgais – mainīgais, kurš ir sasniedzams no jebkuras vietas programmā [Term].

Integrētā izstrādes vide – integrētu rīku kopums programmatūras izstrādei. Rīki parasti tiek darbināti no vienas lietotāja saskarnes un sastāv no kompilatora, redaktora, atkļūdotāja un citiem rīkiem [Term].

Kompilēšana – augsta līmeņa valodā uzrakstītas programmas translēšana mašīnvalodā [Term].

Lokālais mainīgais – mainīgais datora programmā, ko definē un izmanto tikai vienā programmas modulī [Term].

Mainīgais – rakstzīme vai rakstzīmju secība, ko izmanto, lai apzīmētu kādu saglabājamu lielumu, kam ir savs vārds un vērtība un ko var mainīt programmas izpildes gaitā [Term].

Mainīgā objekts – objektorientēts GDB MI saskarnes elements, kas reprezentē kādu izteiksmi (piemēram, mainīgo) programmas atkļūdošanas laikā [GDBMi].

Norāde – objekts, kura vērtība ir kāda cita objekta adrese.

Parsētājs – programma, kas lielus datu blokus sadala mazākās, vieglāk interpretējamās daļās.

Pārtraukumpunkts – vieta programmā, kur tās izpilde var tikt pārtraukta, lai izdrukātu mainīgo vērtības vai veiktu programmas testēšanu [Term].

Statisks mainīgais – mainīgais, kuram atmiņa izdalīta programmas kompilēšanas laikā.

Turis – lokāls identifikators, ar kura palīdzību var realizēt piekļuvi kādai ierīcei vai tādiem objektiem kā dators, logi vai dialoglodziņi [Term].

IEVADS

Atkļūdošana ir nozīmīga programmu izstrādes procesa sastāvdaļa, ar ko saskaras gan pieredzējuši programmētāji, gan studenti, kas sākuši apgūt programmēšanu. Bieži lielākā daļa programmatūras izstrādes laika tiek veltīta testēšanai un atkļūdošanai. Valodā C++ visbiežākās kļūdas saistītas ar atmiņas nekorektu izmantošanu [Bod, 2015]. Nekorekta atmiņas izmantošana ir viena no visgrūtāk atrodamajām kļūdām, jo dati atmiņā var tikt "sabojāti" pavisam citā programmas vietā, kā varētu šķist, pārskatot programmas kodu.

Lielākā daļa programmas mainīgo "dzīvo" dinamiskajā atmiņā. Stekā atrodas tikai neliela daļa no tiem. Steka atmiņa tiek izdalīta secīgi, bet dinamiskajā atmiņā objekti bieži ir "izmētāti" pa atmiņu. Ja programmas izpildes laikā pamanīta kļūda, programmētājam ne vienmēr ir vienkārši saprast, kuros atmiņas objektos ir kļūdainie dati un vēl svarīgāk – kurā brīdī datos ieviesusies kļūda.

Minēto problēmu ilustrē šāds piemērs. 1. attēlā parādīts vienkāršas C++ programmas "Simple" kods.

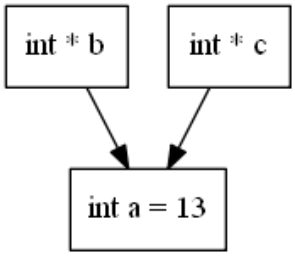
```
#include <iostream>
using namespace std;
int main()
{
    int a = 7;
    int * b = &a;
    int * c = b;
    *c = 13;
    cout << "The value of a is " << a << endl;
    return 0;
}
```

1. att. **Programmas "Simple" kods C++ valodā**

Pavirši pārskatot kodu, var šķist, ka programmas izdrukātajam teikumam vajadzētu būt "The value of a is 7", bet, darbinot programmu, tiek attēlots "The value of a is 13". Ja programmētājs, rakstot programmu, bija iecerējis, ka izdrukā vajadzētu parādīties skaitlim 7, programmā ir kļūda. Programmētājam nākas atrast kļūdas cēloni. Biežāk lietotie rīki – mainīgo vērtību izdruka tieši no programmas vai atkļūdotāja lietošana.

Abos gadījumos programmētājs bez īpašas piepūles var iegūt lokālo mainīgo sarakstu un to vērtības. Lai iegūtu informāciju par dinamiski veidotajiem mainīgajiem, programmētājam nākas papildus veidot metodes, kas nodrošina šo datu izdruku, vai, lietojot atkļūdotāju – prasīt atkļūdotājam informāciju par šiem objektiem vai nu tieši pēc adreses vai izteiksmēm, kuras ietver lokālos mainīgos.

Nevienā no šiem veidiem nav redzams visu objektu kopskats, tāpēc programmētājs parasti ņem talkā zīmuli un papīru, iegūstot zīmējumu, kas pēc satura ir līdzīgs 2. attēlā parādītajam. Turklāt, ja darbošanās ar norādēm programmas izpildes laikā ir apjomīgāka un sarežģītāka, programmētājam nākas veidot vairākus šādus zīmējumus, lai atrastu kļūdas cēloni. Rodas jautājums: vai nebūtu iespējams mainīgo vērtību saraksta vietā no kāda rīka iegūt jau šādu attēlu?



2. att. **Programmas "Simple" atmiņas objektu stāvoklis pirms izdrukas**

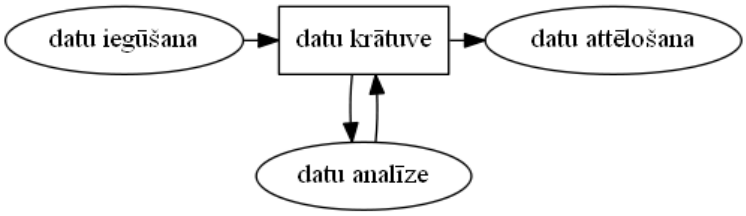
Bakalaura darbā tiek piedāvātas idejas, kā vizuāli attēlot atmiņas objektus atkļūdošanas laikā, padarot atkļūdošanu ērtāku. Darbā sīkāk pētīta C++ programmu atkļūdošana, bet sniegtās risinājuma idejas var tikt pielietotas arī citās valodās rakstītu programmu atkļūdošanai, tās atbilstoši pielāgojot valodas un ar valodu saistīto rīku specifikai.

Šo problēmu pētījuši citi Latvijas Universitātes studenti. Andrejs Vihrovs kursa darbā "Datu struktūru grafiskā attēlošana C++ programmām" izveidojis bibliotēku "memchart", kuru lietojot un veicot nelielas izmaņas atkļūdojamās programmas pirmkodā, iespējams izvēlētajos punktos iegūt dinamiskajā atmiņā esošo programmas atmiņas objektu attēlojumu [Vih, 2010]. Andreja Vihrova iesākto turpinājis Agnis Āriņš savā bakalaura darbā "Dinamiska programmu atmiņas objektu analīze", papildinot pētījumu ar datu analīzes iespējām, piemēram, iespējamas atmiņas noplūdes kļūdas atrašanai [Āri, 2011]. Ainārs Augulis bakalaura darbā "Java programmas atmiņas objektu vizualizācija" pētījis atmiņas objektu attēlošanas iespējas Java valodā [Aug, 2012]. Daļa šajā bakalaura darbā piedāvāto ideju grafiska atkļūdotāja izveidei aizgūtas no minētajiem darbiem, kā arī darba vadītāja Gunta Arnicāna ieteikumiem.

Bakalaura darbā atmiņas objektu grafiska attēlošana aplūkota kā trīs secīgu darbību kopums. Šīs darbības ir: datu iegūšana par atmiņas objektiem, iegūto datu analīze un datu attēlošana (skatīt 3. attēlu). Papildus šīm darbībām nepieciešams nodrošināt arī iegūto datu saglabāšanu.

Šāds dalījums izvēlēts, jo katru no minētajām darbībām var realizēt kā atsevišķu komponenti, kuras realizācija nav atkarīga no pārējo komponenšu realizācijas. Pietiktu definēt

datu krātuves saskarnes ar katru no šīm komponentēm.



3. att. **Atmiņas objektu grafiskas attēlošanas sastāvdaļas**

Datu iegūšana ietver informācijas informācijas iegūšanu par visiem tim atmiņas objektiem, kuriem programmai ir pieeja kādā noteiktā laika momentā. Dati, kurus nepieciešams iegūt par katru no objektiem, ir tā datu tips, vārds (ja programmētājs to ir devis), vērtība un adrese.

Datu analīze ietver saistītu datu objektu struktūras analīzi un iespējamo kļūdu meklēšanu. Struktūras analīzes laikā tiek noskaidrots datu struktūras veids (koks, saraksts, grafs u.tml.). Šāda informācija varētu būt noderīga datu attēlošanas posmā vai arī tā varētu tikt izmantota, ja programmētājs jau definējis, kāda struktūra ir iecerēta – ja rezultāts atšķirtos no iecerētā, varētu veikt vēl dziļāku kļūdu cēloņa analīzi. Piemēram, ja lietotājs ir iecerējis sarakstu, bet tā vidū ir cikls, datu analīzes laikā varētu tikt atrasta kļūdainā norāde un saglabāta informācija par to datu krātuvē. Datu attēlotājs to izmantotu, lai īpaši iezīmētu kļūdaino norādi.

Datu attēlošanas laikā lietotājam tiek parādīts grafisks atmiņas objektu attēlojums vai nu attēla vai interaktīvu grafisku elementu veidā. Objekti tiek parādīti kā ģeometriskas figūras, kurās ierakstīti objektu dati. Norādes starp objektiem tiek attēlotas kā bultiņas (piemēram, 2. attēlā ir attēloti programmas “Simple” objekti).

Datu krātuvei būtu jāsaglabā iegūtie dati par atmiņas objektiem, kā arī cita informācija, kas būtu interesanta datu attēlošanas laikā. Papildus iespējas rastos, ja datu krātuve saglabātu arī vēsturiskus datus. Tas dotu iespēju pēc tam programmētājam šos datus pārskatīt uz priekšu un atpakaļ, lai vieglāk ieraudzītu kļūdas cēloni.

Bakalaura darbā apkopotas idejas šo komponenšu realizācijai, kā arī ir izstrādāts prototips – Linux vidē darbināma QT ietvara lietotne. Tajā realizētas atsevišķas minētās idejas.

Bakalaura darbu veido trīs nodaļas daļas. Bakalaura darba pirmajā nodaļā sniegts sīkāks ieskats problēmā un tās esošajos risinājumos. Bakalaura darba otrajā daļā apkopotas idejas grafiska atkļūdotāja izveides variantiem. Darba trešajā daļā detalizētāk aprakstīts atsevišķu ideju risinājums, kā arī aprakstīts izstrādātais grafiska atkļūdotāja prototips.

1. PROBLĒMA UN TĀS ESOŠIE RISINĀJUMI

Doma par atmiņas programmu grafisku attēlošanu nav jauna [Myers, 1980]. Idejas par grafisku atkļūdotāju atrodamas vairākos avotos, bet šobrīd industrijā nav daudz rīku, kas kaut vai daļēji ietvertu šādu funkcionalitāti.

1.1. Risināmā problēma un tās pamatojums

Lielu daļu sava laika programmētāji pavada, testējot un atkļūdojot programmas. Ne vienmēr uzrakstītā programma darbojas tā, kā programmētājs to ir iecerējis. Programmas atkļūdošana sevī ietver kļūdas cēloņa atrašanu un novēršanu. Kļūdas cēloni var atrast, atrodot vietu, kurā programma strādā citādi, nekā bija iecerēts.

Visbiežāk programmētājs, projektējot programmā lietotās datu struktūras, izmanto to grafisku attēlošanu – objektu attēlojumam lieto taisnstūrus vai apļus, bet norāžu attēlojumam lieto bultiņas. Atkļūdošana būtu ērta tad, ja informācija par programmas objektiem tiktu iegūta šādā – programmētājam pazīstamā veidā. Programmētājam atliktu salīdzināt savu ieceri ar faktisko programmas norisi.

Valodā C++ visbiežākās kļūdas saistītas ar atmiņas nekorektu izmantošanu. Piemēram, uz vienu un to pašu objektu norāda divi norādes tipa mainīgie, bet programmētājs bija iecerējis, ka mainīgie rāda katrs uz savu objektu. Ja programmā tiek mainīta objekta vērtība caur kļūdaino norādi, programmētājs bez rūpīgas izpētes kļūdu var arī neatrast.

Ievadā tika ieskicēta programma “Simple” un tās atkļūdošanas iespējas. Vēlreiz aplūkosim programmu (skatīt 1.1 att.) un detalizētāk analizēsim programmētāja iespējamās atkļūdošanas darbības.

```
#include <iostream>
using namespace std;
int main()
{
    int a = 7;
    int * b = &a;
    int * c = b;
    *c = 13;
    cout << "The value of a is " << a << endl;
    return 0;
}
```

1.1. att. **Programmas “Simple” kods C++ valodā**

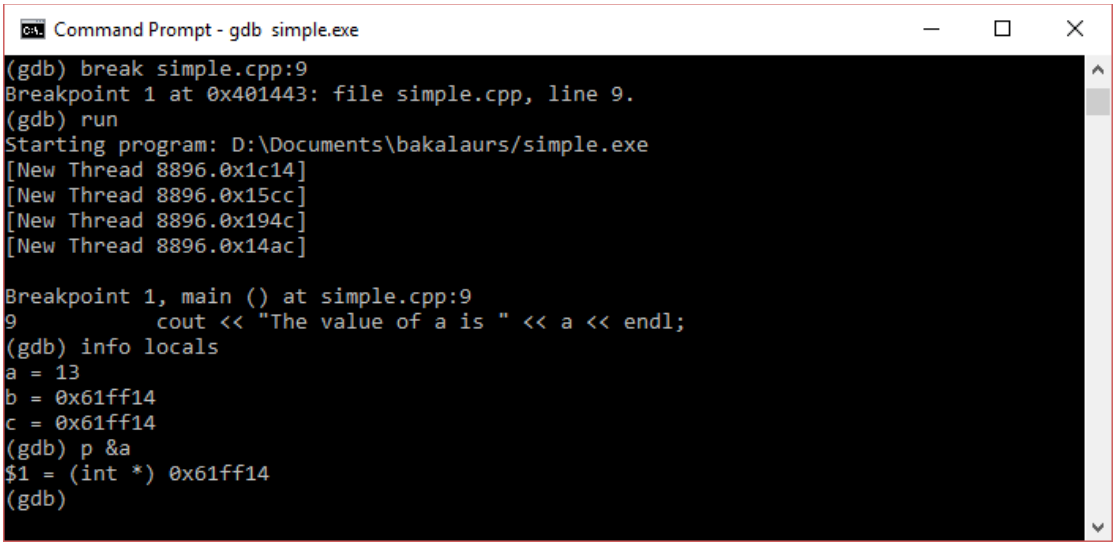
Parasti kļūda tiek meklēta vienā no diviem veidiem: analizējot programmas tekstu vai iegūstot programmā izmantoto atmiņas objektu vērtības dažādos laika brīžos.

Programmas teksta analīzes scenārijā darbību secība varētu būt apmēram šāda. Vispirms programmētājs pārskata kodu, pārliecinoties, ka mainīgajam *a* tiešām nekur netiek izmainīta vērtība no sākotnēji piešķirtās vērtības 7. Pēc tam ierauga, ka mainīgajam *b* kā vērtība tiek piešķirta manīgā *a* adrese, bet arī mainīgajam *b* nekur netiek mainīta vērtība. Tad ierauga, ka mainīgajam *c* kā vērtība tiek piešķirta mainīgā *b* vērtība un ka adresē, uz kuru norāda mainīgais *c* tiek ierakstīta vērtība 13.

Cits scenārijs būtu noteiktos programmas brīžos vienkārši izdrukāt interesējošā mainīgā vērtību (var izmantot to pašu izdruku, ko jau lieto programmas noslēgumā, tikai atkārtojot to kaut vai pēc katra soļa). Šādi programmētājs atrod, ka koda rinda, kas "sabojā" *a* vērtību ir "*c = 13". Šeit vienalga nākas atgriezties pie koda analīzes, lai saprastu, kādā veidā mainīgais *c* ir saistīts ar mainīgo *a*.

Pieredzējis programmētājs izdruku vietā lietotu atkļūdotāju. 1.2. attēlā parādīts fragments no komandrindas GDB atkļūdotāja sesijas. Secīgi izpildītas šādas darbības:

1) pirms izdrukas uzstādīts pārtraukumpunkts ar komandu "break simple.cpp:9";
2) darbināta programma līdz pārtraukumpunktam ar komandu "run";
3) iegūtas lokālo mainīgo vērtības ar komandu "info locals";
4) iegūta mainīgā *a* adrese ar komandu "p &a".



1.2. att. **GDB atkļūdotāja sesijas fragments**

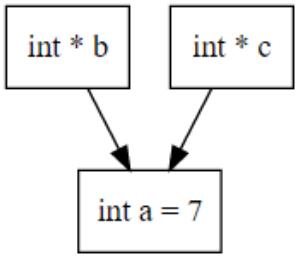
Kā redzams, mainīgo *b* un *c* vērtības ir adreses, kas nozīmē, ka programmētājam nākas lietot papildus vaicājumus, lai uzzinātu kopainu. Jau 3 mainīgo gadījumā tas nav vienkārši. Sarežģītas programmas gadījumā ar daudz mainīgajiem tas būtu vēl laikietilpīgāk. Turklāt saraksts ar mainīgo vērtībām un adresēm nerada izpratni par saitēm starp atmiņas objektiem.

Alternatīva būtu programmētājam lieto kādu integrētās izstrādes vidi ar jau iekļautu

atkļūdotāju. Parasti šie iekļautie atkļūdotāji dod iespēju darbināt programmu soli pa solim, vai apstāties pārtraukumpunktos un apskatīt mainīgo vērtības ērtā veidā, nemācoties komandrindas atkļūdotāja komandas. Parasti šādos atkļūdotājos mainīgo vērtības tiek attēlotas tabulas veidā vai labākajā gadījumā – koka veidā (ļaujot norādes tipa mainīgos "izvērst", iegūstot datus par objektu, uz kuru šis mainīgais norāda).

Lai arī koka veida struktūras lietošana padara atkļūdošanu mazliet ērtāku, tā nav derīga visos gadījumos. Piemēram, situācijā, kad kāds klases objekts atrodas dinamiskajā atmiņā, bet uz to norāda lokāls mainīgais, ir ērti, ka 2 soļos var piekļūt klases objektam (attēlo lokālos mainīgos, izvērš interesējošo). Tomēr aplūkotajā programmā "Simple" koka veida attēlošana arī nedod lielu pienesumu. Izvēršot mainīgo *c* uzzinām, ka objekta, uz kuru tas rāda, vērtība ir 13. Tas nepalīdz saprast, ka objekts, uz kuru *c* norāda ir *a*.

Ja kodā tiek lietotas norādes, neatkarīgi no izmantotā scenārija vērtību un adrešu iegūšanai, bieži programmētājam nākas lietot papīru un rakstāmo, lai saprastu, kurš mainīgais uz kuru norāda un kuri mainīgie varētu ietekmēt kāda objekta vērtību. Pēc koda pārskatīšanas soli pa solim līdz izdrukas vietai aplūkotajā programmā "Simple", programmētājs varētu nonāk pie līdzīga zīmējuma, kāds parādīts 1.3. attēlā.



1.3. att. **Programmas "simple.cpp" atmiņas objektu stāvoklis pirms izdrukas**

Pēc šādas ainas ieraudzīšanas jau uzreiz kļūst saprotams, ka mainīgā *a* vērtību var "sabojāt" visi trīs lokālie mainīgie. Bet cik daudz laika nepieciešams, lai šādu attēlu uzzīmētu, ja ir vairāk mainīgo un starp to radīšanas un izdrukas brīdi notiek vairāk darbības? Vai nebūtu iespējams mainīgo vērtību saraksta vietā no atkļūdotāja iegūt šādu attēlu?

Problēmu demonstrē arī piemēra programma "SimpleList", kas parādīta 1.4. attēlā. Kā redzams, programma izveido saistīto sarakstu, izmantojot struktūru *Node*, kuras lauki ir skaitliska vērtība *value* un norāde uz nākošo struktūras *Node* elementu laukā *next*. Aplūkotā programma izveido sarakstu ar 12 elementiem un divus norādes mainīgos – *n*, kurš rāda uz saraksta pirmo elementu, un *m*, kurš rāda uz saraksta pēdējo elementu. Pieņemsim, ka kādā programmas izpildes brīdī gribam apskatīt saraksta 10. elementa vērtību. Lietojot komandrindas atkļūdotāju, jāiegūst informācija par objektu, kurš atrodams caur izteiksmi "n-

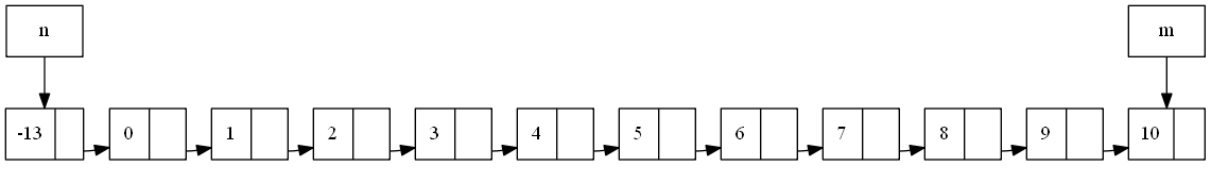
>next->next->..->next->value", vajadzīgo skaitu reižu (9 reizes) lietojot norādi *next*. Līdzīgi varam darboties ar izstrādes vides iebūvēto atkļūdotāju – vai nu paprasot tam vērtību pēc šīs izteiksmes, vai koka struktūras gadījumā – vajadzīgo skaitu reižu izvēršot objektu, uz kuru norāda kārtējais *next*.

```
#include <iostream>
using namespace std;
struct Node
{
    int value;
    Node * next;
};
int main()
{
    Node * n = new Node();
    n->value = -13;
    n->next = new Node();
    Node * m = n;
    for (int i = 0; i < 11; i++)
    {
        m->next = new Node();
        m = m->next;
        m->value = i;
    }
    m->next = NULL;
    return 0;
}
```

1.4. att. **Programmas "SimpleList" kods**

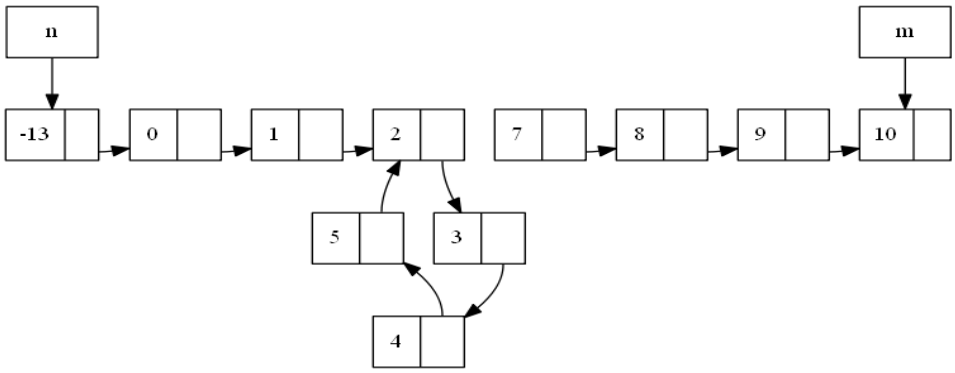
Lai aplūkotu visus saraksta elementus, vienkāršākais veids būtu izveidot funkciju, kas izdrukā visas saraksta vērtības (izdruku pārtraucot, kad *next* vērtība ir nulles norāde). Šādu funkciju ir viegli izveidot, turklāt datus izdrukāt var samērā pārskatāmi (piemēram, ja interesē elementa indekss sarakstā, šādu informāciju var vienkārši pievienot izdrukai).

Vēl patīkamāk būtu ieraudzīt sarakstu bez papildus izdruku veidošanas, piemēram, 1.5. attēlā redzamajā veidā.



1.5. att. **Programmas "SimpleList" atmiņas objektu attēlojums soli pirms programmas izpildes beigām**

Saistītu sarakstu gadījumā viena no tipiskākajām kļūdām mēdz būt cikla radīšana sarakstā. Aplūkosim gadījumu, kad, papildinot programmas "SimpleList" kodu, programmētājs izveidojis tādu programmu, kuras atmiņas objekti kādā brīdī izskatās tā, kā parādīts 1.6. attēlā.



1.6. att. **Atmiņas objektu stāvoklis kļūdainā "SimpleList" programmā**

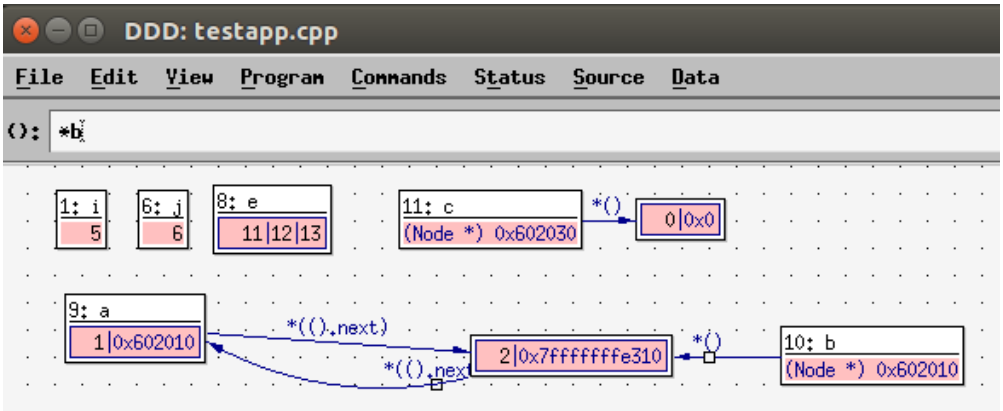
Kā redzams, saistītajā sarakstā ir radies cikls. Ja izveidota metode saistītā saraksta izdrukai, kas beidz drukāšanu tad, kad objekta lauks *next* ir nulles norāde, šī metode ieciklosies un programmētājam būs jāmēģina noskaidrot atmiņas objektu stāvokli citā veidā. Arī atkļūdotāja ar objektu kokveida attēlošanu izmantošana novedīs līdz līdzīgiem rezultātiem – būs iespēja bezgalīgi apskatīt nākamo objektu.

Šis piemērs parāda, ka vizuāla atkļūdotāja nepieciešamību rada vairākas problēmas un tā izveidošana tādā formā, kā tas aprakstīts turpmākajās nodaļās, atkļūdošanu padarītu ievērojami ērtāku.

1.2. Esošie risinājumi

1.2.1. "DataDisplayDebugger"

"DataDisplayDebugger" ir lietotne, kas nodrošina grafisku saskarni tādiem komandrindas atkļūdotājiem kā GDB, DBX, WDB, Ladebug, JDB, XDB, un citiem. Šīs rīks iekļauj visas ierastās atkļūdotāja funkcijas. Tas kļuvis pazīstams, jo piedāvā attēlot atmiņas objektus grafu veidā (skatīt 1.7. attēlu) [DDD].



1.7. att. **Lietotnes "DataDisplayDebugger" ekrānuzņēmums**

1.7. attēlā parādīts vienkāršas programmas atmiņas objektu grafs "DataDisplayDebugger" piedāvātajā attēlojumā. Programma satur divus *int* tipa mainīgos *i* un *j*, *int* masīvu *e*, lietotāja definētas klases *Node* instanci *a* un divas norādes uz klases *Node* objektiem – mainīgos *b* un *c*. Lai iegūtu attēlā parādīto ainu, jāprasa parādīt katru no mainīgajiem. Lai arī lietotne satur funkciju, kas piedāvā parādīt visus lokālos mainīgos, to vērtības tiek attēlotas bez mainīgo nosaukumiem un nav ērti pārskatāmās.

"DataDisplayDebugger" piedāvā objektus skatam pievienot vai nu pēc to izteiksmes (mainīgo vārda vai norāžu gadījumā – ceļa līdz objektam), vai atrisinot jau attēlotajos objektos ietvertās norādes – iezīmējot tās un izvēloties funkciju *Display*. Izveidotos skatus iespējams saglabāt, bet jauni objekti automātiski netiek attēlam pievienoti – lietotājam pašam jāseko līdzi, vai nav iespējams palūgt atrisināt kādu jaunu norādi.

Pēc autores domām, "DataDisplayDebugger" tomēr pilnībā neatbilst tam, kāds tiek meklēts šī bakalaura darba ietvaros. Šis rīks automātiski nepiedāvā attēlot visus objektus atmiņā. Rīka lietojamība, pēc autores domām, nav ērta un intuitīva. Jaunākā lietotnes versija, kas šobrīd pieejama ir versija 3.3.12, kas iznākusi 2009. gada februārī. Lietotne darbojas tikai "Unix" tipa vidēs.

1.2.2. Andreja Vihrova izstrādātā "memchart" bibliotēka

"Memchart" bibliotēkas detalizēta funkcionalitāte aprakstīta Andreja Vihrova kursa darbā [Vih, 2010]. Bibliotēka piedāvā atkļūdot C++ programmas, atmiņas objektus iegūstot tieši no programmas, tās izpildes laikā. Atkļūdojamajā programmā iekļaujama minētā bibliotēka, un veicami nelieli pielāgojumi atkļūdojamās programmas pirmkodā. Norādītajā pārtraukumpunktā (tas norādāms, izejas kodā iekļaujot metodes "memchart::update()" izsaukumu) lietotājam tiek attēloti atmiņas objekti, kas atrodas dinamiskajā atmiņā.

Pēc autores domām, šis ir viens no labākajiem šobrīd pieejamajiem risinājumiem, taču tam ir divi trūkumi:

1) programmētājam nākas pielāgot atkļūdojamās programmas kodu;
2) nav iespējams iegūt datus par tādiem atmiņas objektiem, kuru inicializēšanā netiek lietots operators *new*.

1.2.3. Citi risinājumi

Ir atrodami arī citi rīki, kas noteiktām valodām vai noteiktām vidēm piedāvā nelielu daļu no tās funkcionalitātes, kas būtu sagaidāma no grafiska atkļūdotāja. Salīdzinot ar rīkiem, kas minēti citu studentu darbos (kuri tapuši pirms 4 līdz 6 gadiem), diemžēl jāatzīst, ka nav izdevies atrast jaunus rīkus, kas piedāvātu vēlamo funkcionalitāti.

2. PIEDĀVĀTĀ IDEJA UN RISINĀJUMA VARIANTI

Darba pirmajā nodaļā sniegts ieskats problēmās, ko nav iespējams vienkārši un ērti risināt ar šobrīd pieejamajiem rīkiem. Šajā nodaļā aprakstīta rīka vai rīku kopuma (turpmāk tekstā sauks – grafisks atkļūdotājs) ideja – tā funkcionalitāte, arhitektūra loģiskā līmenī un daži realizācijas varianti.

2.1. Grafiska atkļūdotāja funkcionalitāte

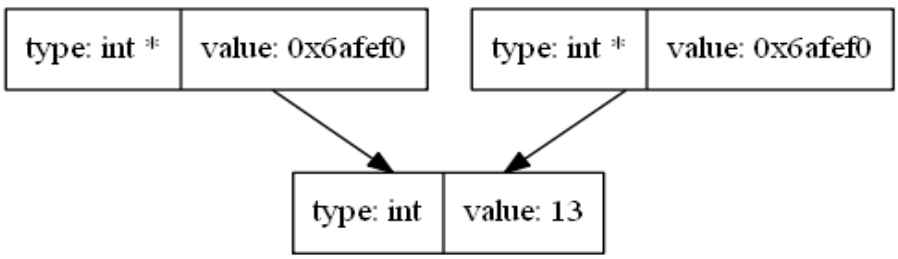
Lai sniegtu idejas grafiska atkļūdotāja arhitektūrai, vispirms jānoskaidro, kāda funkcionalitāte varētu tikt sagaidīta no šāda rīka un kas nepieciešams, lai šādu funkcionalitāti varētu nodrošināt.

2.1.1. Pamata funkcionalitāte

Grafiskam atkļūdotājam jānodrošina atkļūdojamās programmas redzamo objektu grafiska attēlošana noteiktā programmas izpildes brīdī. Redzamie objekti ir tie atmiņas objekti, kas sasniedzami, virzoties pa norādēm un klašu elementiem, sākot no kāda lokālā mainīgā.

Tas nozīmē, ka grafiskam atkļūdotājam jāspēj piekļūt programmas objektiem un saņemt par tiem nepieciešamo informāciju. Tam būtu jāspēj noteikt attēlojamo objektu kopu (iekļaujot visus attēlojamos, neiekļaujot liekus objektus), jāspēj uzzināt par tiem minimāli nepieciešamo informāciju – objekta datu tipu, vērtību un adresi.

2.1. attēlā parādīts piemērs, kā varētu izskatīties 1. nodaļā aplūkotās programmas "Simple" objektu minimāli attēlojamā aina. Datu tipu un vērtību attēlošanas varianti detalizēti aplūkoti vēlāk.



2.1. att. **Programmas "Simple" objektu pamata atribūtu attēlojums soli pirms programmas darba beigām**

2.1.2. Papildus informācijas attēlošana

Papildus pamata informācijai par objektu, varētu tikt attēlota arī cita informācija,

piemēram, vārds, kādu programmētājs tam devis programmas pirmkodā. Tāpat varētu attēlot informāciju par iepriekšējām objekta vērtībām programmas izpildes laikā (piemēram, ar peli uzbraucot objekta figūrai, parādītos saraksts ar tā iepriekšējām vērtībām).

Lai attēlotu objekta vārdu, nepieciešams programmas pirmkods, jo koda kompilācijas laikā mainīgā vārds var tikt pazaudēts. Objekta vārdu var iegūt vai nu to meklējot pirmkodā, vai arī, kodu kompilējot, jālieto īpaši karogi, kas nodrošina papildus atkļūdošanas informācijas saglabāšanu.

Lai attēlotu objekta iepriekšējās vērtības, tās ir jāuzkrāj. Šeit atsevišķi aplūkojami 2 gadījumi:

1) tiek saglabātas un attēlotas vērtības noteiktos programmas pārtraukumpunktos, ko norādījis lietotājs;
2) tiek saglabātas objektu vērtības katrā programmas izpildes solī.

2.1.3. Programmas reversa atkļūdošana

Grafisks atkļūdotājs varētu piedāvāt lietotājam vispirms programmu izpildīt, to nepārtraucot pārtraukumpunktos, un pēc tam aplūkot atmiņas objektu stāvokli kā attēlu sēriju. Lietotājs varētu aplūkot sagatavotos attēlus pa soļiem uz priekšu un atpakaļ. Pārtraukumpunkti šeit varētu tikt uztverti kā īpašas iezīmes attēlu sērijā – lai lietotājam būtu vieglāk nokļūt interesējošā programmas izpildes brīdī.

Lai nodrošinātu šādu attēlu sērijas parādīšanu, būtu jāuzkrāj dati par atmiņas objektu vērtībām pa soļiem visā programmas izpildes laikā. Datu uzkrāšanai varētu būt 2 pieejas:

1) katrā solī saglabāt visu informāciju par atmiņas objektiem;
2) katrā solī saglabāt izmaiņas attēlojamajos datos pret iepriekšējo soli.

Izmantojot pirmo pieeju, uzkrātais datu apjoms būtu lielāks. Izmantojot otro pieeju, būtu jārisina, kā nodrošināt datu reversu attēlošanu.

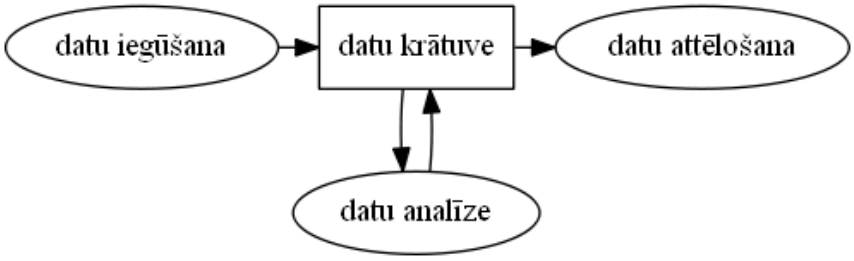
2.2. Grafiska atkļūdotāja loģiskā arhitektūra

Atmiņas objektu grafisku attēlošanu var aplūkot kā trīs secīgu darbību kopumu, kuru starprezultāti tiek saglabāti datu krātuvē (skatīt 2.2. attēlu).

Vispirms nepieciešams iegūt datus par atmiņas objektiem vienā vai vairākos programmas izpildes brīžos. Šos datus saglabā datu krātuvē.

Datu analīzi var uzskatīt par atsevišķu sastāvdaļu, jo vienkāršākos risinājumos šis posms nav obligāts. Tomēr datu analīzes lietošana sniedz lielākas iespējas kļūdas atklāšanā,

atvieglojot darbu lietotājam. Datu analīzei tiek izmantota datu iegūšanas posmā uzzinātā informācija par atmiņas objektiem. Datu analīzes rezultātā datu krātuvē tiek saglabāta papildus informācija.



2.2. att. **Atmiņas objektu grafiskas attēlošanas sastāvdaļas**

Datu attēlošana ir pēdējais posms, kas krātuvē saglabāto informāciju attēlo lietotājam ērtā un pārskatāmā veidā.

Datu krātuve rūpējas par datu uzkrāšanu un saskarnēm ar minēto posmu risinājumiem. Datu krātuves forma atkarīga no risinājuma apjoma un sarežģītības. Datu krātuve šī darba izpratnē var būt jebkas, kur var saglabāt un pēc tam no tā iegūt datus – vienkāršiem risinājumiem dati var tikt glabāti vienkārši programmas atmiņā vai teksta datnē, sarežģītākiem risinājumiem tā var būt arī datu bāze.

Pēc autores domām, šāda struktūra ļauj sadalīt risināmo problēmu mazākās problēmās un aplūkot tās atsevišķi, neatkarīgi no pārējo daļu risinājumu. Ja tiktu definētas saskarnes ar datu krātuvi, katru no posmiem varētu realizēt kā pilnīgi neatkarīgu projektu.

Tālāk šajā nodaļā atsevišķi aplūkota katra no komponentēm, idejiski doti tās risinājuma varianti un minēti aspekti, kur vēl būtu veicami tālāki pētījumi.

2.3. Datu iegūšana

Datu iegūšanu nosacīti var sadalīt divos soļos: visu to objektu saraksta iegūšana, kuri ir "redzami" attiecīgajā brīdī un objektu atribūtu (piemēram, datu tipa un vērtības) iegūšana.

Minimālā nepieciešamā informācija par katru no objektiem:

1) objekta adrese atmiņā – lai nodrošinātu iespēju attēlot norādes starp objektiem;
2) objekta datu tips – lai objektu analīzē atšķirtu, vai objekts ir masīvs, norāde vai vienkāršs mainīgais, un lai objektu attēlotu lietotājam saprotamā veidā;
3) objekta vērtība konkrētā laika brīdī – lai iegūtu visus objektus (ja objekts ir norāde – jāiegūst objekts, uz ko tas norāda) un lietotājam attēlotu objektu vērtības un norādes starp objektiem.

Papildus šiem atribūtiem būtu noderīgi iegūt objekta mainīgā vārdu lietotāja deklarētiem mainīgajiem, lai objektu attēlotu lietotājam atpazīstamā veidā.

Iegūstot datus, jāņem vērā, ka atmiņas objekti var tik saglabāti stekā vai dinamiskajā atmiņā. Datus par atmiņas objektiem programmas izpildes laikā var iegūt, vai nu no pašas atkļūdojamās programmas, vai sadarbojoties ar kādu atkļūdotāju. Šajā nodaļā netiek aplūkota tāda iespēja, kā atsevišķa pilnīgi jauna atkļūdotāja izveidošana. Ja izvēlēts šāds risinājums, var uzskatīt, ka izveidotā atkļūdotāja darbība ietvers funkcionalitāti, kas aprakstīta 2.3.1. apakšpunktā.

2.3.1. Datu iegūšana no jau esoša atkļūdotāja

Šīs pieejas gadījumā datu iegūšanas komponente datus iegūst no kāda cita atkļūdotāja. Datu komponentei jānodrošina saziņa ar atkļūdotāju – iespēja tam padot komandas (jautāt pēc mainīgā vai tā atribūtiem) un saņemt atbildes. Saziņu ar esošu atkļūdotāju iespējams risināt divos veidos:

1) datu iegūšanas komponenti veidojot kā paplašinājumu kādai atkļūdotāja saskarnei, kura jau satur rīkus atmiņas objektu atribūtu iegūšanai,
2) datu iegūšanas komponenti veidojot kā atsevišķu programmu, kura kā procesu darbina kādu komandrindas atkļūdotāju, padodot tam komandas un apstrādājot saņemtās atbildes.

Pirmajā veidā vieglāka būs objektu saraksta un objektu atribūtu iegūšana – šie dati jau būs atkļūdotājā. Nepieciešamības gadījumā jāpapildina atmiņas objektu atribūtu iegūšana un saglabāšana, ja datu attēlošanai nepietiek ar jau esošajiem datiem.

Otrajā gadījumā visu “redzamo” objektu saraksta iegūšana notiek, jautājot atkļūdotājam visu lokālo mainīgo sarakstu un papildinot to ar objektiem, uz kuriem norāda sarakstā esošie objekti (atkārtoti – kamēr nav jaunu objektu, ko pievienot sarakstam). Šeit jāpievērš uzmanība, lai sarakstā kāds no atmiņas objektiem nenonāk vairāk kā vienu reizi (piemēram, ja divi lokālie mainīgie norāda uz vienu un to pašu atmiņas objektu). Arī objektu atribūtu iegūšana notiek, jautājot tos atkļūdotājam.

Detalizēta komandrindas atkļūdotāja darbināšana C++ valodā un datu iegūšana no GDB atkļūdotāja aprakstīta šī darba 3. nodaļā.

Lai iegūtu datus no atkļūdotāja, jāpārliecinās, ka atkļūdojamā programma kompilēta, iekļaujot papildus atkļūdošanas informāciju. Piemēram, GCC kompilators iekļauj šādu informāciju tad, ja norādīts karodziņš, kas sākas ar -g (pieejami karodziņi -g, -ggdb, -gstabs+ un citi) [GDBOpt]. Ja atkļūdojamā programma kompilēta bez šādas pazīmes, datus no

atkļūdotāja iegūt pēc 3. nodaļā aprakstītās shēmas nav iespējams. Šī bakalaura darba ietvaros sīkāk nav pētīts, vai vispār iespējams iegūt datus no atkļūdotāja bez šādas pazīmes.

Brīžus, kad programmas izpildei jāapstājas, lai iegūtu informāciju, lietotājs definē ar pārtraukumpunktiem. Lielākā daļa atkļūdotāju piedāvā pārtraukumpunktus izveidot vai nu kādas funkcijas sākumā vai norādot pirmkoda rindas numuru. Atbilstoši būtu jāpielāgo šīs komponentes lietotāja saskarne. Būtu jādod lietotājam iespēju norādīt vienu vai vairākas atkļūdojamās programmas pirmkoda datnes. Pēc tam vai nu jāparāda kodā atrodamo funkciju saraksts vai jāattēlo pirmkods, lai lietotājs varētu iezīmēt pārtraukumpunktus.

Komponentei jānodrošina pārtraukumpunktu pārvaldība. Tai jāatceras, kādus pārtraukumpunktus lietotājs izveidojis (ideālā gadījumā – attēlojot tos lietotājam arī saraksta veidā un ļaujot lietotājam tos ieslēgt un atslēgt arī šajā sarakstā), jādod lietotājam iespēju izdzēst izveidotos pārtraukumpunktus un, uzsākot programmas izpildi, jāpadod šī informācija atkļūdotājam.

2.3.2. Atmiņas objektu "atcerēšanās" programmas izpildes laikā un to atribūtu iegūšana pēc adreses

Atmiņas objektu vērtību programmas izpildes laikā iespējams iegūt no pašas programmas – vaicājot pēc mainīgā nosaukuma vai pēc tā adreses. Izvēloties šo pieeju, visu objektu sarakstu iespējams iegūt:

1) statiskajiem mainīgajiem – analizējot programmas izejas kodu, atrodot mainīgo deklarēšanas vietu;
2) dinamiskajiem mainīgajiem – pārdefinējot operatorus *new* un *delete*, tādejādi piefiksējot mainīgo izveidošanas un dzēšanas brīžus.

Datu ieguves komponentes veidošana ar šo pieeju C++ programmām sīki aprakstīta Andreja Vihrova kursa darbā "Datu struktūru grafiskā attēlošana C++ programmām" un pēc tam turpināta Agņa Āriņa bakalaura darbā "Dinamiska programmu atmiņas objektu analīze".

Andreja Vihrova izstrādātās bibliotēkas "memchart" darbības pamatā ir C++ operatoru *new* un *delete* pārdefinēšana. Katrs šo operatoru izsaukums attiecīgo objektu vai nu ieliek vai izņem no objektu tabulas. Tabulā tiek glabāta norāde uz objekta tipu un objekta adrese atmiņā.

Bibliotēkas lietotājam jāpapildina programmas pirmkods, klašu definēšanā iezīmējot laukus, par kuriem būtu jāattēlo informācija. Iezīmēšanai tiek izmantoti makrosi. Piemēram, klases definēšanā iekļautas šādas koda rindas:

"MC_BEGIN(Node)
MC_MEMBER(data)

MC_MEMBER(left)

MC_MEMBER(right)

MC_END”

iezīmē klasi “Node” un tās laukus “data”, “left” un “right” [Vih, 2010].

Savukārt programmas kodā iekļauta komanda “memchart::update()” nozīmē, ka šajā programmas izpildes vietā tiks atjaunoti dati objektu tabulā [Vih, 2010].

Agnis Āriņš savā bakalaura darbā turpina uzlabot “memchart” bibliotēku, padarot to ērtāku lietotājam. Agis Āriņš piedāvā lietotāja pirmkodu anotēt automātiski, lai lietotājam katras klases definēšanā nebūtu jāiekļauj minētie makrosi. [Āri, 2011].

Agņa Āriņa darbā pieminēti arī šīs pieejas trūkumi:

1) bibliotēka neietver šablonu mehānisma atbalstu;
2) nav iespēja iegūt informāciju par statiski izveidotiem mainīgajiem, kuru radīšanai netiek lietots *new* operators;
3) nav iespēja piefiksēt mainīgo vārdiskos nosaukumus, jo izpildes laikā C++ tie ir pilnībā pazuduši [Āri, 2011].

2.3.3. Pieeju salīdzinājums

Katrai no minētajām pieejām ir savas stiprās un vājās puses. Atkļūdotāja izmantošanas galvenā priekšrocība ir tā, ka lietotājam nav nepieciešams papildus anotēt pirmkodu, lai attēlotu informāciju. Arī to programmas vietu iezīmēšana, kurās iegūstami dati par atmiņas objektiem, ir vienkāršāka (no lietotāja viedokļa).

Savukārt izvēloties pieeju, ka dati tiek iegūti tieši no programmas, lietotājam nav jāsatraucas par atkļūdotāja instalēšanu.

Jāpiebilst, ka izveidot universālu risinājumu, kurš māk sazināties ar visiem atkļūdotājiem praktiski nav iespējams. Lai arī idejiski atkļūdotāju komandas ir līdzīgas (ievietot pārtraukumpunktu, darbināt programmu, attēlot mainīgā vērtību), to sintakse un saņemto atbilžu saturs ir atšķirīgs. Tas nozīmē, ka katram no atkļūdotājiem jāveido sava klase, kas nodrošina saziņu ar konkrēto atkļūdotāju.

Dati, kas iegūti, izmantojot vienu vai otru pieeju var būt atšķirīgi. Piemēram, izmantojot datu iegūšanu no atkļūdotāja, nav iespējams iegūt datus par tiem atmiņas objektiem, kas ir izveidoti programmas izpildes laikā, bet uz tiem neved neviens ceļš no lokālajiem mainīgajiem.

2.4. Datu analīze

Datu analīze ietver saistītu datu objektu struktūras analīzi un iespējamo kļūdu meklēšanu. Struktūras analīzes laikā varētu tikt noskaidrots datu struktūras veids (koks, saraksts, grafs u.tml.). Šāda informācija varētu būt noderīga datu attēlošanas posmā, ļaujot parādīt datus lietotājam atpazīstamā veidā.

Datu struktūras noskaidrošana var tikt izmantota arī iespējamo kļūdu atklāšanai. Ja grafisks atkļūdotājs dod iespēju lietotājam norādīt datu struktūru, kādai atbilst noteikts programmas atmiņas objektu kopums (to varētu izdarīt vai nu atsevišķā konfigurācijas datnē, vai lietotāja saskarnē), varētu salīdzināt, vai lietotāja norādījumi saskan ar analīzē iegūto. Piemēram, ja lietotājs ir iecerējis sarakstu, bet tā vidū ir cikls, datu analīzes laikā varētu tikt atrasta kļūdainā norāde un saglabāta informācija par to datu krātuvē. Datu attēlotājs to varētu izmantot, lai īpaši iezīmētu kļūdaino norādi.

Analizējot iegūtos datus, būtu iespējams atrast arī citas kļūdas, piemēram, atmiņas izmantošanas un piekļuves kļūdas, kas parasti rodas, nekorekti darbojoties ar norādēm. Tālāk minēti daži kļūdu veidi un mēģināts spriest par to, kādi dati nepieciešami, lai tās atklātu.

2.4.1. Atmiņas noplūde

C++ valodā rakstītām programmām atmiņas noplūde ir bieži sastopama kļūda, jo programmētājam pašam jānodrošina dinamiski izdalītas atmiņas atbrīvošana. Atmiņas noplūde notiek, kad programma pieprasa dinamiski piešķirt atmiņu, ko pēc tam neatbrīvo. [Oua, 1995].

Citās programmēšanas valodās par šo problēmu rūpējas drazu savācējs. Atmiņas noplūde ir problēma tajās programmēšanas valodās, kur jaunu objektu inicializēšana un esošo dzēšana ir pilnībā atstāta programmētāja ziņā.

Iespējas detektēt atmiņas noplūdi analizētas Agņa Āriņa bakalaura darbā. Lai detektētu atmiņas noplūdi, nepietiek ar visu "redzamo" (to, kas sasniedzami konkrētā programmas izpildes brīdī) atmiņas objektu iegūšanu. Šeit interesējošie objekti ir tieši tie, kas nav sasniedzami t.i. tie objekti, kuri programmas izpildei vairs nav vajadzīgi, bet to izmantotā atmiņa nav korekti atbrīvota.

Izmantojot 2.3.2. apakšpunktā aprakstīto pieeju datu iegūšanai un saglabājot iepriekšējā soļa datus, šo kļūdu iespējams atrast, jo minētā pieeja "pazaudē" objektus no sava redzesloka tikai tad, kad tie tiek iznīcināti. Izmantojot datu ieguvi no esoša atkļūdotāja, objekti tiek "pazaudēti" tad, kad tie vairs nav redzami. Lai detektētu atmiņas noplūdi, datus iegūstot no

atkļūdotāja, nepieciešams papildus pētījums par to, kā piekļūt šiem objektiem.

Pilnīgi visas atmiņas noplūdes nav iespējams detektēt. Agnis Āriņš apraksta algoritmu atmiņas noplūžu detektēšanai, kas balstās uz to, ka tiek meklēti objekti, uz kuriem nekas nenorāda. Šāds algoritms neatradīs atmiņas noplūdi, piemēram, gadījumā, kad divi objekti rāda viens uz otru [Āri, 2011].

2.4.2. Kļūdainas norādes

Norāde ir objekts, kura vērtība ir kāda cita objekta adrese. Programmas izpildes laikā, programma var nonākt stāvoklī, kad kāda norāde ir kļūdaina, šādos gadījumos:

1) objekts, uz kuru rāda norāde, tiek dzēsts;
2) kļūda norāžu vērtību aritmētikā.

Abos gadījumos mēģinājuma darboties ar objektu, uz kuru tiek norādīts, rezultāti ir neprognozējami. Ja atmiņa izdalīta citam objektam, kas ir redzams šajā programmas apgabalā, tā vērtība var tikt izmainīta un lietotājs pat var nepamanīt, ka notikusi kļūda. Savukārt ja objekts kam iedalīta atmiņa nav "redzams", tad tiks izdota atmiņas piekļuves pārkāpuma kļūda (*segmentation fault*, *access violation*).

Agņa Āriņa darbā aprakstītas iespējas šo kļūdu detektēt, ja dati iegūti tieši no programmas. Ja dati tiek iegūti no atkļūdotāja, rodas līdzīga problēma kā atmiņas noplūžu gadījumā. Šī darba ietvaros sīkāk nav pētīts, kā rīkoties ar šo kļūdu, iegūstot datus no atkļūdotāja.

2.4.3. Rīki atmiņas analīzei

Tāpat kā datu iegūšanai var lietot esošu atkļūdotāju vai datu attēlošanai – rīku grafu veidošanai, arī atmiņas analīzei var izmantot esošus rīkus. "Valgrind" ietvars piedāvā rīkus, kas spēj automātiski atklāt vairākas atmiņas pārvaldības kļūdas. Populārākais no tiem ir "Memcheck", kas spēj atklāt vairākas ar atmiņas izmantošanu saistītas kļūdas, kas ir bieži sastopamas C un C++ programmās [Valgrind]. "Memcheck" iespējams darbināt no komandrindas, kas dod iespēju to integrēt grafiskajā atkļūdotājā pēc līdzīgas shēmas, kā sazinoties ar komandrindas atkļūdotāju.

Atrodami arī citi rīki, kas piedāvā atklāt atmiņas noplūdes un citas saistītas kļūdas. Šie rīki un iespējas tos integrēt nav sīkāk pētīti šī darba ietvaros.

2.5. Datu attēlošana

Datu attēlošanai grafiskā atkļūdotājā būtu jābūt pēc iespējas tādai, it kā attēlu būtu

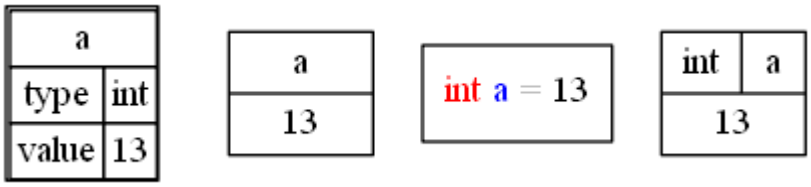
zīmējis pats programmētājs.

2.5.1. Vienkāršu datu attēlošana

Vienkāršus datu tipus vislabāk būtu attēlot kā ģeometriskas figūras (elipses vai taisnstūrus). Norādes vajadzētu attēlot, nevis, parādot to vērtību (adresi), bet gan, novelkot bultiņu uz objektu, uz kuru tiek norādīts.

Figūrās attēlojamās informācijas daudzums un izkārtojums varētu būt atkarīgs no lietotāja vēlmēm un atkļūdojamās programmas specifikas. Tāpēc būtu labi veidot attēlošanas komponenti konfigurējamu, ļaujot lietotājam izvēlēties piemērotāko attēlošanas veidu.

2.3. attēlā parādīti piemēri, kā varētu tikt attēlots atmiņas objekts, kurā tiek glabāta mainīgā *a* vērtība 13, mainīgā datu tips ir *int*.



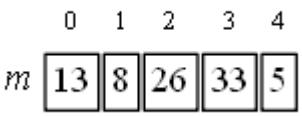
2.3. att. **Dažādi atmiņas objekta attēlošanas veidi**

Objekta attēlošanai var izmantot gan dažādu izkārtojumu, gan arī lietot krāsas, lai nodalītu atspoguļojamās informācijas veidus.

2.5.2. Saliktu datu tipu attēlošana

Salikta datu tipa objektu vajadzētu attēlot kā vienu figūru, nevis, izmantojot bultas tā elementu attēlošanai, lai attēlojums maldīgi nebūtu līdzīgs norādēm. Tā kā klases lauki ir klases sastāvdaļa, nevis "bērni" vai "kaimiņi" grafa izpratnē, pēc autores domām, klasi vajadzētu attēlot līdzīgi kā pārējos datu tipus – kā noslēgtu ģeometrisku figūru, piemēram, kā taisnstūri. Savukārt klases laukus (to vārdus, vērtības un pēc vajadzības – datu tipus) varētu attēlot šīs figūras iekšienē.

Masīvu vajadzētu attēlot līdzīgi kā ierasts to zīmēt – tā elementi secīgi blakus viens otram, to indeksi norādīti virs vai zem elementa. Piemērs masīva attēlošanai parādīts 2.4. attēlā.



2.4. att. **Masīva attēlošanas variants**

Vairāki varianti iespējami garāku masīvu attēlojumā, kuru visi elementi neietilpst attēlam atvēlētajā laukumā. Viens no variantiem būtu tos attēlot "čūskas" veidā – sākot no kreisās puses uz labo, tad uz leju, tad no labās puses atpakaļ uz kreiso u.t.t.

Cits variants būtu attēlot masīva sākumu un beigas, vidū norādot pārtraukumu (piemēram, lietojot daudzpunkti). Tāpat varētu attēlot tikai tās masīva vērtības, kuras lietotājam varētu būt "interesantas", pārējās paslēpjot (zem daudzpunktes vai cita apzīmējuma). Vērtības, kas lietotājam varētu būt "interesantas" būtu, piemēram:

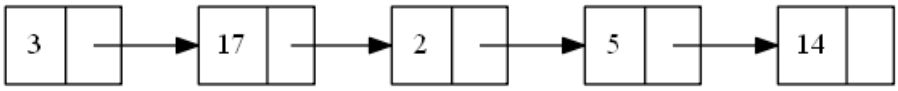
1) nesen mainījušās vērtības;
2) ja masīva vērtības pārsvarā vienveidīgas – atšķirīgās vērtības;
3) ja masīva vērtības savā starpā salīdzināmas – noteikts skaits lielāko vai mazāko vērtību.

Ja netiek attēlotas visu masīva elementu vērtības, būtu jānodrošina paslēpto vērtību attēlošana lietotājam pēc pieprasījuma (piemēram, piedāvājot izvēlēties intervālu, kuru attēlot).

2.5.3. Zināmāko datu struktūru attēlojums

Lai arī minimāli vajadzīgā prasība būtu objektus izkārtot tā, lai tie pēc iespējas nepārklātos, saistītu objektu grupas būtu ieteicams izkārtot tā, lai lietotājam būtu skaidrāks, kādu datu struktūru tās reprezentē.

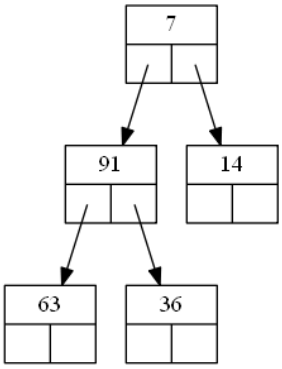
Saistītu sarakstu parasti attēlo, elementus zīmējot kā taisnstūrus, kas sadalīti 2 daļās – vienā daļā ir elementa vērtība, no otras daļas centra iziet bulta, kas rāda uz nākamo elementu (skatīt 2.5. attēlu).



2.5. att. **Saistīta saraksta attēlošana**

Divvirzienu saistītu sarakstu attēlo līdzīgi, tikai elements sastāv no trīs daļām, no kurām vidējā satur elementa vērtības, bet no abu malējo daļu centriem iziet bultas uz blakus elementiem [Cor, 2009].

Koka struktūru parasti attēlo, saknes elementu novietojot attēla augšā un pa līmeņiem izvietojot pārējos koka elementus [Cor, 2009]. Ja koka mezglam ir fiksēts bērnu skaits, zem vērtības varētu attēlot šī skaita rūtiņas, lai uzsvērtu, kurš bērns atbilst kurai norādei (skatīt 2.6. attēlu).



2.6. att. **Koka stuktūras attēlošana**

Ja koka mezglam bērnu skaits nav noteikts (piemēram, bērni saglabāti sarakstā), norāžu bultas uz visiem bērniem varētu iziet no mezgla apakšējās malas vidus.

Patvaļīga grafa izkārtojumam galvenais nosacījums būtu pēc iespējas mazāka virsotņu elementu un norāžu pārklāšanās. Eksistē vairāki grafu izvietošanas algoritmi. Tie detalizēti aprakstīti Andreja Vihrova darbā [Vih, 2010].

2.5.4. Konfigurējama objektu attēlošana

Lietotājam varētu piedāvāt izvēlēties ne tikai objekta attēlošanas veidu, bet arī to, kādi objekti vispār tiek attēloti. Sarežģītākas atkļūdojamās programmas gadījumā attēlojamo objektu skaits varētu būt liels. Iespējams, ne visi objekti attēlo lietotājam svarīgu informāciju. Lietotājam varētu piedāvāt paslēpt atsevišķus objektus (vai nu izvēloties no saraksta, vai interaktīva attēlojuma gadījumā – caur izvēlni, kas būtu pieejama ar labās peles pogu klikšķinot uz paša objekta). Tāpat varētu piedāvāt paslēpt arī objektu grupas. Objektu kārtojums grupās varētu būt interesants pēc šādām pazīmēm:

1) pēc datu tipa – piemēram, nerādīt primitīvos datu tipus, attēlot tikai saliktas datu struktūras, lietotāja definētus tipus un norādes;

2) pēc to redzamības apgabala – piemēram, attēlot tikai tos mainīgos, kas deklarēti tā bloka vai funkcijas ietvaros, kurā apstājusies programmas izpilde, un dinamiskās atmiņas mainīgos uz kuriem tie norāda;

3) pēc objektu izmaiņu vecuma – piemēram, parādot tikai tos objektus, kuru vērtības mainījušās noteikta intervāla (soļu skaita) laikā.

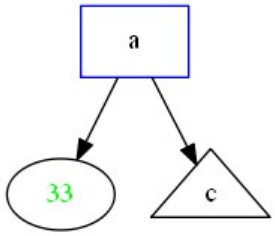
Krāsas varētu izmantot ne tikai objektu etiķešu attēlojumā, bet arī iezīmējot kādus noteiktus objektus. Iekrāsot varētu objektus, kas mainījušies pēdējā soļa laikā, tādējādi pievēršot tiem lietotāja uzmanību. Ja grafiskajā atkļūdotājā tiktu realizēta datu analīze, dažādi iekrāsot varētu arī potenciālās kļūdas.

2.5.5. Esošu grafu attēlošanas rīku lietojums

Atmiņas objektu attēlošanai iespējams izmantot kādu jau esošu grafu attēlošanas rīku vai bibliotēku. Viens no populārākajiem grafu attēlošanas rīkiem ir "graphviz". "Graphviz" ir interesants ar to, ka lietojams gan kā neatkarīgs rīks caur komandrindas izsaukumu, gan arī to iespējams pievienot programmai kā bibliotēku.

Grafu uzdošana "graphviz" rīkam notiek, izmantojot *dot* valodu. Tā ir vienkārša un ātri apgūstama. *Dot* valodā aprakstīta grafa piemērs parādīts 2.7. attēlā.

```
digraph {
a [shape=box color=blue]
b [fontcolor=green label="33"]
c [shape=triangle]
a->b
a->c
}
```



2.7. att. **Dot valodā aprakstīts grafs un tā attēls**

"Graphviz" piedāvātās funkcionalitātes klāsts ir apjomīgs. Izmantojot šo rīku, iespējams attēlot gan saistītus, gan nesaistītus grafus. Attēlojot grafa elementus, iespējams mainīt gan to figūru, gan krāsu, gan izmēru un citus atribūtus. Grafu elementu izvietošanai iespējams izvēlēties vairākas programmas. Divas populārākās ir *dot* un *neato*.

Dot attēlo orientētu grafu hierarhiskā struktūrā. Iespējams izvēlēties struktūras virzienu (no augšas uz leju vai no labās puses uz kreiso) kā arī norādīt, kuri elementi attēlojami vienā līmenī [Dot].

Neato attēlo grafu, izmantojot uz spēkiem balstītu algoritmu. Kā izejas punkts tiek izmantots *dot* izkārtojums. Tiek radīts virtuāls fizikāls modelis un darbināts iteratīvs risinātājs, kas meklē zemas enerģijas konfigurāciju [Neato].

"Graphviz" vai līdzīga rīka lietošana grafiska atkļūdotāja datu attēlošanas komponentē atvieglotu tās izveidi. Grafu elementu izkārtošana nav triviāls uzdevums. Parasti minētie rīki, pietiekamā līmenī apgūstot to lietojumu, piedāvā ļoti labus rezultātus.

2.6. Datu krātuve

Datu krātuves realizācija tieši atkarīga no veidotā grafiskā atkļūdotāja apjoma un piedāvātās funkcionalitātes. Pamata funkcionalitātes nodrošināšanai datu krātuvi var realizēt grafiskā atkļūdotāja atmiņā.

Lielāka loma datu krātuvei ir tādos risinājumos, kuros nepieciešams uzkrāt informāciju

par objektiem vairākos soļos (piemēram, lai detektētu atmiņas noplūdes kļūdas vai lai piedāvātu apgriezto atkļūdošanu). Galvenā izvēle kas šeit izdarāma – vai dati katrā solī tiks saglabāti pilnībā vai tiks saglabāta tikai informācija par izmaiņām, salīdzinot ar iepriekšējo soli.

Ja izvēlēts saglabāt informāciju tikai par izmaiņām, risinājums būs atkarīgs no tā, kāda datu ieguve izvēlēta. Ja izvēlēts datus iegūt no atkļūdojamās programmas, iegūtie dati reprezentēs aktuālo stāvokli. Tie jāsalīdzina ar iepriekšējā soļa rezultātiem, lai iegūtu mainīto informāciju.

Ja izvēlēts datus iegūt no esoša atkļūdotāja, var sīkāk meklēt iespējas no atkļūdotāja saņemt nevis pilnu atmiņas objektu ainu, bet tikai mainītos objektus. Kā piemēru var minēt GDB atkļūdotāja mainīgo objektus. Pēc katra soļa izpildes iespējams atkļūdotājam padot komandu "-var-update", kas atgriež informāciju par mainītajiem objektiem. Lai realizētu šo risinājumu, papildus jāpēta, kā iegūt informāciju par jauniem objektiem.

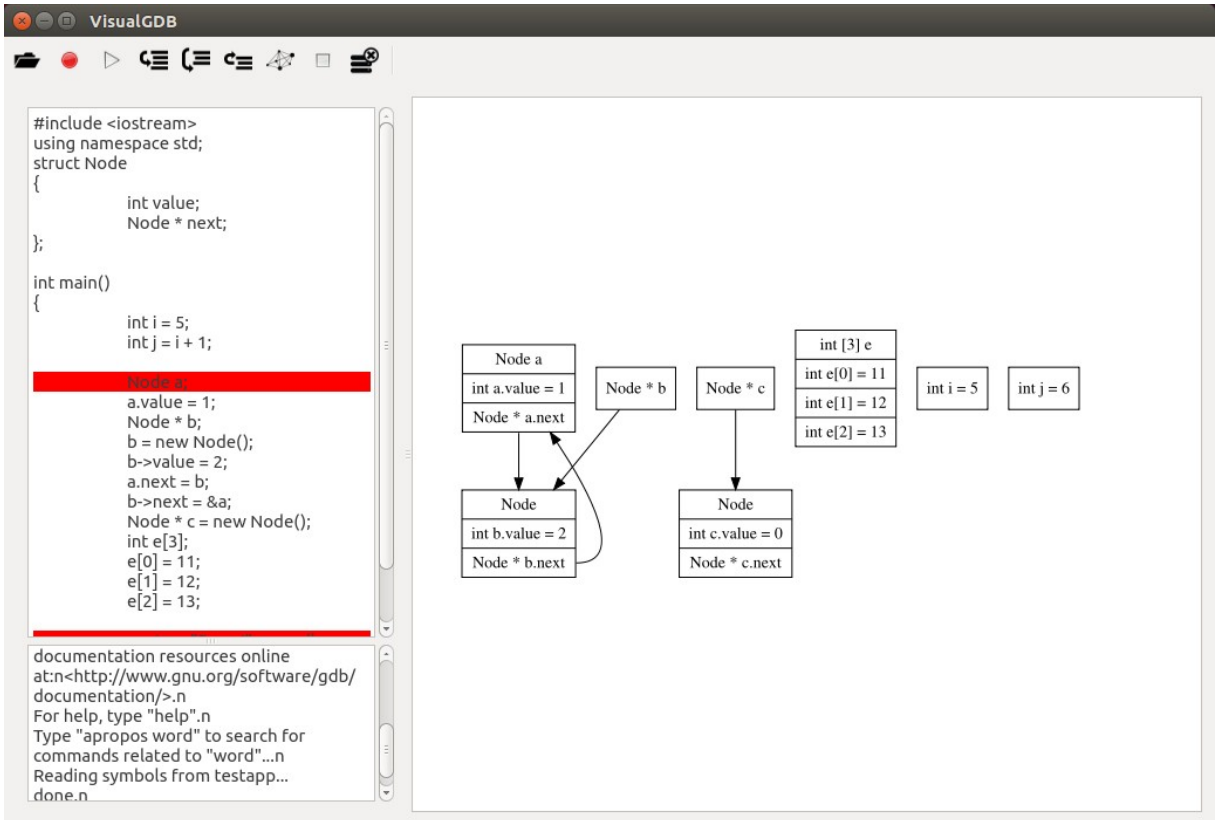
3. IZVEIDOTĀ RISINĀJUMA DETALIZĒTS APRAKSTS

Andrejs Vihrovs un Agnis Āriņš savos darbos izvēlējušies datu iegūšanu tieši no izpildāmās programmas un datu attēlošanai pašu veidotus algoritmus. Lai pilnīgāk varētu novērtēt atšķirības starp datu ieguves un attēlošanas veidiem un spriest par to stiprajām un vājajām pusēm, šajā bakalaura darbā izvēlēts risinājums, kurā dati tiek iegūti no esoša komandrindas atkļūdotāja, savukārt datu attēlošanai izmantots "graphviz" rīks.

Šajā nodaļā aprakstīts izstrādātais prototips – vienkāršota grafiskā atkļūdotāja versija, kurā realizētas atsevišķas 2. nodaļā minētās idejas. Dažām risinājumā iekļautajām idejām dots detalizēts risinājums arī ārpus prototipā tieši iekļautās funkcionalitātes.

3.1. Prototipa atbalstītā vide un arhitektūra

Bakalaura darbā izstrādāta Qt ietvara lietotne, kas atkļūdo programmu, ļaujot lietotājam soli pa solim virzīties pa programmas izpildi un attēlot atmiņas objektus vēlamajos momentos (skatīt 3.1 att.).



3.1. att. **Izstrādātā rīka ekrānuzņēmums**

Risinājums veidots C++ valodā Qt ietvarā. Ietvars izvēlēts, jo tajā salīdzinoši viegli

veidot lietotāja saskarni, ietvars satur ērtas bibliotēkas, turklāt lietotne viegli pielāgojama dažādām platformām.

Risinājums veidots Linux vidē, bet Qt ietvars nodrošina, ka risinājumu var viegli pielāgot arī Windows un Mac vidēm, atbilstoši pievienojot iekļautās bibliotēkas.

No komandrindas atkļūdotājiem izvēlēts GDB, jo tas ir viens no populārākajiem komandrindas atkļūdotājiem un tā MI saskarnei pieejamas bibliotēkas, kas palīdz atkļūdotāja atbildes parsēšanā.

Objektu attēlošanai izvēlēts "graphviz" rīks tā vienkāršās saskarnes un dažādo attēlošanas formātu dēļ. Risinājumā objektu attēlošana notiek, izveidojot grafa aprakstu valodā dot, kurš tiek padots caur komandrindu izsauktajam "graphviz" rīkam. Ģenerētais attēls tiek parādīts lietotājam izveidotajā lietotnē.

Izstrādātās lietotnes ekrāns sadalīts vairākās daļās, kuru izmērus lietotājs var mainīt, bīdot daļu atdalītāju (skatīt 3.1. att.). Ekrāna kreisās puses augšējā daļā tiek parādīts atkļūdojamās programmas izejas kods (teksts nav rediģējams, bet lietotājs var tajā iezīmēt rindas, pirms kurām jāveido pārtraukumpunkti), apakšējā daļā tiek parādīti paziņojumi, kas saņemti no atkļūdotāja, bet labajā pusē tiek attēloti atmiņas objekti. Virs programmas teksta izvietota rīkjosla ar pogām, kas nodrošina komandu padošanu atkļūdotājam.

Ar izveidoto rīku iespējams atkļūdot programmas, kas rakstītas valodās, ko atbalsta GDB atkļūdotājs. Lai programmu varētu atkļūdot ar izveidoto risinājumu, tai jābūt kompilētai ar pazīmi "-g", kas nodrošina papildus atkļūdošanas informācijas izveidi.

3.2. Komandrindas atkļūdotāja darbināšana valodā C++

Neizmantojot speciālas bibliotēkas, komandrindas atkļūdotāja darbināšana ir atšķirīga, programmējot Windows vai Linux operētājsistēmām.

3.2.1. Atkļūdotāja darbināšana Windows operētājsistēmā

Izsmeļošs apraksts par to, kā izsaukt bērna procesu un sazināties ar to dots Microsoft MSDN lapā [MSDN].

Lai sazinātos ar atkļūdotāju, nepieciešams radīt bērna procesu, kas darbina komandrindu, un izveidot turus (handle), kas nodrošina pieeju bērna procesa standarta ievadei (lai padotu komandas atkļūdotājam) un izvadei (lai saņemtu atbildes no atkļūdotāja).

Pielāgojot doto parauga kodu [MSDN], var iegūt programmu, kas secīgi:

1) iedarbina komandrindu, atverot tajā GDB atkļūdotājā testa programmu;
2) uzstāda pārtraukumpunktu testa programmas *main* funkcijā;

3) darbina testa programmu līdz pārtraukumpunktam;
4) turpina testa programmas darbināšanu līdz beigām;
5) iziet no GDB atkļūdotāja.

Pilns pielāgotās programmas pirmkods dots 1. pielikumā. Aplūkosim atsevišķus nozīmīgus koda fragmentus.

```
// Create a pipe for the child process's STDOUT.
if (!CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0))
    cout << "ERROR in StdoutRd CreatePipe" << endl;
else cout << "STDOUT Pipe created.\n";

// Ensure the read handle to the pipe for STDOUT is not inherited.
if (!SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0))
    cout << "ERROR in Stdout SetHandleInformation" << endl;
```

3.2. att. **Kods kanāla izveidošanai**

3.2. attēlā redzams koda fragments, kas izveido kanālu uz bērna procesa standarta izvadi. Tā kā bērns izveides laikā mantos vecāka turus, jānodrošina, ka bērns nemanto turi savas standarta izvades lasīšanai. Līdzīgi tiek izveidots kanāls uz bērna procesa standarta ievadi, kur attiecīgi bērnam nav jāmanto turis savas ievades rakstīšanai.

```
// Create the child process.
bSuccess = CreateProcess(NULL,
    szCmdline,     // command line
    NULL,          // process security attributes
    NULL,          // primary thread security attributes
    TRUE,          // handles are inherited
    0,
    NULL,          // use parent's environment
    NULL,          // use parent's current directory
    &siStartInfo,  // STARTUPINFO pointer
    &piProcInfo);  // receives PROCESS_INFORMATION
```

3.3. att. **Kods bērna procesa izveidei**

3.3. attēlā parādīts kods bērna procesa izveidei. Kā redzams, funkcijai *CreateProcess* var padot dažādus atribūtus, atkarībā no vajadzības. Aplūkotajā programmā mainīgajam *szCmdline* pirms šīs funkcijas izsaukuma tiek piešķirta vērtība "gdb MD1test.exe --interpreter=mi2", t.i., tieši tāds teksts, kāds tiktu lietots, ja GDB atkļūdotāju izsauktu no komandrindas. Teksta daļa “gdb” norāda uz izsaucamo komandu – GDB atkļūdotājs. “MD1test.exe” ir programma, kas tiks atkļūdota. Parametrs “--interpreter=mi2” norāda, ka GDB atkļūdotājs tiks darbināts caur GDB/MI saskarni, kas ir mašīnorientēta un paredzēta sistēmām, kas sazinās ar GDB atkļūdotāju [GDBMi].

3.4. attēlā parādīts kods, kas nodrošina iespēju nosūtīt bērna procesam ziņu. Kā redzams – ziņa tiek ierakstīta bērna standarta ievadē. Mainīgais *dwRead* pēc funkcijas *WriteFile* izsaukšanas satur informāciju par to, cik garu ziņu izdevies nosūtīt. Vajadzības gadījumā, izmantojot šo vērtību, var atklāt kļūdas, kas radušās ziņas nosūtīšanas laikā.

```
void WriteToPipe(char * message)
{
    DWORD dwRead, dwWritten;
    WriteFile(g_hChildStd_IN_Wr,
              message,
              sizeof(char) * strlen(message),
              &dwWritten,
              NULL);
    cout << "Message sent :" << message << endl;
}
```

3.4. att. **Kods ziņas nosūtīšanai bērna procesam**

3.5. attēlā parādīts kods, kas nodrošina iespēju saņemt atbildi no bērna procesa. Tā kā GDB atbildes var būt sadalītas vairākās rindās, atbildes beigas var atpazīt, ja pēdējie saņemtie simboli ir "(gdb)". Tādējādi šīs funkcijas laikā tiek lasīts bērna standarta izvades kanāls, kamēr tiek saņemti šie simboli. Jāpiebilst, ka šāds lasīšanas veids nav absolūti drošs, jo var gadīties kļūda bērna procesā un atbilde var nebeigties ar šādu simbolu virkni. Tas nozīmē, ka, realizējot saziņu ar bērna procesu šādā zemā līmenī, jāpārdomā lasīšanas beigu atpazīšana pēc vairākiem parametriem.

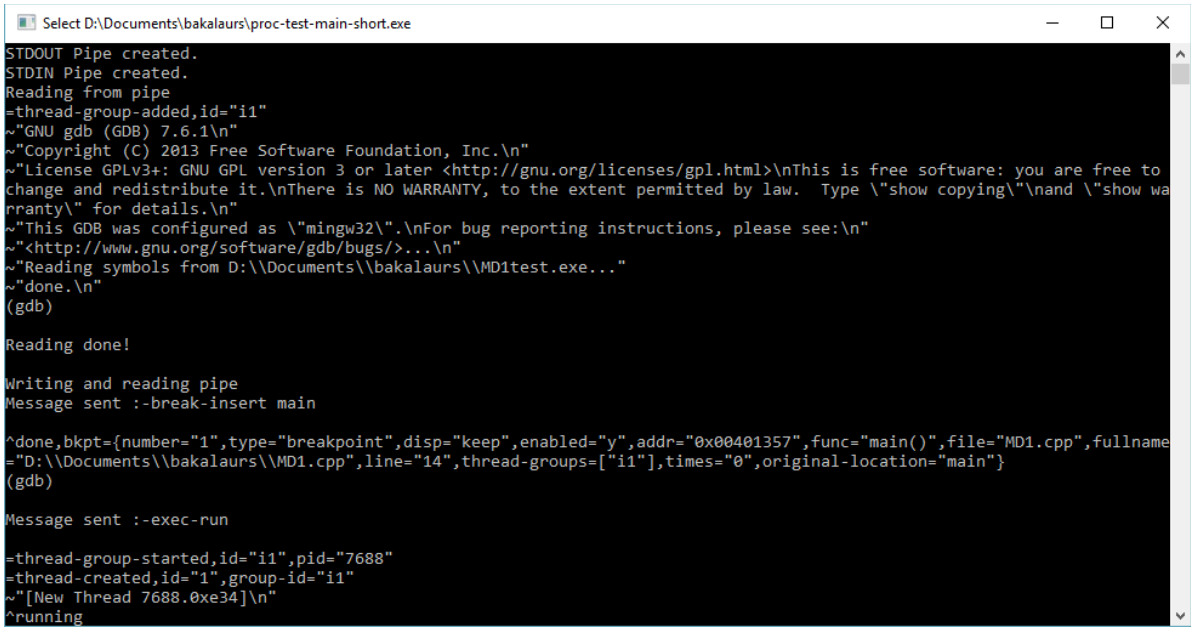
```
string ReadFromPipe()
{
    DWORD dwRead, dwWritten;
    CHAR chBuf[BUFSIZE] = "";
    BOOL bSuccess = FALSE;
    string ret = "";
    string gdb = "(gdb)";
    char checkGdb[6] = "";
    while (gdb.compare(0, 5, checkGdb, 0, 5))
    {
        bSuccess = ReadFile( g_hChildStd_OUT_Rd, chBuf, BUFSIZE, &dwRead, NULL);
        chBuf[dwRead] = '\0';
        memcpy(checkGdb, &chBuf[dwRead-8],5);
        ret += chBuf;
    }
    return ret;
}
```

3.5. att. **Kods atbildes saņemšanai no bērna procesa**

Komandas, kas tiek padotās GDB atkļūdotājam atbilst GDB/MI saskarnes sintaksei. Piemēra programmā GDB atkļūdotājam tiek padotas šādas komandas:

1) "-break-insert main" – ievieto pārtraukumpunktu galvenās (*main*) funkcijas sākumā;

2) “-exec-run” – startē atkļūdojamo programmu, tās izpilde turpināsies līdz ievietotajam pārtraukumpunktam;
3) “q” – beidz GDB atkļūdotāja darbību.



3.6. att. **Piemēra programmas darbināšanas ekrānuzņēmums - fragments no atbildēm, kas saņemtas no GDB atkļūdotāja**

3.6. attēlā redzams piemēra programmas izvada logs – sākuma daļa no programmas darbības. Redzams, ka sekmīgi izdevies izveidot kanālus un bērna procesu un atkļūdojamajā programmā ievietot pārtraukumpunktu. Var redzēt, ka GDB atbildes MI saskarnes gadījumā nav tik viegli pārskatāmas, kā darbinot GDB atkļūdotāju noklusētajā veidā, bet tās satur palīgsimbolus, kas atvieglo atbilžu parsēšanu un vēlāku izmantošanu.

3.2.2. Atkļūdotāja darbināšana Linux operētājsistēmā

Linux operētājsistēmā saziņa ar bērna procesu algoritmiski notiek līdzīgi, atšķiras funkciju izsaukumi bērna radīšanai un kanālu atvēršanai. Pilns programmas pirmkods, kas balstīts uz piemēru [Fork] parādīts 2. pielikumā.

```
//create 2 pipes to talk from child to parent and parent to child
int child_to_parent[2];
int parent_to_child[2];
pipe(child_to_parent);
pipe(parent_to_child);
```

3.7. att. **Kods kanālu atvēršanai**

3.7. attēlā redzams kods kanālu atvēršanai. Katru kanālu raksturo *int* tipa masīvs ar 2 vērtībām: pirmā – lasīšanai, otra – rakstīšanai. Tā kā nepieciešama saziņa abos virzienos (no

bērna uz vecāku un no vecāka uz bērnu), tiek veidoti divi kanāli.

Bērna process tiek radīts ar funkcijas *fork* izsaukumu. Izsaucot šo funkciju tiek radīts jauns process, kurš ir kopija procesam, kas izsauca funkciju *fork*. Tam procesam, kurš tika radīts, funkcijas vērtība ir 0, savukārt procesam kurš radīja bērna procesu, funkcija atgriež bērna procesa identifikatoru (*pid*). Tādējādi gan pamatprocesa, gan bērna procesa izejas kods ir viens un tas pats, atšķirt to, vai darbošanās notiek bērna vai pamatprocesā turpmāk var izmantojot funkcijas *fork* atgriezto vērtību.

Pēc bērna procesa radīšanas, bērna process izsauc funkciju *gdb_agent* (skatīt 3.8. attēlā).

```
void gdb_agent (int* child_to_parent, int* parent_to_child)
{
     dup2(parent_to_child[0], STDIN_FILENO);
     dup2(child_to_parent[1], STDOUT_FILENO);
     execlp("gdb", "gdb", "testapp", "--interpreter=mi2", NULL);
};
```

3.8. att. **Bērna procesa funkcijas kods**

Funkcija *dup2* sasaista radītos kanālus ar bērna procesa standarta ievadi un izvadi [Friedl]. Savukārt funkcija *execlp* aizstāj bērna procesu ar GDB atkļūdotāja izsaukumu komandrindā.

```
ilze@ilze-Ubuntu: ~/Documents
ilze@ilze-Ubuntu:~/Documents$ ./proc-test
CHILD CREATED 4334
thread-group-added,id="i1"
~"GNU gdb (Ubuntu 7.7-0ubuntu3) 7.7\n"
~"Copyright (C) 2014 Free Software Foundation, Inc.\n"
~"License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>\
nThis is free software: you are free to change and redistribute it.\nThere is NO
 WARRANTY, to the extent permitted by law.  Type \"show copying\"\nand \"show wa
rranty\" for details.\n"
~"This GDB was configured as \"x86_64-linux-gnu\".\nType \"show configuration\"
for configuration details."
~"\nFor bug reporting instructions, please see:\n"
~"<http://www.gnu.org/software/gdb/bugs/>.\n"
~"Find the GDB manual and other documentation resources online at:\n<http://www.
gnu.org/software/gdb/documentation/>.\n"
~"For help, type \"help\".\n"
~"Type \"apropos word\" to search for commands related to \"word\"...\n"
~"Reading symbols from testapp..."
~"done.\n"
(gdb)
Writing break

^done,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x0000000
0004008a5",func="main()",file="testapp.cpp",fullname="/home/ilze/Documents/testa
```

3.9. att. **Piemēra programmas darbināšanas ekrānuzņēmums - fragments no atbildēm, kas saņemtas no GDB atkļūdotāja**

Turpmāka darbošanās ar atkļūdotāju notiek tāpat kā Windows sistēmas gadījumā – bērna procesam tiek padotas komandas ar *write* izsaukumu, savukārt atbildes tiek saņemtas ar

read izsaukumu. Programmas darbības rezultāta fragments redzams 3.9. attēlā.

Kā redzams, saņemtās atbildes nav atkarīgas no operētājsistēmas. Arī atbilžu saņemšana un parsēšana apstrādājama līdzīgi.

3.3. Komandrindas atkļūdotāja darbināšana izmantojot Qt ietvaru

Qt ietvaru var lietot dažādu lietojumprogrammu izstrādē. Qt atbalsta vadošās platformas. Izmantojot Qt, iespējams izstrādāt gan darbvirsmas lietotnes, gan mobilās vai iegultās lietotnes. Ietvars satur Qt bibliotēkas un vairākus izstrādes rīkus, ieskaitot integrēto izstrādes vidi.

Komandrindas atkļūdotāja izsaukšana, izmantojot Qt bibliotēkas, ir ļoti vienkārša, turklāt uzrakstīto izejas kodu iespējams kompilēt gan darbam Windows gan Linux vidē.

3.3.1. Procesa izveide un sākšana

Lai darbotos ar procesu, nepieciešams iekļaut bibliotēku "QProcess". Aplūkosim atsevišķi 3.10. attēlā parādītā koda rindas.

```
QString exeFile = "testapp";
QString exeDirectory = "/home/ilze/Documents";
QString program = Qstring(
         "gdb %1 -iex \"set auto-load safe-path /\" --interpreter=mi2")
         .arg(exeFile);
QProcess process = new QProcess();
process->setProcessChannelMode(QProcess::MergedChannels);
process->setWorkingDirectory(exeDirectory);
process->start(program);
```

3.10. att. **Procesa izveide un startēšana, izmantojot Qt bibliotēkas**

Simbolu virknes *exeFile* un *exeDirectory* norāda atkļūdojamās programmas atrašanās vietu. Lietotnē šo mainīgo vērtības būtu iegūstamas no lietotāja. Simbolu virkne *program* ir procesa izpildāmā programma. To veido:

1) gdb – komandrindā izpildāmais atkļūdotājs;
2) mainīgā *exeFile* vērtība – atkļūdojamās programmas nosaukums;
3) -iex "set auto-load safe-path /" - pazīme, ka šajā gdb sesijā atkļūdotājam visas datnes jāuztver kā uzticamas [GDBSec];
4) --interpreter=mi2 – norāde, ka GDB lietojams caur MI saskarni.

Process tiek izveidots ar operatoru *new*. Procesa metodes *setProcessChannelMode* izsaukums ar argumentu *Qprocess::MergedChannels* norāda, ka procesam standarta izvades un standarta kļūdu kanāli tiks apvienoti, kļūdas izvadot standarta izvadē. Procesa metodes

setWorkingDirectory izsaukums uzstāda norādīto mapi kā procesa darba mapi (process atkļūdojamās programmas izpildes un izejas koda failiem mēģinās piekļūt no šīs mapes). Procesa metode *start* uzsāk procesu, izpildot argumentā padoto programmu.

3.3.2. Sloti un signāli

Qt ietvars notikumu apstrādei lieto signālus un slotus. Signāls tiek raidīts, kad notiek kāds notikums. Slots ir funkcija, kas tiek izsaukta, kā atbilde noteiktam signālam [Qt]. Piemēram, kad lietotājs nospiež pogu "Aizvērt", tiek raidīts signāls par pogas nospiešanu, kurš ir piesaistīts lietotnes slotam, kurš nodrošina lietotnes aizvēršanu.

Qt logrīkiem un citām klasēm ir definēti vairāki signāli un sloti, bet lietotājs var definēt papildus signālus un slotus (atvasinot Qt logrīkus). Arī lietotāja veidotām klasēm iespējams definēt savus signālus un slotus un tos savienot.

Piedāvātajā risinājumā saziņa ar atkļūdotāju notiek nevis secīgi, kā aplūkots iepriekš, bet gan izmantojot Qt ietvara signālus un slotus. Atkļūdotājam tiek padotas komandas, negaidot atbildes. Atbildes, kas saņemtas no atkļūdotāja, tiek apstrādātas asinhroni – kad process raida signālu *readyReadStandardOutput*, atbildes tiek nolasītas no tā standarta izvades un padotas atbilžu parsētājam.

3.3.3. Komandu nosūtīšana atkļūdotājam

Piedāvātajā risinājumā komandu nosūtīšanai izveidota klase *GDBMIWriter*. Galvenajā programmā inicializējama šīs klases instance, padodot tai norādi uz procesu, kas darbina GDB atkļūdotāju. Komandu nosūtīšana atkļūdotājam notiek, izsaucot šīs klases metodes. Risinājumā iekļautās metodes neaptver visu GDB piedāvāto funkcionalitāti. Iekļautas tās komandas, kas nodrošina vajadzīgo pamatfunkcionalitāti – pārtraukumpunktu ievietošanu, programmas izpildi, apstājoties pārtraukumpunktos, porgrammas izpildi pa solim, atmiņas objektu attēlošanu. Pēc autores domām risinājums veidots tā, ka tajā ietvertā funkcionalitāte ir viegli papildināma, pievienojot papildus komandas.

Būtiskākās atkļūdotāja komandas un tām atbilstošās klases *GDBMIWriter* metodes attēlotas 3. pielikumā [GDBMi].

Mainīgo objekti ir MI saskarnes objektorientēti elementi, kas nodrošina vienkāršu un ērtu iespēju aplūkot un mainīt izteiksmju vērtības. Mainīgā objekts tiek identificēts pēc vārda. Veidojot mainīgā objektu, tam var piešķirt vārdu un izteiksmi. Izteiksme var būt atkļūdojamās programmas mainīgais vai jebkāda izteiksme, kas atbilst atkļūdojamās programmas valodai [GDBMi].

Mainīgo objektiem ir hierarhiska koka struktūra. Mainīgā objektam, kas atbilst saliktam

tipam, ir bērna objekti. Piemēram, mainīgā objektam, kas raksturo struktūru, bērna objekti ir katrs no struktūras elementiem. Ja arī kāds no tiem ir struktūra, tam arī ir bērna objekti. Mainīgā objekts, kam nav bērnu (mainīgo objektu koka lapa) parasti raksturo kādu iebūvēto tipu. Bērnu objekti tiek radīti tikai tad, ja tas tiek pieprasīts ar komandu "-list-children". Mainīgā objektu dzēšot, tiek dzēsti arī tā bērni [GDBMi].

GDB dokumentācijā aprakstīta komanda "-var-update", kas atgriež tos mainīgo objektus, kuru vērtības mainījušās kopš pēdējās šīs komandas izsaukuma [GDBMi]. Piedāvātajā risinājumā šī pieeja nav izmantota, jo nav pilnībā izpētīta šīs komandas darbība, pārvietojoties pa redzamības apgabaliem. Pēc autores domām, šīs komandas izpēte un pielietošana varētu dot ieguldījumu piedāvātā risinājuma uzlabošanā. Tā atvieglotu izmaiņu attēlošanu lietotājam – būtu vieglāk (ātrāk) iegūstama informācija par izmaiņām, tādējādi lietotājam varētu īpaši iezīmēt tās vērtības, kas mainījušās. Šī pieeja būtu noderīga arī scenārijā, kad programmas izpildi fonā pārtrauc pa soļiem, neattēlojot rezultātu lietotājam. Iegūtās vērtības tiek attēlotas pēc tam, lietotājam dodot iespēju pārskatīt tās turp un atpakaļ. Ja izmaiņas atmiņas objektos tiktu iegūtas šādā veidā, būtu mazāks virstēriņš.

Pēc autores domām, nepieciešamības gadījumā viegli iespējams izveidot klases citu atkļūdotāju vai GDB atkļūdotāja citas saskarnes izmantošanai. Galvenajā programmā būtu jānomaina klases *GDBMIWriter* instances inicializācija ar citas klases instances inicializāciju un izveidotais risinājums šādi būtu viegli pielāgojams dažādiem atkļūdotājiem.

3.3.4. Atkļūdotāju atbilžu apstrāde

Atkļūdotāja atbilžu pirmsapstrādei piedāvātajā risinājumā izmantota bibliotēka *gbdwire*. Šī bibliotēka veidota saziņai ar GDB atkļūdotāju caur MI saskarni. Bibliotēka piedāvā metodes, kas atšķetina atkļūdotāja atbildes līdz divu simbolu virkņu pārim (*key-value* formā) [Brasko].

Piedāvātajā risinājumā šī bibliotēka izmantota klasē *GDBMIParser.* Šīs klases instance tiek inicializēta risinājuma galvenajā programmā, kā argumentus padodot norādes uz galveno programmu un logrīku (teksta redaktoru), kurā atspoguļot informatīvus un kļūdu paziņojumus.

Klases *GDBMIParser* metode *doParse* tiek izsaukta, kad nolasīta atbilde no atkļūdotāja, padodot kā argumentu nolasīto atbildi.

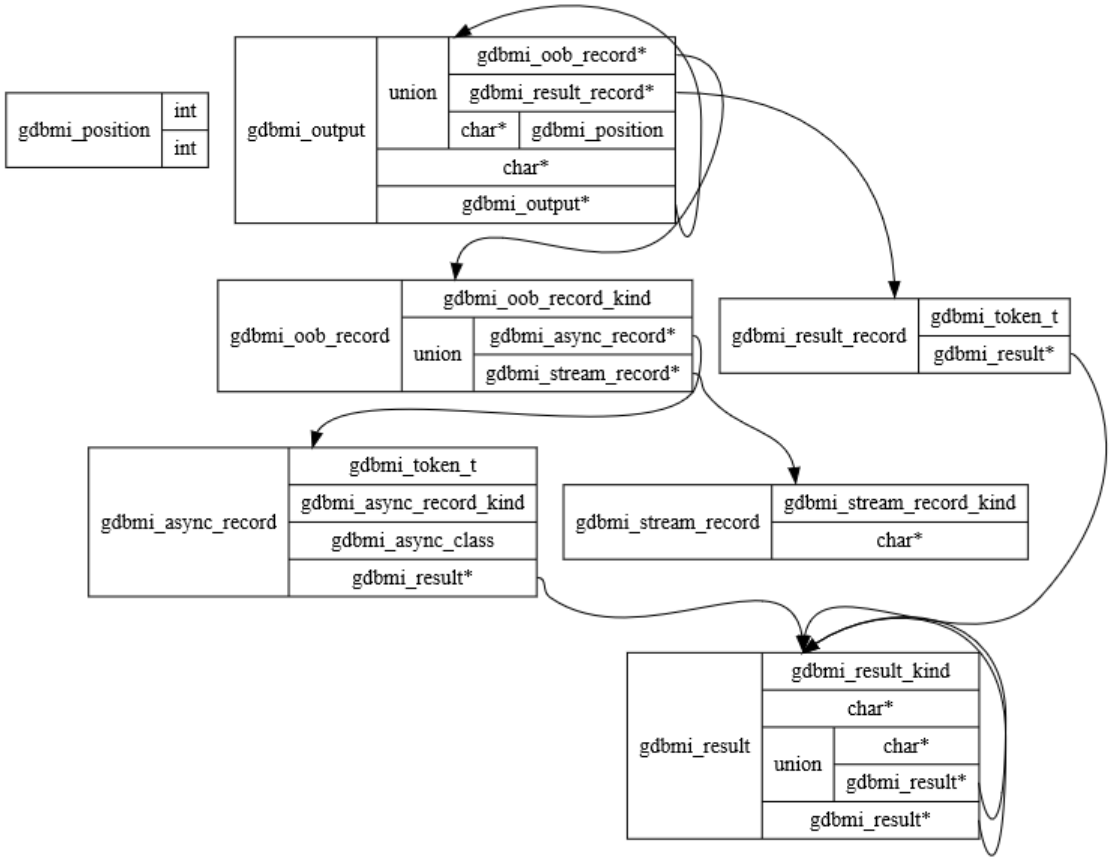
Klases *GDBMIParser* sadarbība ar bibliotēku *gdbwire* notiek vairākos soļos. Klases konstruktorā tiek izveidota struktūra *gdbmi_parser_callbacks callbacks = {this, parser_callback}*, ko veido norāde uz klases instanci un norāde uz funkciju, kura tiks

izsaukta, kad bibliotēka būs apstrādājusi kādu atbilžu bloku. Šī struktūra tiek padota kā arguments bibliotēkas funkcijai *gdbmi_parser_create*, kas atgriež bibliotēkas galveno objektu – struktūru *gdbmi_parser*. Funkcijas *parser_callpack* realizācija parādīta 3.11. attēlā.

```
void parser_callback(void *context, struct gdbmi_output *output)
{
    GDBMIParser * parser = (GDBMIParser * )context;
    parser->unwrapAnswer_Output(output);
    gdbmi_output_free(output);
}
```

3.11. att. **Funkcija parser_callback**

Funkcija *parcer_callback* izsauc klases *GDBMIParser* funkciju *unwrapAnswer_Output* kā argumentu padodot tai saņemto struktūru *gdbmi_output*. Pēc šīs struktūras apstrādes, tā tiek iznīcināta ar bibliotēkas funkciju *gdbmi_output_free*.



3.12. att. **Struktūras gdbmi_output sastāvs**

Klases *GDBMIParser* metodē *doParse* tiek izsaukta bibliotēkas funkcija *gdbmi_parser_push*, kā argumentus padodot konstruktora laikā izveidoto struktūru *gdbmi_parser* un apstrādājamo atkļūdotāja atbildi. Kad bibliotēkas funkcija apstrādājusi atbildi, tā izsauc funkciju *parser_callback*.

Bibliotēkas struktūra *gdbmi_output* sevī ietver citas bibliotēkas struktūras. Detalizēta struktūra parādīta 3.12. attēlā.

3.3.5. Atmiņas objektu informācijas ieguve un saglabāšana

Atmiņas objektu apstrādei piedāvātajā risinājumā izveidotas klases *VDVariable* un *VDVariableList*. Klase *VDVariable* par katru iegūto atmiņas objektu saglabā šādu informāciju:

1) varobject – mainīgā objekta vārds GDB procesā;
2) name – mainīgā pilns vārds, pēc kura programmā iespējams piekļūt atmiņas objektam (piemēram, struktūras elementam tas būs formā – *struktūras nosaukums.elementa nosaukums*);
3) shortname – mainīgā īsais vārds programmā (piemēram, struktūras elementam – tikai elementa nosaukums);
4) type – mainīgā datu tips (arī, ja tas ir lietotāja definēts datu tips);
5) value – atmiņas objekta vērtība – mainīgā vērtība simbolu virknes veidā;
6) address – atmiņas objekta adrese;
7) parentPrefix – vecāka vārds (piemēram, struktūras elementam – struktūras vārds);
8) numChildren – atmiņas objektam atbilstošā mainīgā objekta bērnu skaits;
9) isNamed – pazīme, ka objektam ir "vārds" programmas kontekstā (piemēram, ja programmā ir rinda "Node * n = new Node();", tad tiks izveidoti divi atmiņas objekti – viens ar tipu "Node *" un vārdu "n", otrs ar tipu "Node"; pirmajam objektam šī pazīme būs ar vērtību "true", otrajam ar vērtību "false").
10) isPointer – pazīme, ka objekts ir norāde;
11) members – saraksts ar norādēm uz klases *VDVariable* objektiem (atmiņas objektiem, kas reprezentē norādes un vienkāršus datu tipus, tas būs tukšs, saliktiem datu tipiem tas saturēs norādes uz elementu objektiem);
12) parent – norāde uz klases *VDVariable* objektu, kurš ir šī objekta vecāks (ja objektam nav vecāku, šī elementa vērtība ir NULL).

Klase *VDVariableList* rūpējas par visu atmiņas objektu iegūšanu, apstrādi un attēlošanu lietotājam. Pilns klases pirmkods dots 4. pielikumā. Aplūkosim būtiskākos klases elementus un metodes.

Lai pārvaldītu visus atmiņas objektus, klase *VDVariableList* uztur trīs attēlojumus (*QMap* klases instances):

1) nameMap (ar tipu QMap<QString, VDVariable*>), kas atmiņas objekta vārdu sasaista ar norādi uz to;

2) addressMap (ar tipu QMap<QString, VDVariable*>), kas atmiņas objekta adresi sasaista ar norādi uz to;
3) tokenMap (ar tipu QMap<int, VDVariable*>), kas marķieri sasaista ar atmiņas objekta norādi.

Šie attēlojumi ļauj viegli iegūt norādi uz atmiņas objektu, zinot tā vārdu, adresi vai marķieri, kas nosūtīts atkļūdotājam, vaicājot datus par šo atmiņas objektu.

Lai izprastu, kā notiek visu redzamo atmiņas objektu iegūšana, secīgi aplūkosim klases *VDVariableList* lietotās metodes.

Metode *void createVariable(char * name)* tiek izsaukta, kad saņemts lokālo mainīgo saraksts no atkļūdotāja. Tiek pārbaudīts, vai saņemtais mainīgā vārds jau nav vārdu attēlojumā *nameMap.* Ja vārds tur nav jau iekļauts – izveido atmiņas objektu, izsaucot funkciju *addVarObject*, tai kā argumentu padodot saņemto vārdu *name.*

Metode *void addVarObject(char * name, VDVariable * parent, VDVariable * pointer)* kā argumentus saņem atmiņas objekta vārdu, un norādes uz objekta vecāku vai objektu, kas norāda uz izveidojamo objektu. Tiek izveidoti divi marķieri – atmiņas objekta atribūtu un adreses iegūšanai. Tālāk tiek izveidota klases *VDVariable* instance, kuras vārds atkarīgs no objekta vecāka veida (ja nav vecāka – vienkārši padotais vārds (*name*), ja objekts ir cita objekta elements, tad vārds ir formā *vecāka vārds.vārds*, ja objekts ir masīva elements, tad vārds ir formā *vecāka vārds[vārds]*). Izveidotajā atmiņas objektā tiek saglabāti arī abi marķieri (lai vēlāk būtu iespējams atpazīt, uz kuru objektu attiecas no atkļūdotāja saņemtās atbildes) un citi atribūti. Tālāk uz atkļūdotāju tiek nosūtītas komandas mainīgā objekta izveidei, izmantojot klases *GDBMIWriter* metodi *writeVarCreate*.

Metode *bool updateVariableAtributes(char * token, char * atribute, char * value)* tiek izsaukta, kad no atkļūdotāja tiek saņemta atbilde par mainīgā objekta izveidošanu, secīgi saņemot arī objekta atribūtus. Tā kā atbildes no atkļūdotāja tiek atšķetinātas līdz divām simbolu virknēm, tās arī tiek padotas šai metodei kopā ar no atkļūdotāja saņemto marķieri. Attēlojumā *tokenMap* tiek atrasts atbilstošais klases *VDVariable* objekts, kuram tiek uzstādīti saņemtie atribūti. Īpaši tiek apstrādāts gadījums, kad saņemta atmiņas objekta adrese, kuras vērtība ir tukša simbolu virkne, vai simbolu virkne "0x0". Šāda adrese norāda, ka atmiņas objekts patiesībā neeksistē. Šādu mainīgo var iegūt, piemēram, ja mēģina veidot mainīgā objektu atmiņas objektam, uz kuru it kā norāda kāda norāde, bet norādes vērtība nav inicializēta. Saņemot šādu adresi, mainīgā objekts tiek izmests no klases *VDVariableList* pārvaldītajiem objektiem.

Metode *void removeToken(char * token)* tiek izsaukta, kad pabeigta tāda atkļūdotāja

atbildes parsēšana, kurai ir norādīts marķieris. Ja attēlojumā *tokenMap* ir šis marķieris, tiek izsaukta metode *resolve* kā atribūtu padodot atbilstošo atmiņas objektu. Pēc tam tiek pārbaudīts, vai tiek gaidīta atbilde vēl par kādu marķieri (attēlojums *tokenMap* nav tukšs). Ja visas atbildes saņemtas, tiek izsaukta metode *listValues*.

Metode *void resolve(VDVariable * var)* veic divas funkcijas. Vispirms tiek pārbaudīts, vai iegūtais atmiņas objekts jau nav saņemts, ejot pa citu norāžu ceļu. Pārbaude tiek veikta pēc atmiņas objekta adreses (izmantojot attēlojumu *adressMap*). Ja atmiņas objekts ar šādu adresi jau eksistē, tiek izsaukta metode *resolveExistingAddress*. Pēc tam, atkarībā no mainīgā datu tipa tiek meklēti citi redzamie atmiņas objekti. Ja datu tips ir norāde (beidzas ar simbolu "*"), tiek izsaukta metode *getVariablePoint*. Ja mainīgais ir salikts datu tips (tam ir bērni), tiek izsaukta metode *getVariableMembers*.

Metode *void resolveExistingAddress(VDVariable * var)* apstrādā atmiņas objektus ar vienādām adresēm. Viens no objektiem tiek izmests. Priekšroka palikt tiek dota tādam atmiņas objektam, kuram ir programmētāja dots vārds (piemēram, ja vienam un tam pašam elementam var piekļūt tieši caur vārdu "a" vai izmantojot izteiksmi "b->next", paturēts tiks pirmais objekts). Izņēmums ir gadījums, kad bērna objekts ir vienā adresē ar vecāka objektu (piemēram, struktūras pirmā elementa adrese būs tāda pati kā pašai struktūrai). Šādā gadījumā bērna objekts netiek iekļauts adrešu attēlojumā *adressMap*, bet tas netiek izmests.

Metode *void getVariablePoint(VDVariable * var)* izsauc metodi *addVarObject* kā argumentus padodot saņemtā atmiņas objekta nosaukumu un tā norādi kā vecāka (*parent*) argumentu.

Metode *void getVariableMembers(VDVariable * var, char * keyword)* izveido marķieri un izsauc klases *GDBMIWriter* metodi *writeVarListChildren*. Simbolu virknes *keyword* izmantošana nepieciešama, lai atšķetinātus tādus saliktus datu tipus, kuru mainīgie pakārtoti zem modifikatoriem (piemēram, *private* un *public*). Šādiem datu tipiem galvenajam objektam ir viens vai vairāki bērni – minētie modifikatori, kuriem ir viens vai vairāki bērni – saliktā datu tipa elementi. Tā kā vēlāk šiem elementiem iespējams piekļūt lietojot sintaksi *vecāka vārds.bērna vārds* (un lietotājam tas arī būtu saprotamāk attēlojumā), modifikators tiek nosacīti ignorēts, bērna objektus tieši iekļaujot vecāka objekta sastāvā.

Metode *void createChildVariable(char * name, char * token)* tiek izsaukta, kad no atkļūdotāja saņemts objekta bērnu saraksts, kā argumentus padodot saņemtā bērna vārdu un marķieri. Metode atrod vecāka objektu pēc marķiera. Ja saņemtā bērna vārds ir modifikators (piemēram, *private* vai *public*), tiek vēlreiz izsaukta metode *getVariableMembers*, padodot tai modifikatoru kā atslēgvārdu. Ja saņemtā bērna vārds nav modifikators, tiek izsaukta metode

addVarObject, radod jaunu atmiņas objektu.

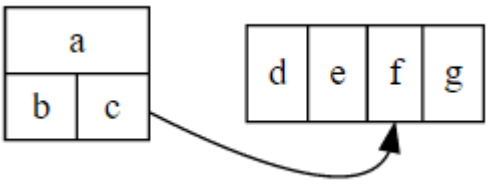
Metode *void listValues()* pārbauda, vai iegūti visi atmiņas objekti un izsauc metodes, kas nodrošina datu attēlošanu lietotājam.

3.4. Atmiņas objektu informācijas sagatavošana dot valodā

Graphviz atbalstītās valodas *dot* datnes sagatavošanu nodrošina klases *VDVariableList* metode *makeGraphFile()*. Metode izveido datni "graphviz.gv", kura satur grafa aprakstu *dot* valodā. Grafs tiek veidots, pārstaigājot attēlojumu *nameMap* un iekļaujot grafā mainīgo objektus kā virsotnes un norādes starp objektiem kā šķautnes.

Saliktu datu tipu – struktūru un masīvu attēlošanai virsotnei tiek uzstādīts atribūts "shape=record". Šī atribūta lietojums ļauj virsotnes etiķeti veidot saliktu, izmantojot simbolus "|", "{" un "}". Simbols "|" sadala virsotnes figūru daļās, savukārt iekavas "{}" norāda uz dalījuma virziena maiņu. Katram etiķetes elementam var piekārtot arī iezīmi, ievietojot to iekavās "<>". Veidojot šķautnes no vai uz šo virsotni, šķautnes raksturošanā var iekļaut šo iezīmi (aiz virsotnes nosaukuma un kola), tādējādi norādot, ka šķautnei jāiziet no šī elementa vai jāienāk šajā elementā. Visu šo simbolu lietojums parādīts 3.13. attēlā.

```
digraph {
rankdir=LR
node [shape=box]
n1 [shape=record, label="a|{b|c}"]
n2 [shape=record, label="{d|e|<f>f|g}"]
n1 -> n2:f
}
```



3.13. att. **Atribūta "shape=record" lietojums**

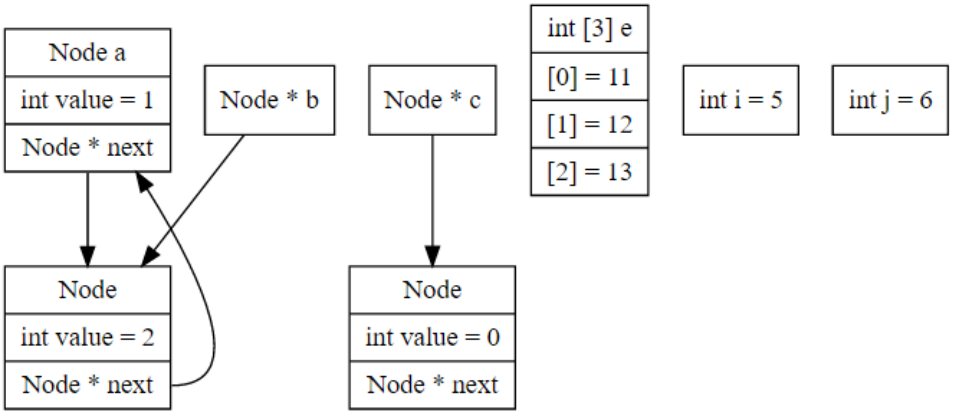
Lai atvieglotu apraksta sagatavošanu, klasei *VDVariable*, kas satur informāciju par attēlojamo objektu izveidotas papildus metodes:

1) *graphvizId*, kas atgriež virsotnes vai tās sastāvdaļas identifikatoru, ko izmantot šķautņu aprakstīšanai;
2) *graphvizLabel*, kas izveido attēlojamo etiķeti, saliktiem datu tipiem iekļaujot arī pakārtotos elementus.

Izveidotajā prototipā šobrīd iekļauts viens veids atmiņas objektu attēlošanai. Šobrīd izvēlēts vienkāršots objektu attēlošanas veids, kas attēlo pilnīgu informāciju par katru atmiņas objektu, ieskaitot tā datu tipu. Tipiskāko atmiņas objektu veidu (vienkāršs mainīgais, norāde, masīvs un struktūra) attēlojums parādīts 3.14. attēlā.

Kā redzams, vienkārša datu tipa mainīgie i un j parādīti kā taisnstūri, etiķetē iekļaujot

pilnu informāciju par datu tipu, mainīgā vārdu un vērtību.



3.14. att. **Atmiņas objektu attēlojums izstrādātajā prototipā**

Masīvs šobrīd attēlots vertikālā formā, pirmajā rindā norādot tā datu tipu (*int [3]* apzīmē *int* tipa masīvu ar 3 elementiem) un vārdu. Pārējās rindās attēloti masīva elementi, katram norādot tā indeksu un vērtību.

Norādes mainīgais attēlots kā taisnstūris, no kura iziet bultiņa. Ja norādes vērtība ir nulles norāde, kā piemēram *c->next* vērtība, tad bultiņa netiek attēlota. Norādes etiķete satur tās datu tipu un mainīgā vārdu, ja tas zināms.

Klases instance tiek attēlota kā taisnstūris. Tā augšējā daļā norādīts klases tips un mainīgā vārds. Pārējās rindās norādīti klases lauki, katru no tiem parādot kā vienkārša datu tipa mainīgo.

3.15. attēlā parādīts izveidotās programmas ģenerēts teksts 3.14. attēla veidošanai.

```
digraph {
node [shape=box]
v3 [shape=record, label = "{Node a|<v23>int value = 1|<v25>Node * next}"]
v3:v25 -> v15
v11 [label = "Node * b"]
v11 -> v15
v15 [shape=record, label = "{Node |<v33>int value = 2|<v35>Node * next}"]
v15:v35 -> v3
v5 [label = "Node * c"]
v5 -> v13
v13 [shape=record, label = "{Node |<v29>int value = 0|<v31>Node * next}"]
v7 [shape=record, label = "{int [3] e|<v17>[0] = 11|<v19>[1] = 12|<v21>[2] = 13}"]
v1 [label = "int i = 5"]
v9 [label = "int j = 6"]
}
```

3.15. att. **Programmas ģenerētais dot valodas teksts atmiņas objektu attēlošanai**

Tā kā izveidotā prototipa galvenais mērķis bija izprast grafiska atkļūdotāja veidotās iespējas, nevis izveidot lietojamu grafisku atkļūdotāju, noteikti nav uzskatāms, ka šāds objektu attēlojums ir vislabākais. Izveidotais dot datnes ģenerētājs noteikti papildināms,

iekļaujot vairākus attēlošanas veidus un uzlabojot objektu izvietojumu.

3.5. “graphviz” darbināšana attēla ieguvei

Izstrādātajā prototipā realizēta “graphviz” izsaukšana caur komandrindu. Šīs darbības kods parādīts 3.16. attēlā.

```
QString gvprogram = "dot -Tsvg -o graphviz.svg graphviz.gv";
QProcess * gvprocess = new QProcess();
gvprocess->setProcessChannelMode(QProcess::MergedChannels);
gvprocess->start(gvprogram);
gvprocess->waitForFinished();
gvprocess->close();
emit window->signalImageParced();
```

3.16. att. **Kods graphviz darbināšanai komandrindā**

Kā redzams, tiek izveidots jauns process tieši tāpat, kā tika veidota saziņa ar atkļūdotāju. Mainās izpildāmā programma – šoreiz izsauc programmu “dot”, ar pazīmi “Tsvg” norāda sagatavojamā faila formātu, karodziņš “-o” norāda, ka grafa attēls jāsaglabā datnē “graphviz.svg”, kā pēdējo norāda grafu aprakstošo datni “graphviz.gv”.

Pēc procesa startēšanas tiek sagaidīts, kad tas pabeigs savu izpildi, izmantojot metodi *waitForFinished*. Pēc tam process tiek aizvērts, un galvenajai programmai tiek padots signāls, ka izveidots attēls, kurš tai jāattēlo lietotājam.

Realizētajam risinājumam iespējami vairāki uzlabojumi. Sagatavoto “.gv” datni un pēc tam sagatavoto “.svg” datni nav obligāti saglabāt tādā veidā, kā tas parādīts. Datus nodot komandrindā izsauktajam “graphviz” rīkam un saņemt ģenerēto attēlu var, izmantojot kanālus.

“Graphviz” iespējams darbināt ne tikai caur komandrindu, bet arī iekļaut projektā kā bibliotēku. Izvēloties šādu risinājumu, grafu reprezentē noteikta datu struktūra, kurai var pievienot virsotnes un šķautnes, kā arī norādīt visus tos pašus atribūtus, kā sagatavojot šo informāciju dot valodā. Pēc grafa sagatavošanas tiek izsaukta šīs bibliotēkas metode, kas nodrošina grafa izvietošanu. Informācija par izvietojamu nolasāma grafa datu struktūrā.

Šāda pieeja dod iespēju izvietoto grafu parādīt lietotājam nevis vienkārši kā attēlu, bet gan dod iespēju to padarīt interaktīvu, attēlošanu veicot ar kādas citas bibliotēkas rīkiem. Tīmeklī atrodami avoti [Laz, 2010], kuros detalizēti aprakstīta šāda pieeja Qt ietvara lietotnēm. Lai arī šī darba ietvaros šī iespēja nav sīkāk pētīta, pēc autores domām, iespēja attēlojamos objektus padarīt interaktīvus varētu sniegt papildus iespējas piedāvātajā funkcionalitātē.

3.6. Iegūtā attēla parādīšana lietotājam

Attēlu vai citu grafisku elementu parādīšanai Qt lietotnē tiek lietots logrīks *QGraphicsView*. Attēla parādīšanu lietotājam veido divi posmi: logrīka sagatavošana, atverot programmu, un datu atjaunošana logrīkā. Logrīka inicializēšanas komandas parādītas 3.17. attēlā.

```
QGraphicsScene scene = new QGraphicsScene();
QGraphicsView view->setScene(scene);
QPixmap image;
QGraphicsPixmapItem item = scene->addPixmap(image);
```

3.17. att. **Kods logrīka, kurā tiks parādīts attēls, sagatavošanai**

Kā redzams, papildus minētajam logrīkam nepieciešams sagatavot vēl vairāku Qt bibliotēkas klašu objektus.

Kad saņemts signāls par to, ka attēls ir gatavs parādīšanai, nepieciešams attēlu parādīt sagatavotajā logrīkā (skatīt 3.18. attēlu).

```
QPixmap image("graphviz.svg");
item->setPixmap(image);
view->show();
```

3.18. att. **Kods attēla atjaunošanai logrīkā**

Lai parādītu logrīkā atmiņas objektus, tiek nolasīts “graphviz” rīka sagatavotais attēls. Tas tiek uzstādīts logrīka elementam, un logrīks tiek atjaunots ar komandu *show*.

Kā jau minēts iepriekšējā punktā, atmiņas objektus var attēlot interaktīvā veidā. Šādā gadījumā katrs no attēlojamajiem atmiņas objektiem atkal kļūst par mainīgo (iespējams izmantot jau esošo klasi *VDVariable*, papildinot to ar koordinātām un citu nepieciešamo informāciju), tam tiek definēti signāli un sloti (piemēram, kam jānotiek, ja lietotājs novietojis peli virs šī objekta) un tas tiek iezīmēts logrīkā.

REZULTĀTI

Bakalaura darbā apkopotas idejas grafiska atkļūdotāja funkcionalitātei, arhitektūrai un iespējamajiem risinājuma variantiem.

Padziļināti pētīta datu iegūšana no komandrindas atkļūdotāja. Darbā detalizēti parādīta atkļūdotāja darbināšana ar standarta C++ iespējām, bez papildus bibliotēku iekļaušanas gan Linux, gan Windows operētājsistēmām. Aprakstīta arī komunikācijas ar atkļūdotāju realizēšana, izmantojot Qt ietvaru.

Bakalaura darba tapšanas laikā izveidots grafiska atkļūdotāja vienkāršots prototips – Qt lietotne, kas sazinās ar komandrindas GDB atkļūdotāju, izmantojot tā MI saskarni. Lietotne spēj saņemt datus no atkļūdotāja par atkļūdojamās programmas atmiņas objektiem, darbināt atkļūdojamo programmu pa solim un attēlot atmiņas objektu vērtības noteiktā programmas izpildes brīdī. Atmiņas objektu attēlošanai izmantots “graphviz” rīks, to izsaucot caur komandrindu.

SECINĀJUMI

Atkļūdošana ir svarīga programmatūras izstrādes procesa sastāvdaļa. Uzskatāmāka atkļūdošana atvieglotu programmētāja darbu.

Darba izstrādes laikā netika atrasts publiski pieejams rīks, kas nodrošinātu meklēto funkcionalitāti. Rīks, kas citu autoru darbos ir minēts, kā labākais, – "DataDisplayDebugger" – ir novecojis. Tā pēdējā versija iznākusi 2009. gadā.

Var uzskatīt, ka grafisku atkļūdotāju veido trīs secīgas darbības: datu iegūšana, datu analīze un datu attēlošana. Tās savā starpā savieno datu krātuve, kas kalpo par katras darbības ieejas datiem un vietu, kur tiek saglabāts darbības rezultāts. Atkļūdotāja sadalīšana šādās loģiskās komponentēs, var atvieglot to izstrādi.

Grafisku atkļūdotāju iespējams realizēt, izmantojot dažādas pieejas. Darbā minētajām alternatīvajām idejām, katrai var atrast tādus ieguvumus, ko nesniedz citas alternatīvas. Ļoti iespējams, ka labs grafisks atkļūdotājs savā risinājumā ietvers vairākas pieejas, kuru sniegtais labums dažviet pārklāsies, lai nodrošinātu pēc iespējas pilnīgāku informāciju.

Kā liecina citu studentu darbos un šajā darbā sasniegtais rezultāts – grafiska atkļūdotāja izveide, kas vismaz daļēji aptvertu šī darba 2. daļā minēto funkcionalitāti, ir iespējama.

Grafiska atkļūdotāja veidošanā var izmantot jau gatavus rīkus, tos integrējot atkļūdotājā vai darbinot caur komandrindu. Kā piemērus var minēt "graphviz" rīku objektu attēlošanai un "Valgrind" rīkus datu analīzei. Ir ieteicams šo rīku piedāvātās funkcijas un iespējamo lietojumu atkļūdotājā turpināt pētīt.

Darbā minētas vairākas problēmas, kas netika sīkāk aplūkotas. Pētījumi par to, kā izveidot grafisku atkļūdotāju, noteikti ir turpināmi, jo problēma ir aktuāla un šobrīd nav atrodams publiski pieejams rīks, kas to risinātu.

PATEICĪBAS

Autore izsaka pateicību darba vadītājam Guntim Arnicānam par ieteikumiem tēmas izvēlē un darba koncepcijas izstrādē.

IZMANTOTĀ LITERATŪRA UN AVOTI

[Term] Akadēmiskā terminu datubāze [Tiešsaiste], pieejams: http://termini.lza.lv [skatīts 23.05.2016].

[GDBMi] "The gdb/mi Interface" [Tiešsaiste], pieejams: https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html [skatīts 23.05.2016].

[Bod, 2015] V. Bodrozic "Top 10 Most Common C++ Mistakes That Developers Make" [Tiešsaiste], pieejams: https://www.toptal.com/c-plus-plus/top-10-common-c-plus-plus-developer-mistakes [skatīts 25.05.2016].

[Vih, 2010] A. Vihrovs, "Datu struktūru grafiskā attēlošana C++ programmām", Kursa darbs, LU DF, 2010.

[Āri, 2011] A. Āriņš, "Dinamiska programmu atmiņas objektu analīze", Bakalaura darbs, LU DF, 2011.

[Aug, 2012] A. Augulis, "Java programmas atmiņas objektu vizualizācija", Bakalaura darbs, LU DF, 2012.

[Myers, 1980] B. Myers, "Displaying Data Structures for InteractiveDebugging", MIT, 1980.

[DDD] "DataDisplayDebugger" [Tiešsaiste], pieejams: https://www.gnu.org/software/ddd/ [skatīts 25.05.2016.].

[GDBOpt] "Options for Debugging Your Program or GCC" [Tiešsaiste], pieejams; https://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/Debugging-Options.html#Debugging-Options [skatīts 23.05.2016.]

[Oua, 1995] S. Oualline, *Practical C++ Programming*, O'Reilly & Associates, Inc, 1995.

[Valgrind] "Valgrind" [Tiešsaiste], pieejams http://valgrind.org/ [skatīts 29.05.2016.]

[Cor, 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT, 2009.

[Dot] "Drawing graphs with dot" [Tiešsaiste], pieejams: http://www.graphviz.org/pdf/dotguide.pdf [skatīts 23.05.2016.].

[Neato] "Drawing graphs with NEATO" [Tiešsaiste], pieejams: http://www.graphviz.org/pdf/neatoguide.pdf [skatīts 23.05.2016.].

[MSDN] "Creating a Child Process with Redirected Input and Output" [Tiešsaiste], pieejams: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682499(v=vs.85).aspx [skatīts 23.05.2016.].

[Fork] "C++, fork(), IPCs using pipe. Multiple Forks, Multiple pipes, Example" [Tiešsaiste] pieejams: http://www.sfu.ca/~reilande/ [skatīts 23.05.2016.].

[Friedl] "Mapping UNIX pipe descriptors to stdin and stdout in C" [Tiešsaiste], pieejams: http://unixwiz.net/techtips/remap-pipe-fds.html [skatīts 23.05.2016.].

[GDBSec] "Security restriction for auto-loading" [Tiešsaiste], pieejams: https://sourceware.org/gdb/onlinedocs/gdb/Auto_002dloading-safe-path.html [skatīts 27.05.2016.].

[Qt] Qt Documentation [Tiešsaiste], pieejams: http://doc.qt.io/qt-5/ [skatīts 25.05.2016.]

[Brasko] "gdbwire source code" [Tiešsaiste], pieejams: https://github.com/brasko/gdbwire [skatīts 20.05.2016.].

[Laz, 2010] S. D. Lazaro, "How-to: Use Graphviz to Draw Graphs in a Qt Graphics Scene" [Tiešsaiste], pieejams: http://www.mupuf.org/blog/2010/07/08/how_to_use_graphviz_to_draw_graphs_in_a_qt_graphics_scene/ [skatīts 20.05.2016.]

PIELIKUMI

1. pielikums

Pirmkods C++ programmai Windows vidē saziņai ar komandrindas atkļūdotāju

```
#include <windows.h>
#include <iostream>

using namespace std;

#define BUFSIZE 4096

HANDLE g_hChildStd_IN_Rd = NULL;
HANDLE g_hChildStd_IN_Wr = NULL;
HANDLE g_hChildStd_OUT_Rd = NULL;
HANDLE g_hChildStd_OUT_Wr = NULL;

HANDLE g_hInputFile = NULL;

void CreateChildProcess(PROCESS_INFORMATION piProcInfo);
void WriteToPipe(char *);
string ReadFromPipe();

int main()
{
    SECURITY_ATTRIBUTES saAttr;
    // Set up members of the PROCESS_INFORMATION structure.
    PROCESS_INFORMATION piProcInfo;
    ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION) );

    // Set the bInheritHandle flag so pipe handles are inherited.
    saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
    saAttr.bInheritHandle = TRUE;
    saAttr.lpSecurityDescriptor = NULL;

    // Create a pipe for the child process's STDOUT.
    if (!CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0))
        cout << "ERROR in StdoutRd CreatePipe" << endl;
    else cout << "STDOUT Pipe created.\n";

    // Ensure the read handle to the pipe for STDOUT is not inherited.
    if (!SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0))
        cout << "ERROR in Stdout SetHandleInformation" << endl;

    // Create a pipe for the child process's STDIN.
    if (!CreatePipe(&g_hChildStd_IN_Rd, &g_hChildStd_IN_Wr, &saAttr, 0))
        cout << "ERROR in Stdin CreatePipe" << endl;
    else cout << "STDIN Pipe created.\n";

    // Ensure the write handle to the pipe for STDIN is not inherited.
    if (!SetHandleInformation(g_hChildStd_IN_Wr, HANDLE_FLAG_INHERIT, 0))
        cout << "ERROR in Stdin SetHandleInformation" << endl;

    // Create the child process.
    CreateChildProcess(piProcInfo);

    cout << "Reading from pipe" << endl;
    cout << ReadFromPipe() << endl;
    cout << "Reading done!" << endl;
    cout << endl;

    cout << "Writing and reading pipe" << endl;
    WriteToPipe("-break-insert main\n");
    cout << ReadFromPipe() << endl;
    WriteToPipe("-exec-run\n");
    cout << ReadFromPipe() << endl;
    cout << ReadFromPipe() << endl;
    WriteToPipe("q\n");
```

```
    cout << "Done!" << endl;

    // Close the pipe handle so the child process stops reading.
    if ( ! CloseHandle(g_hChildStd_IN_Wr) )
        cout << "ERROR in StdInWr CloseHandle" << endl;


    cout << "End of parent execution.\n";

    // The remaining open handles are cleaned up when this process terminates.
    // To avoid resource leaks in a larger application, close handles explicitly.
    CloseHandle(piProcInfo.hProcess);
    CloseHandle(piProcInfo.hThread);
    return 0;
}

void CreateChildProcess(PROCESS_INFORMATION piProcInfo)
// Create a child process that uses the previously created pipes for STDIN and
STDOUT.
{
    TCHAR szCmdline[] = TEXT("gdb MD1test.exe --interpreter=mi2");

    STARTUPINFO siStartInfo;
    BOOL bSuccess = FALSE;

    // Set up members of the STARTUPINFO structure.
    // This structure specifies the STDIN and STDOUT handles for redirection.
    ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
    siStartInfo.cb = sizeof(STARTUPINFO);
    siStartInfo.hStdError = g_hChildStd_OUT_Wr;
    siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
    siStartInfo.hStdInput = g_hChildStd_IN_Rd;
    siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

    // Create the child process.
    bSuccess = CreateProcess(NULL,
        szCmdline,     // command line
        NULL,          // process security attributes
        NULL,          // primary thread security attributes
        TRUE,          // handles are inherited
        0,
        NULL,          // use parent's environment
        NULL,          // use parent's current directory
        &siStartInfo,  // STARTUPINFO pointer
        &piProcInfo);  // receives PROCESS_INFORMATION

    // If an error occurs, exit the application.
    if ( ! bSuccess )
        cout << "ERROR in CreateProcess" << endl;
    else
    {
        // Close handles to the child process and its primary thread.
        // Some applications might keep these handles to monitor the status
        // of the child process, for example.
        CloseHandle(piProcInfo.hProcess);
        CloseHandle(piProcInfo.hThread);
    }
}

void WriteToPipe(char * message)
{
    DWORD dwRead, dwWritten;

          WriteFile(g_hChildStd_IN_Wr,  message,  sizeof(char)  *  strlen(message),
&dwWritten, NULL);
    cout << "Message sent :" << message << endl;
}
```

```
string ReadFromPipe()
// Read output from the child process's pipe for STDOUT
// and write to the parent process's pipe for STDOUT.
// Stop when there is no more data.
{
    DWORD dwRead, dwWritten;
    CHAR chBuf[BUFSIZE] = "";
    BOOL bSuccess = FALSE;
    string ret = "";
    string gdb = "(gdb)";
    char checkGdb[6] = "";

    while (gdb.compare(0, 5, checkGdb, 0, 5))
    {
        bSuccess = ReadFile( g_hChildStd_OUT_Rd, chBuf, BUFSIZE, &dwRead, NULL);
        chBuf[dwRead] = '\0';
        memcpy(checkGdb, &chBuf[dwRead-8],5);
        ret += chBuf;
    }
    return ret;
}
```

2. pielikums

Pirmkods C++ programmai Linux vidē saziņai ar komandrindas atkļūdotāju

```
#include<iostream>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include <stdio.h>
#include <string.h>
using namespace std;

void gdb_agent (int* child_to_parent, int* parent_to_child);
char * readAnswer(int);

int main()
{
     //create 2 pipes to talk from children to parent and parent to children
     int child_to_parent[2];
     int parent_to_child[2];
     pipe(child_to_parent);
     pipe(parent_to_child);

     //pidID. used for joining threads at end of program
     pid_t shut_down;

     //for keeping track if we are in the parent of child for each fork
     int pid;

     //create the child
     pid = fork();

     //error catching.  a pid < 0 is returned when there is a failure
        if (pid < 0)
     {
          cout << "Child failed";
          return -1;
        }

        //if the pid is 0, we are in the child process.
        if (pid == 0)
     {
             cout << "CHILD CREATED " <<  getpid() << endl;
          //start a gdb agent function
          gdb_agent(child_to_parent, parent_to_child);
     }

        //if the pid is greater than 0, the parent is running.
     else
     {
          //remember pid to wait for it to die
                 shut_down = pid;
        }

     if (pid>0)
     { //parent and control function
          //when a process reads from a pipe, if there is nothing in it to read, it
will wait for something
          //to be written into it before continuing.  When something is written into
a pipe, that
          //data can only be read once and only by one process.
          //write to the child command and then wait for a response from the child
          cout << readAnswer(child_to_parent[0]) << endl;

          cout << "Writing break" << endl;
          write(parent_to_child[1], "-break-insert main\n", sizeof(char)*19);
          cout << readAnswer(child_to_parent[0]) << endl;

          cout << "Writing run" << endl;
```

```
            write(parent_to_child[1], "-exec-run\n", sizeof(char)*19);
            cout << readAnswer(child_to_parent[0]) << endl;
            cout << readAnswer(child_to_parent[0]) << endl;
            cout << readAnswer(child_to_parent[0]) << endl;

            cout << "Writing quit" << endl;
            write(parent_to_child[1], "q\n", sizeof(char)*19);

            cout << "Waiting for PID: " << shut_down << " to finish" << endl;

            waitpid(shut_down, NULL, 0);

            cout << "PID: " << shut_down << " has shut down" << endl;
        }

        return 0;
};

void gdb_agent (int* child_to_parent, int* parent_to_child)
{
        dup2(parent_to_child[0], STDIN_FILENO);
        dup2(child_to_parent[1], STDOUT_FILENO);
        execlp("gdb", "gdb", "testapp", "--interpreter=mi2", NULL);
};

char * readAnswer(int p)
{
        char * answer = new char[1000];
        int i = 0;
        char c;
        read(p, &c, sizeof(c));
        while (i < 5 || answer[i-1] != ')' || answer[i-2] != 'b' || answer[i-3] != 'd'
|| answer[i-4] != 'g' || answer[i-5] != '(')
        {
            read(p, &c, sizeof(c));
            answer[i] = c;
            i++;
        }
        answer[i] = '\0';
        return answer;
}
```

3. pielikums
Klases *GDBMIWriter* metožu atšifrējums

Metode	**Atbilstošā atkļūdotāja komanda**	**Skaidrojums**
writeBreak(QString place)	-break-insert [place]	Ievieto pārtraukumpunktu norādītajā vietā. Vieta var būt: funkcija (piemēram, “main”) vai pirmkoda datne un prorgammas rinda (piemēram, “testapp.cpp:20”).
writeRun()	-exec-run	Uzsāk atkļūdojamās programmas izpildi.
writeContinue()	-exec-continue	Atsāk atkļūdojamās programmas izpildi.
writeStepin()	-exec-step	Atsāk atkļūdojamās programmas izpildi, apstājoties pirms nākamās izejas koda rindas. Ja nākamā izejas koda rinda ir funkcijas izsaukums, apstājas pirms funkcijas pirmās rindas.
writeStepover()	-exec-next	Atsāk atkļūdojamās programmas izpildi, apstājoties pirms nākamās izejas koda rindas.
writeStepout()	-exec-finish	Atsāk atkļūdojamās programmas izpildi, apstājoties, kad pabeigta pašreizējās funkcijas izpilde.
writeListLocals()	-stack-list-locals 0	Atgriež visu lokālo mainīgo sarakstu. Šīs funkcijas atbildes apstrādes laikā atkļūdotājam tiek padotas citas komandas, kas nodrošina visu redzamo atmiņas objektu informācijas ieguvi.
writeVarCreate(int token, QString name)	[token]-var-create [varObjectName] * [name]	Izveido mainīgā objektu ar vārdu [varObjectName] un izteiksmi [name]. Izveidotā objekta atribūti tiek saņemtie atbildē ar marķieri [token]. Atribūts [varObjectName] netiek padots metodes izsaukumā, jo klase GDBMIWriter rūpējas par mainīgo objektu uzskaiti un iznīcināšanu.
writeVarListChildren(int token, QString varObjectName)	[token]-var-list-children 0 [varObjectName]	Atgriež mainīgā objekta ar vārdu [varObjectName] bērna objektu sarakstu.
writeVarDeleteAll()	-var-delete [varObjectName]	Izdzēš visus izveidotos mainīgo objektus. Metodei nav atribūtu, jo klase GDBMIWriter rūpējas par mainīgo objektu uzskaiti un iznīcināšanu
writeQuit()	q	Aizver GDB atkļudotāju.

```
#include <vdvariablelist.h>
#include <vdvariable.h>
#include <gdbmiparser.h>
#include <QProcess>
#include <gdbmiparser.h>
#include <QMessageBox>
#include <QString>
#include <QList>
#include <QThread>
#include <QFile>
#include <QTextStream>
#include <vdwindow.h>
#include <gdbmiwriter.h>
VDVariableList::VDVariableList(GDBMIParser * pa, GDBMIWriter * wr, VDWindow * w)
{
    parser = pa;
    writer = wr;
    localsParsed = false;
    window = w;
}
VDVariableList::~VDVariableList()
{
}
void VDVariableList::createVariable(char * n)
{
    if (!nameMap.contains(n))
    {
        addVarObject(n, NULL, NULL);
    }
}
void VDVariableList::createChildVariable(char * n, char * t)
{
    int token = 0;
    if (t != NULL) token = atoi(t);
    if (token == 0) return;
    VDVariable * parent = tokenMap.value(token);
    if (parent->childrenToken != token) return;
      if (strcmp(n, "private") == 0 || strcmp(n, "public") == 0 || strcmp(n,
"protected") == 0) //not a member
    {
        getVariableMembers(parent, n);
        return;
    }
    QString fullNameForMap;
    if (parent->isNamed)
    {
        fullNameForMap = QString("%1.%2").arg(parent->name).arg(n);
    }
    else
    {
        fullNameForMap = QString("[%1.%2]").arg(parent->name).arg(n);
    }
    if (!nameMap.contains(fullNameForMap))
    {
        addVarObject(n, parent, NULL);
    }
}
void VDVariableList::addVarObject(char * shortname, VDVariable * parent, VDVariable
* pointer)
{
    int varToken = parser->getToken();
    int addressToken = parser->getToken();
    QString nameString;
    if (parent == NULL)
    {
```

```
        nameString = shortname;
    }
    else
    {
        if (parent->type.endsWith("]"))
        {
            nameString = QString("%1[%2]").arg(parent->name).arg(shortname);
        }
        else
        {
            nameString = QString("%1.%2").arg(parent->name).arg(shortname);
        }
    }
        VDVariable * var = new VDVariable(nameString, varToken, addressToken,
shortname);
    tokenMap.insert(varToken, var);
    tokenMap.insert(addressToken, var);
    QString prefix;
    if (parent != NULL)
    {
        var->parentPrefix = parent->name;
        var->parent = parent;
        parent->members->append(var);
        var->isNamed = parent->isNamed;
    }
    if (pointer != NULL)
    {
        var->isNamed = false;
        var->isPointer = true;
        prefix = "*";
    }
    else
    {
        prefix = "";
    }
    nameMap.insert(var->fullName(), var);
    writer->writeVarCreate(varToken, QString("%1%2").arg(prefix).arg(var->name));
         writer->writeVarCreate(addressToken,  QString("&%1%2").arg(prefix).arg(var-
>name));
}
bool VDVariableList::updateVariableAtributes(char * t, char * atribute, char *
value)
{
    bool isAtribute = false;
    int token = 0;
    token = atoi(t);
    if (token == 0) return false;
    VDVariable * var;
    if (tokenMap.contains(token))
    {
        var = tokenMap.value(token);
    }
    else
    {
        return false;
    }
    if (var->varObjectToken == token)
    {
        isAtribute = true;
        if (strcmp(atribute, "name") == 0)
        {
            var->setVarObject(value);
        }
        else if (strcmp(atribute, "numchild") == 0)
        {
            var->setNumChildren(value);
        }
        else if (strcmp(atribute, "type") == 0)
```

```
        {
            var->setType(value);
        }
        else if (strcmp(atribute, "value") == 0)
        {
            var->setValue(value);
        }
    }
    else if (var->addressToken == token)
    {
        isAtribute = true;
        if (strcmp(atribute, "value") == 0)
        {
            if(strcmp(value,"") == 0 || strcmp(value, "0x0") == 0)
            {
                removeVariable(var);
            }
            else
            {
                var->setAddress(value);
            }
        }
    }
    return isAtribute;
}
void VDVariableList::getVariableMembers(VDVariable * var, char * keyword)
{
    int token = parser->getToken();
    var->childrenToken = token;
    tokenMap.insert(token, var);
    QString string;
    if (keyword == NULL)
    {
        string = var->varobject;
    }
    else
    {
        string = QString("%1.%2").arg(var->varobject).arg(keyword);
    }
    writer->writeVarListChildren(token, string);
}
void VDVariableList::getVariablePoint(VDVariable * var)
{
    addVarObject(var->name.toUtf8().data(), NULL, var);
}
void VDVariableList::resolveExistingAddress(VDVariable * var)
{
    VDVariable * existing = addressMap.value(var->address);
    if (existing == var) return;
    if (existing->name == var->parentPrefix) return;
    if (var->isNamed && !existing->isNamed) //not named = not needed
    {
        nameMap.remove(existing->fullName());
        existing->name = var->name;
        existing->isNamed = true;
    }

    removeVariable(var);
}
void VDVariableList::resolve(VDVariable * var)
{
    if (!var->address.isEmpty())
    {
        if (!addressMap.isEmpty() && addressMap.contains(var->address))
        {
           resolveExistingAddress(var);
        }
        else
```

```
        {
            addressMap.insert(var->address, var);
            if(var->type.endsWith("*"))
            {
                if(!var->address.isEmpty())
                {
                    getVariablePoint(var);
                }
            }
            else
            {
                if(var->numChildren > 0)
                {
                    getVariableMembers(var, NULL);
                }
            }
        }
    }
}
void VDVariableList::listValues()
{
    if(!localsParsed) return;
    if(!tokenMap.isEmpty()) return;
    makeGraphFile();
}
void VDVariableList::removeToken(char * t)
{
    int token = 0;
    token = atoi(t);
    if (token != 0)
    {
        VDVariable * var = tokenMap.value(token);
        if (var->addressToken == token || var->varObjectToken == token)
        {
            resolve(var);
        }
        tokenMap.remove(token);
        if (tokenMap.isEmpty())
        {
            listValues();
        }
    }
}
void VDVariableList::removeVariable(VDVariable *var)
{
    if (addressMap.contains(var->address))
    {
        VDVariable * existing = addressMap.value(var->address);
        if (existing == var)
        {
            addressMap.remove(var->address);
        }
    }
    if (nameMap.contains(var->fullName()))
    {
        nameMap.remove(var->fullName());
    }
    delete var;
}
void VDVariableList::makeGraphFile()
{
    QFile file("graphviz.gv");
    file.open(QIODevice::WriteOnly);
    QTextStream out(&file);
    out << "digraph {" << endl;
    out << "node [shape=box]";
    QString pointId;
    foreach(VDVariable * var, nameMap)
```

```
    {
        QString id = var->graphvizId();
        if (var->parent == NULL)
        {
            QString label = var->graphvizLabel().replace(' ', "&#92; ");
            if (var->members->isEmpty()) //simple
            {
                out << QString("%1 [label = \"%2\"]\n").arg(id).arg(label);
            }
            else //structure or array
            {
                                    out  <<  QString("%1  [shape=record,  label  =
\"%2\"]\n").arg(id).arg(label);
            }
        }
        if(var->type.endsWith("*") && addressMap.contains(var->value))
        {
            VDVariable * point = addressMap.value(var->value);
            if (point != NULL)
            {
                pointId = point->graphvizId();
                out << QString("%1 -> %2\n").arg(id).arg(pointId);
            }
        }
    }
    out << "}";
    file.close();
    QString gvprogram = "dot -Tsvg -o graphviz.svg graphviz.gv";
    QProcess * gvprocess = new QProcess();
    gvprocess->setProcessChannelMode(QProcess::MergedChannels);
    gvprocess->start(gvprogram);
    gvprocess->waitForFinished();
    gvprocess->close();
    emit window->signalImageParced();
}
```

Bakalaura darbs “Atmiņas objektu attēlošana programmas atkļūdošanas laikā” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autore: Ilze Dzene-Vanaga

Rekomendēju/nerekomendēju darbu aizstāvēšanai (nevajadzīgo izsvītrot).

Vadītājs: Dr. dat. Guntis Arnicāns

Recenzents:

Darbs iesniegts Datorikas fakultātē

Dekāna pilnvarotā persona:

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

Komisijas sekretāre: