Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tadej Golobič

Program za avtomatsko preverjanje algoritmov napisanih v programskem jeziku C++

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Tomaž Dobravec

Ljubljana 2015

To delo je ponujeno pod licenco *Creative Commons Priznanje avtorstva-Deljenje pod enakimi pogoji 2.5 Slovenija* (ali novejšo različico). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani creativecommons.si ali na Inštitutu za intelektualno lastnino, Streliška 1, 1000 Ljubljana.

Izvorna koda diplomskega dela, njeni rezultati in v ta namen razvita programska oprema je ponujena pod licenco GNU General Public License, različica 3 (ali novejša). To pomeni, da se lahko prosto distribuira in/ali predeluje pod njenimi pogoji. Podrobnosti licence so dostopne na spletni strani `http://www.gnu.org/licenses/`.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Program ALGator je namenjen avtomatskem preverjanju pravilnosti in kakovosti delovanja algoritmov, napisanih v programskem jeziku JAVA. ALGator algoritme preverja na vnaprej določenih standardnih testnih primerih. Po zgledu programa ALGator izdelajte program, ki bo omogočal avtomatsko preverjanje algoritmov, napisanih v programskem jeziku C++. Program naj se v podatkovnem delu povsem ujema s programom ALGator, da bomo lahko rezultate obeh programov preverjali z istimi orodji. Program napišite kot zaključen projekt, ki poleg dobre dokumentacije vsebuje tudi vse potrebno za preprosto namestitev in uporabo.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Tadej Golobič sem avtor diplomskega dela z naslovom:

Program za avtomatsko preverjanje algoritmov napisanih v programskem jeziku C++

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 21. avgusta 2015 Podpis avtorja:

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Pregled vsebine 2

2 ALGator **3**

2.1 Vloge v sistemu 3

2.2 ALGatorC 4

3 C++ **11**

3.1 Prevajalnik 12

3.2 Razhroščevalnik 13

3.3 C++0x oziroma C++11 13

4 Knjižnice **19**

4.1 Statične knjižnice 19

4.2 Dinamične knjižnice 22

5 Autoconf in Automake **27**

5.1 Primer uporabe 28

5.2 Navodila za uporabo 32

6 Primer uporabe programa ALGatorC **39**

6.1 Testni primeri 39

6.2 Izvorna koda projekta 41

6.3 Implementacija algoritma BubbleSort 47

6.4 Rezultati 48

7 Sklepne ugotovitve **51**

Seznam uporabljenih kratic

kratica	angleško	slovensko
OS	operating system	operacijski sistem
STL	standard template library	standardna knjižnica predlog
API	application programming interface	programski vmesnik

Povzetek

Testiranje igra zelo pomemben del pri procesu razvoja programske opreme. Ne glede na to kateri proces razvoja programske opreme izberemo, je vedno treba preveriti pravilnost delovanja algoritma. V svojem diplomskem delu sem želel nadgraditi oziroma razširiti sistem ALGator, da ne bi samo preverjal pravilnosti delovanja algoritmov, napisanih v programskem jeziku JAVA, ampak, da bi tudi preverjal pravilnosti delovanja algoritmov, napisanih v programskem jeziku C++. Glede na to, da naš pristop temelji na že obstoječem sistemu ALGator, bi radi vsem uporabnikom tega sistema omogočili enako izkušnjo na sistemu ALGator, kot na njegovi razširitvi imenovani ALGatorC. Prav tako bi radi preverjali iste parametre tako v programskem jeziku C++ kot tudi v programskem jeziku JAVA, da bi lahko kasneje te rezultate tudi primerjali med seboj.

Ključne besede: C++, prevajalnik, statične knjižnice, dinamične knjižnice, linux, ubuntu, razhroščevalnik, automake, autoconf.

Abstract

Testing plays a very important role in the process of software development. Accuracy of the correctly functioning algorithm must be checked, regardless of the developing software process we use. In my thesis I wanted to upgrade or more precisely – to expand ALGator system, so that it would not only verify the accuracy of algorithms functionality written in JAVA programming language, but to also check the functionality of algorithms written in C++ programming language. Our approach is based on a already existing system – ALGator, but we would like to enable equal experience while working on ALGatorC - expanded version of the programme. We would also like to check the same parameters in C++ and JAVA programming languages, so we could be able to compare given results.

Keywords: C++, compiler, debugger, static libraries, shared libraries, linux, ubuntu, automake, autoconf.

Poglavje 1

Uvod

Vsak programer se je vsaj enkrat v življenju srečal s testiranjem programa. Testiranje je osredotočeno na iskanje napak, cilj pa je najti napake v programu. Testiranje ni demonstracija pravilnega delovanja programa pač pa je demonstracija obstoja napake. Lahko pokaže prisotnost napak, ne more pa dokazati odsotnost napak. Je del preverjanja in potrjevanja. Namen testiranja je pokazati kaj program dela, da ustreza zahtevam (to se pokaže razvijalcu in uporabniku), najti napake preden program damo v uporabo in najti situacije, v katerih je delovanje programa nepravilno, nezaželeno ali ne ustreza zahtevam.

V diplomski nalogi želimo poenostaviti preverjanje pravilnosti delovanja algoritmov napisanih v programskem jeziku C++. Želimo ustvariti program, ki bi lahko preverjal pravilnost katerega koli algoritma. Glede na to da že obstaja sistem ALGator, ki preverja pravilnosti delovanja algoritmov napisanih v programskem jeziku JAVA, ga bomo vzeli za oporno točko in se ravnali po njemu.

To diplomsko delo je narejeno kot zaključen produkt, ki poleg programa `algatorc` vsebuje vse potrebno za namestitev na platformo Linux in dokumentacijo. V tem diplomskem delu natančno opišem postopek izdelave takšnega zaključenega produkta

1.1 Pregled vsebine

Diplomsko nalogo smo razdelili na pet poglavij. Sistem ALGator in njegovo nadgradnjo ALGatorC smo podrobneje opisali v drugem poglavju. Programski jezik C++, prevajalnik, razhroščevalnik in C++11 pa smo podrobneje opisali v tretjem poglavju. Četrto poglavje je sestavljeno iz statičnih in dinamičnih knjižnicah ter iz primerov uporabe teh knjižnic. V petem poglavju smo si ogledali orodja `autoconf` in `automake`, ki se uporabljata za namestitev programa na sistemih UNIX, prav tako pa smo si v petem poglavju ogledali, kako so sestavljena navodila za uporabo in kako jih namestimo na sistem z uporabo orodja `automake`. V šestem poglavju pa smo opisali primer celotnega projekta.

Večina poglavij vsebuje tudi primere uporabe. Vsi primeri uporabe so bili testirani na platformi Linux na OS Ubuntu 12.04 x64. Za prevajanje primerov smo uporabili prevajalnik `g++` verzije 4.8.1. Pri petem poglavju pa smo uporabili orodje `autoconf` verzije 2.69 in `automake` verzije 1.11.3.

Poglavje 2

ALGator

ALGator je sistem za izvajanje algoritmov na podanih testnih podatkih ter analizo rezultatov izvajanja. Omogoča nam dodajanje in upravljanje s poljubnim številom projektov. V okviru enega projekta je definiran problem, testne množice vhodnih podatkov in način reševanja nalog tega problema. Vsak projekt mora vsebovati vsaj en algoritem, ki naloge rešuje na predpisan način. Sistem nam prav tako omogoča analizo izvajanja posameznega algoritma ter primerjavo med algoritmi istega projekta [4].

2.1 Vloge v sistemu

Sistem ALGator ima več uporabniških vlog. Uporabniške vloge se delijo glede na to kaj uporabnik dela oziroma kakšne pravice ima znotraj sistema ALGator. Te uporabniške vloge so naslednje:

- **Sistemski administrator** je tisti uporabnik v sistemu, ki ta sistem postavi in ga tudi vzdržuje. Prav tako ima vse pravice in dostope do vseh virov v sistemu.
- **Administrator projekta** definira projekt znotraj sistema. K sami definiciji projekta spada definiranje problema, testnih množic, vhodnih podatkov, opis formata vhodnih in izhodnih podatkov in definicija vseh

razredov, ki so potrebni za izvajanje algoritmov tega projekta. Sistemski administrator ima dostop do vseh virov projekta.

- **Raziskovalec** je tisti, ki definira en algoritem znotraj določenega projekta.

- **Gost** je tisti uporabnik, ki lahko samo vidi podatke o javnih algoritmih, javnih projektov.

2.2 ALGatorC

ALGatorC je razširitev sistema ALGator, saj omogoča preverjanje pravilnosti delovanja algoritmov, napisanih v programskem jeziku C++. Če končni uporabnik že pozna sistem ALGator, potem ne bo imel problemov s spoznavanjem sistema ALGatorC, saj je le-ta, narejen na podlagi že obstoječega sistema,imenovanega ALGator.

2.2.1 Namestitev programa

Program ALGatorC lahko namestimo na sistem s sledečimi ukazi:

```
# ./configure && make && make install
```

Ukaz `make install` nam datoteke programa ALGatorC, ki imajo končnico `.hpp` namesti v `/usr/local/include/algatorc/`, knjižnico imenovano `libAlgatorc.a`, nam namesti v `/usr/local/lib/algatorc` in navodila za uporabo nam namesti v `/usr/local/share/man/man1/`.

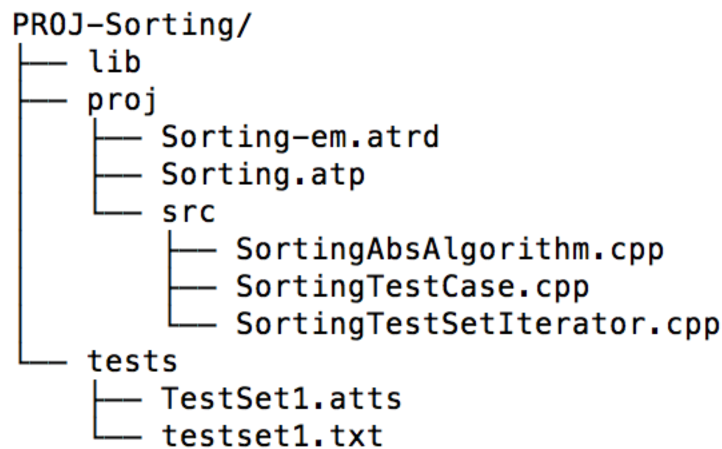
2.2.2 Zgradba ALGatorC projekta

Nov projekt ustvari administrator projekta. Lahko ga ustvari ročno, lahko pa prepusti ustvarjanje novega ALGatorC projekta kar programu samemu. Če prepusti ustvarjanje novega projekta samemu programu, potem bo končal s takšno drevesno strukturo, kot je prikazana na sliki 2.1. Program ALGatorC lahko ustvari takšno drevesno strukturo z ukazom

```
# algatorc -n <ime-projekta>
```

Ko se ustvari nov projekt, se ustvarijo trije direktoriji znotraj projekta, in sicer direktorij lib, direktorij proj in direktorij tests.

Direktorij lib je namenjen shranjevanju knjižnic. K kreiramo nov projekt še nimamo nobene knjižnice, takrat je ta direktorij prazen. Knjižnica se ustvari ob prvem preverjanju algoritma.

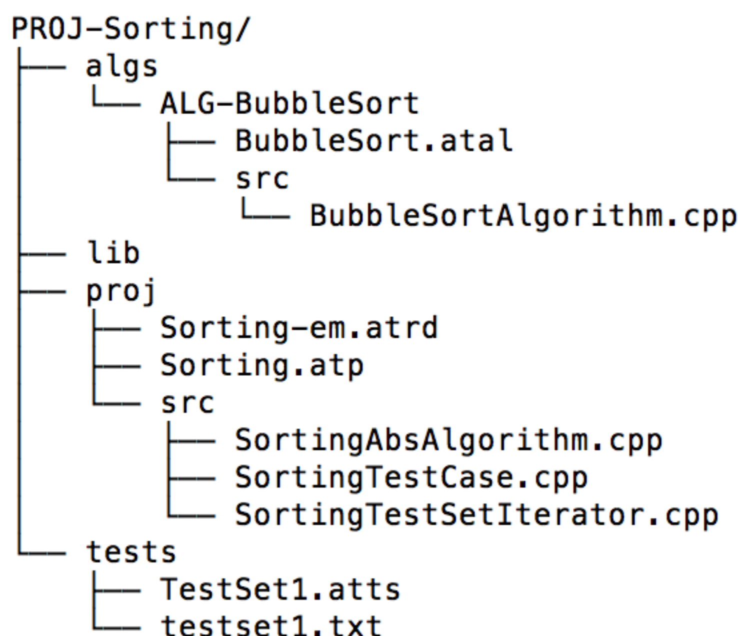


Slika 2.1: Primer drevesne strukture ki jo samodejno ustvari ALGatorC za projekt Sorting

Znotraj direktorija proj se nahajata dve datoteki in sicer datoteka z imenom <ime_projekta>-em.atrd in datoteka <ime_projekta>.atp. Datoteka s končnico .atrd vsebuje opis rezultatov, kjer izvemo, v kakšnem formatu morajo biti rezultati, katere parametre morajo rezultati vsebovati, opis posameznih parametrov, kakšno je ločilo med posameznimi podatki in podobno. Datoteka s kočnico .atp pa vsebuje opis projekta, datum nastanka projekta, avtorja projekta, katere testne množice bo projekt uporabljal, katere algoritme bo uporabljal, podpis metode execute in podobno. Znotraj direktorija proj/src pa se nahajajo tri datoteke. V teh datotekah so trije glavni razredi, ki dedujejo razrede sistema ALGatorC, in sicer dedujejo razrede AbsAlgorithm, TestCase in pa razred TestSetIterator. Razred <ime_projekta>TestCase, predstavlja testni primer za trenutni projekt. Razred <ime_projekta>TestSetIterator, je izpeljan iz ra-

zreda TestSetIterator, je vbistvu iterator nad testno množico. Njegova najbolj pomembna metoda je get_current(), ki vrne testni primer iz testne množice. Razred <ime_projekta>AbsAlgorithm pa ima tri oziroma štiri metode, in sicer init(), ki podan testni primer pretvori v testni primer projekta, metodo run(), ki požene algoritem in metodo done(), ki preveri ali je algoritem nad podanim testnim primerom bil uspešen in kot rezultat vrne parametre testnega primera.

Znotraj direktorija test pa se nahajajo datoteke s končnico .atts in datoteke s končnico .txt. Datoteke s končnico .atts vsebuje podatke o eni testni množici. Ti podatki so ime testne množice, opis testne množice, število testov v testni množici, podatek kolikokrat mora program ponoviti algoritem nad podanim testnim primerom, in podobno. Med vsemi naštetimi podatki je tudi podatek imenovan DescriptionFile. Ta podatek programu pove, kje se nahaja testna množica. Kako je zgrajena testna množica je odvisno od administratorja projekta, saj mora le-ta ustvariti iterator čez testno množico.



Slika 2.2: Primer drevesne strukture ki jo ustvari ALGatorC ob dodajanju novega algoritma v projekt Sorting

Raziskovalec definira algoritem znotraj določenega projekta. Tako kot administrator projekta lahko to ustvari ročno ali pa to prepusti programu. V ta namen ima ALGatorC implementirano dodajanje algoritmov s pomočjo stikala `-a` kateremu sledi ime projekta v katerega želi dodati algoritem in imena algoritmov. Tako mora raziskovalec v terminal napisati sledeči ukaz:

```
# algatorc -a <ime-projekta> <ime-algoritma>
```

Ta ukaz bo dodal nov algoritem znotraj podanega projekta in ustvaril vse potrebne datoteke ter pravilno direktorijsko strukturo direktorija `algs` znotraj podanega projekta, kot je prikazano na sliki 2.2.

Znotraj direktorija `proj/algs` se nahaja en ali več direktorijev. Vsak direktorij vsebuje en algoritem in na podlagi algoritma imajo direktoriji tudi ime. To ime je `ALG-<ime_algoritma>`. Znotraj vsakega takšnega direktorija se nahaja datoteka `<ime_algoritma>.atal`, ki vsebuje podatke o algoritmu. To so podatki kot je ime algoritma, opis algoritma, avtor algoritma, datum in podobno. Znotraj tega direktorija pa se prav tako nahaja direktorij `src`, ki vsebuje eno samo datoteko. V tej datoteki mora raziskovalec implementirati metodo `execute`. Ta metoda mora implementirati ustrezen algoritem.

2.2.3 Zagon programa

ALGatorC lahko zaženemo s pomočjo stikala `-e`, ki mu sledi ime projekta, ime algoritma in ime testne množice. Če bi hoteli izvesti projekt, ki je na sliki 2.2, bi morali pognati naslednji ukaz:

```
# algatorc -e Sorting BubbleSort TestSet1
```

Program bo preveril, ali projekt Sorting, algoritem BubbleSort in testna množica TestSet1, obstajajo. V primeru da kateri ne obstaja, bo program to sporočil in se končal. V primeru da so parametri pravilni, bo program preveril ali obstaja dinamična knjižnica za podani algoritem. V primeru da ne obstaja, jo bo naredil. Če se pa slučajno zgodi, da knjižnica že obstaja, bo program preveril, ali je datum nastanka knjižnice novejši od datuma algoritma in ostalih `cpp` datotek, ki se nahajajo znotraj projekta. Če je katera

izmed datotek novejša, se bo knjižnica na novo ustvarila. Ko je knjižnica ustvarjena, se le-ta naloži v program in iz nje se naložijo simboli oziroma se ustvarijo instance razreda `<ime-projekta>AbsAlgorithm` in razreda `<ime-projekta>TestSetIterator`, ki so pretvorjeni v takšen razred, ki ga ALGatorC pozna. Se pravi so pretvorjeni v razred `AbsAlgorithm` in razred `TestSetIterator`. Nato program pridobi podatke o testnih množicah. Tukaj sta najbolj pomembna podatka kolikokrat se mora en testni primer izvesti in kakšna je vrednost polja `DescriptionFile` znotraj datoteke `TestSet1.atts`. Nato gre program čez vse testne primere in vsak testni primer izvaja tolikokrat, kolikokrat je to zapisano v datoteki `TestSet1.atts`, in za vsakič, ko bo program pognal algoritem nad podanim testnim primerom, bo prav tako meril čas izvajanja algoritma in ta čas shranil. Preden program prebere naslednji testni primer iz podane testne množice, mora rezultate tudi nekam zapisati. Zapiše jih v datoteko `<ime_algoritma>-<ime_testne_mnozice>.em` znotraj direktorija `result`, ki se v ta namen tudi ustvari, če še ne obstaja. Vsaka datoteka z rezultati ima vsaj 4 polja in sicer:

- prvo polje vsebuje **ime algoritma**,
- drugo polje vsebuje **ime testne množice**,
- v tretjem polju se nahaja **id testa**,
- v četrtem polju se nahaja **stanje testa**.

Vsa ostala polja so odvisna od nastavitev projekta. Imena polj lahko dodamo znotraj datoteke `<ime_projekta>-em.atrd`, ki se nahaja v direktoriju `proj` znotraj projekta. Polja se doda v tabelo `TestParameters` ali v tabelo `ResultParameters`. Kot že samo ime pove, se v prvi tabeli nahajajo podatki o samem testu. Tukaj gre bolj za to, da program izpiše kolikokrat je določen test ponovil, koliko je vseh testnih primerov in podobno, medtem, ko se v drugi tabeli nahajajo podatki, ki se navezujejo na en sam testni primer. Znotraj konfiguracijske datoteke projekta, ki opisuje kaj merimo, so pri ključu `ResultParameters` možni naslednji parametri::

- **Tmax**, ki nam da največji čas izvajanja algoritma,
- **Tmin**, ki nam da najmanjši čas izvajanja algoritma,
- **Tavg**, ki nam da povprečni čas izvajanja algoritma,
- **Tsum**, ki nam da skupni čas izvajanja algoritma.

Poleg vseh zgoraj naštetih polj, navadno še administrator projekta doda polje imenovano `Check`, ki preveri in v datoteko z rezultati zapiše, ali so izhodni podatki algoritma pravilni ali napačni.

Poglavje 3

C++

Predhodnik programskega jezika C++ je programski jezik C in je nastal v Bellovih laboratorijih. Prvotno se je jezik uporabljal v sistemih UNIX, predvsem pri izdelavi tega sistema. Čez leta, se je programski jezik C razširil na vse operacijske sisteme. Zelo veliko se uporablja pri pisanju zapletenih programov in pa tudi v krmilni tehniki.

Samo jedro jezika C je v primerjavi z ostalimi jeziki zelo skromno. Jezik C vsebuje samo nekaj definicij spominskih spremenljivk, osnovne matematične in logične operacije, pogojne stavke, skoke in zanke, ter pravila za ustvarjanje funkcij. Zakaj je potem jezik C tako zelo razširjen? Njegova moč je v knjižnicah ki so napisane v jeziku C, zbirniku ali katerem drugem programskem jeziku [1].

Jezik C je strukturirani jezik višjega nivoja. Izvorna koda tega programskega jezika se prevaja v objektno kodo, ta pa se povezuje v izvršljivo kodo.

C++ je objektna verzija jezika C in je splošno namenski programski jezik, ki se je prvič pojavil leta 1983. Programski jezik C++ je razvil Bjarne Stroustrup zaradi potrebe po bolj učinkovitem in elegatnejšem programiranju [15]. Glede na to da je C++ izpeljan iz programskega jezika C, lahko skoraj brez sprememb uporabljamo vse, kar velja v programskem jeziku C. Največji napredek je v predmetno usmerjenem programiranju, kjer se koda in podatki pakirajo v različne razrede. Ti potem dobijo naslednike oziroma

otroke, ki dedujejo podatke od očeta, imajo prijatelje ... Prav tako lahko v programskem jeziku C++ poljubno mešamo predmetno usmerjeno kodo s klasično strukturirano kodo programskega jezika C.

3.1 Prevajalnik

Prevajalnik je poseben program, ki prevede programsko kodo napisano v določenem programskem jeziku in jo spremeni v strojno kodo oziroma v kodo, ki jo računalnikov procesor razume. Najpogostejši razlog za uporabo prevajalnika je ta, da ustvarimo zagonski program.

Navadno uporabnik napiše programsko kodo (recimo v programskem jeziku C++). Nato uporabnik požene prevajalnik nad napisano programsko kodo. Prevajalnik najprej preveri sintakso in semantiko. Prav tako opravi preverjanje podatkovnih tipov in generira napake in opozorila, če le-ta obstajajo. Za tem prevajalnik opravi optimizacijo: odstrani neuporabno oziroma nedosegljivo programsko kodo. Nazadnje ustvari kodo, ki je napisana v zbirnem jeziku.

3.1.1 Primer uporabe prevajalnika g++

Napišimo program `PozdravljenSvet` v programskem jeziku C++ in ga prevedimo z prevajalnikom g++[1]. Narediti moramo novo datoteko, ki jo bomo poimenovali `pozdravljen_svet.cpp` in v njo zapišemo sledečo kodo:

Izsek kode 3.1: Program Pozdravljen svet

```
#include <iostream>
int main(int argc, const char *argv[]) {
   std::cout << "Pozdravljen svet!" << std::endl;
   return 0;
}
```

[1] g++ je prevajalnik za prevajanje programskega jezika C++ in je del zbirke prevajalnikov GNU.

Če želimo sedaj prevesti program 3.1, moramo izvesti naslednji ukaz:

```
# g++ pozdravljen_svet.cpp
```

. Prevajalnik `G++` nam bo prevedel programsko kodo, napisano v programskem jeziku C++, v zagonsko datoteko, ki ima privzeto ime `a.out`.

3.2 Razhroščevalnik

Velikokrat pri programiranju ne gre vse tako kot smo si zamislili in zato program deluje nepričakovano ali pa sploh ne deluje. Za take namene se uporablja razhroščevalnik. Razhroščevalnik nam omogoča, da vidimo, kaj se dogaja znotraj programa med samim izvajanjem in, da vidimo, kaj je program delal v tistem trenutku, ko je iz nenavadnega razloga program nehal z delovanjem [10].

3.3 C++0x oziroma C++11

C++0x je bilo delovno ime za nov standard za programski jezik C++, ki bi dodal številne jezikovne značilnosti. Prvi osnutek tega standarda je bil izdelan že leta 2008 [15]. C++0x je bilo delovno ime zato, ker so ustvarjalci domnevali, da bodo ta standard izdali pred letom 2010, vendar se je izkazalo, da je C++0x postal standard šele septembra leta 2011 in zaradi tega razloga so potem spremenili C++0x v C++11.

C++11 vsebuje široko paleto funkcij: pomembne nove funkcije so podpora za lambda funkcije, večnitnost, poenostavljena for zanka za iteracijo skozi vsebovalnike, ključno besedo auto, izboljšane pametne kazalce ... [15].

3.3.1 Primeri

Izboljšano inicializiranje spremenljivk

Če lahko prevajalnik določi tip spremenljivke iz njene inicalizacije, potem ni potrebno podati tipa spremenljivke. Primer:

Izsek kode 3.2: Enostavna uporaba ukaza auto

```
int x = 3;
auto y = x;
```

Iz tega primera bo prevajalnik sklepal, da je spremenljivka y tipa `int`. To seveda ni najboljši primer uporabe spremenljivke tipa `auto`. Spremenljivke tega tipa najbolj pridejo do izraza, ko imamo opravka s predlogami in z STL-jem [15]. Predstavljajmo si delo z iteratorjem:

Izsek kode 3.3: Dodajanje pošte v slovar

```
std::map<std::string, std::string> poste;
poste[ "Ljubljana" ] = 1000;
//dodamo se par post
```

Sedaj želimo iterirati skozi elemente našega slovarja. To naredimo z uporabo iteratorja:

Izsek kode 3.4: Iteracija skozi slovar z uporabo iteratorja

```
std::map<std::string, std::string>::iterator it = poste.begin();
```

To je zelo dolg tip deklariranja nečesa, za kar že poznamo podatkovni tip. Bilo bi enostavnejše, če bi zapisali sledeče:

Izsek kode 3.5: Iteracija skozi slovar z uporabo iteratorja in ključne besede auto

```
auto it = poste.begin();
```

Ta del kode je krajši in bolj berljiv za razliko od prejšnjega.

Izboljšana for zanka

Ravno primer z iteratorjem je tisti, kjer je standard C++11 prišel do še boljšega načina za ravnanje z njim. To je izboljšana for zanka, ki jo ima sedaj že skoraj vsak programski jezik [15].

Izsek kode 3.6: Primer uporabe izboljšane for zanke

```
std::vector<int> vec;
vec.push_back(10);
vec.push_back(20);

for (int &i : vec )
{
      std::cout << i << std::endl;
}
```

Vse, kar potrebujemo je to, da podamo vsebovalnik (v našem primeru je to vektor celih števil) in ime spremenljivke, v katero se bo posamezen element iz vektorja zapisal.

Kaj pa, če bi želeli iterirati skozi slovar? Kako bi potem napisali vrednost, ki je shranjena v slovarju? Pri vektorju je bilo to enostavno, saj vemo da so vrednosti tipa `int`. Pri slovarju pa je to v bistvu par (`first` in `second`), ki nam da ključ in vrednost. Ampak, če uporabimo ključno besedo auto, potem nam ni treba poznati podatkovnega tipa [14].

Izsek kode 3.7: Primer iteracije skozi slovar z uporabo izboljšane for zanke

```
for ( auto posta_vnos : poste )
{
      std::cout << posta_vnos.first << ": " << posta_vnos.second
         << std::endl;
}
```

Večnitnost

V C++11, lahko nit deli naslov z ostalimi niti. To se dogaja zelo pogosto. Ravno zaradi tega se niti razlikujejo od procesov, saj procesi navadno ne delijo svoj naslov direktno z ostalimi procesi [5].

Poglejmo si sledeči primer:

Izsek kode 3.8: Osnovni primer večnitnosti v C++11

```
#include <iostream>
#include <thread>

void f() {}

struct F {
   F(){}
   void operator()(){}
};

int main(int argc, const char *argv[]) {
   std::thread t1{f};
   std::thread t2{F()};
   return 0;
}
```

Ko prevajamo program, ki uporablja niti kot program 3.8, moramo prevajalniku podati še statično knjižnico imenovano `pthread`. Pri tem programu se `f()` in `F()()` izvedeta na različnih nitih. Problem tega primera je, da ne glede na to, kaj bosta `f()` in `F()()` naredili, ne bomo dobili uporabnega rezultata. Program se bo lahko zaključil preden ali po tem, ko nit `t1` izvede `f()`, in preden ali po tem, ko nit `t2` izvede `F()()`. Da ta problem odpravimo, moramo počakati, da se nit zaključi. To storimo z metodo `.join()`, kar pomeni, da moramo spremeniti našo `main()` funkcijo na naslednji način:

Izsek kode 3.9: Uporaba metode join

```
int main(int argc, const char *argv[]) {
   std::thread t1{f};
   std::thread t2{F()};
   t1.join();
   t2.join();
   return 0;
}
```

Metoda `.join()` nam zagotovi, da se program ne bo končal dokler se niti

ne izvedejo. Z drugimi besedami, metoda `.join()` počaka nit, da se izvede.

Navadno želimo funkciji poslati tudi parametre. Če uporabimo nit, moramo to storiti malce drugače kot običajno. Poglejmo si naslednji primer:

Izsek kode 3.10: Pošiljanje parametrov v funkcijo, ki jo izvede nit

```
void f(std::vector<double> &v) {
   double res = 0;
   for (auto &i : v) {
      res += i;
   }
   std::cout << "Vsota elementov v vektorju je: " << res <<
       std::endl;
}

struct F {
   F(){}
   void operator()(){}
};

int main(int argc, const char *argv[]) {
   std::vector<double> v;
   v.push_back(10.3);
   v.push_back(3.4);
   //napolnimo vektor
   std::thread t1{std::bind(f, v)};
   std::thread t2{F()};

   t1.join();
   t2.join();

   return 0;
}
```

Torej z uporabo funkcije `std::bind()` lahko funkciji pošljemo parametre. Ampak v večini primerov bi želeli tudi, da nam funkcija vrne rezultat. To lahko storimo tako, da funkciji pošljemo naslov spremenljivke, kamor želimo shraniti rezultat, funkcija shrani rezultat na naslov, ki ga je dobila kot para-

meter in ne vrne ničesar.

Izsek kode 3.11: Način, na katerega funkcija, ki je izvedena na posebni niti, "vrne" rezultat

```
#include <iostream>
#include <thread>
#include <vector>

void f(std::vector<double> &v, double *res) {
   *res = 0;
   for (auto &i : v) {
      *res += i;
   }
}

int main(int argc, const char *argv[]) {
   std::vector<double> v;
   v.push_back(10.3);
   v.push_back(3.4);
   double result;
   std::thread t1{std::bind(f, v, &result)};

   t1.join();

   std::cout << "Vsota elementov v vektorju je: " << result <<
      std::endl;

   return 0;
}
```

Poglavje 4

Knjižnice

Eden najbolj pomembnih vidikov v modernem programiranju je zagotovo koncept ponovne uporabe kode. Že sam programski jezik C nam dovoli ponovno uporabo kode v obliki funkcij in struktur. Programski jezik C++ je šel še korak naprej in nam dovoli, da združimo povezane spremenljivke in funkcije, v razrede z enakim namenom. Z uporabo knjižnic pa lahko zadevo odpeljemo na čisto novo raven. Namesto da bi programsko kodo delili samo znotraj enega procesa, lahko kodo delimo znotraj različnih programov.

4.1 Statične knjižnice

Statične knjižnice so starejše knjižnice (od dinamičnih) in niso nič drugega kot zbirka objektnih datotek. Ne glede na to, za kateri operacijski sistem gre, je njihov princip enak. **Vedno morajo obstajati v času prevajanja**, to pa zato, ker se povežejo s programom med prevajanjem programa in so vedno del programa ne glede na to, kje se program nahaja. Končnica datoteke, ki vsebuje statično knjižnico, se razlikuje od operacijskega sistema. Tako imajo statične knjižnice, na distribucijah Linux in na operacijskih sistemih OS X, končnico `.a`, na operacijskih sistemih Windows pa končnico `.lib`. Zaradi prednosti dinamičnih knjižnic statične knjižnice niso več toliko v uporabi, kot včasih.

Statične knjižnice dovolijo uporabniku, da poveže knjižnico s programom, brez potrebe po prevajanju kode, ki se nahaja v sami knjižnici.

4.1.1 Uporaba statične knjižnice

Recimo, da bi radi napisali program, ki bo vseboval razred z imenom `Tocka`. Ta razred je predstavljen z dvema koordinatama (`x` in `y`), namen programa pa je, da lahko točke med seboj seštevamo. Najprej napišemo definicijo razreda in njegovih metod. To naredimo v novi datoteki z imenom `Tocka.hpp`.

Izsek kode 4.1: Definicija razreda Tocka

```
#ifndef Tocka_hpp
#define Tocka_hpp

#include <iostream>

class Tocka{
public:
   int x, y;
   Tocka(int, int);
   std::string print();
   Tocka operator+(const Tocka&);
};

#endif
```

Sedaj moramo vse metode tudi implementirati. To storimo v datoteki z imenom `Tocka.cpp`.

Izsek kode 4.2: Implementacija razreda Tocka

```
#include "Tocka.hpp"

Tocka::Tocka(int a, int b){
   x = a;
   y = b;
}
```

```
std::string Tocka::print(){
   return "[" + std::to_string(x) + ", " + std::to_string(y) +
      "]";
}

Tocka Tocka::operator+(const Tocka &t){
   int a = this->x + t.x;
   int b = this->y + t.y;
   return Tocka(a, b);
}
```

Razred je napisan in na vrsti je ustvarjanje statične knjižnice imenovane `libTocka.a`. Glede na to, da so statične knjižnice samo zbirka objektnih datotek, moramo najprej ustvariti objektno datoteko. Ime objektne datoteke je poljubno, vendar za ta primer bomo dali tej datoteki ime `TockaObj.o`. To storimo z naslednjim ukazom:

```
# g++ -std=gnu++11 -o TockaObj.o -c Tocka.cpp
```

Ta ukaz, samo prevede datoteko `Tocka.cpp` v objektno datoteko imenovano `TockaObj.o`. Za ta specifičen primer smo uporabili `C++11`, saj funkcija `std::to_string` obstaja šele od verzije, ki je bila izdana leta 2011. Sedaj, ko imamo objektno datoteko, lahko iz nje ustvarimo statično knjižnico s pomočjo ukaza `ar`.

```
# ar rv libTocka.a TockaObj.o
```

Ukazu `ar` smo podali tri parametre. Prvi parameter je opcija `rv`. Opcija `r` pomeni, da bi želeli dodati datoteke v statično knjižnico oziroma če ta datoteka v tej knjižnici že obstaja, bi jo radi zamenjali, druga opcija `v` pa pove, da bi želeli informativni izpis na ekran. Drugi parameter označuje ime statične knjižnice, ki bi jo želeli ustvariti oziroma kateri bi želeli dodati objektno datoteko, ki je podana kot tretji parameter. Če želimo, da kdo uporablja naš razred, je dovolj, da ima datoteko z definicijo razreda in njegovih metod ter statično knjižnico. Če kdo želi, lahko sedaj naš razred uporabi na sledeč način:

Izsek kode 4.3: Uporaba razreda Tocka

```
#include <iostream>
#include "Tocka.hpp"

int main(int argc, const char *argv[])
{
   Tocka a = Tocka(1, 2);
   Tocka b = Tocka(3, 4);
   Tocka c = a + b;
   std::cout << c.print() << std::endl;
   return 0;
}
```

Sedaj moramo pri prevajanju samo paziti, da pri povezovanju vključimo tudi pravo knjižnico:

```
# g++ -o tocka main.cpp -L ./ -lTocka
```

Zgornji ukaz nam ustvari program z imenom `tocka` iz izvorne kode, ki se nahaja v datoteki `main.cpp`. Stikalo -L uporabimo zato, da povezovalniku povemo, da se knjižnica nahaja v trenutnem direktoriju (`./`). S stikalom `-l` pa smo podali ime knjižnice. **Stikalo `-l` predvideva, da se ime knjižnice začne z `lib`.**

4.2 Dinamične knjižnice

Dinamične knjižnice so tiste knjižnice, ki se naložijo med samim izvajanjem programa. Predvsem so uporabne za implementacijo modulov ali vtičnikov saj se lahko knjižnica naloži takrat, ko je to potrebno [16].

Glavna razlika med dinamičnimi in statičnimi knjižnicami je ta, da dinamične knjižnice niso naložene med samim povezovanjem programa. Namesto tega imamo API za odpiranje knjižnice, ki zna poiskati simbole, ki zna ravnati z napakami in ki zapre knjižnico.

4.2.1 Uporaba dinamične knjižnice

Recimo, da bi radi napisali dinamično knjižnico, ki vsebuje funkcijo `sestej`, prejme dva parametra in kot rezultat vrne vsoto teh dveh števil. Ustvarimo datoteko `Sestej.cpp`, ki bo vsebovala zgoraj napisano metodo. Datoteka naj bi izgledala tako:

Izsek kode 4.4: Primer funkcije sestej

```
#include <iostream>

int sestej(int a, int b){
   return a + b;
}
```

Ta funkcija naredi točno to, kar smo napisali. Sprejme dva parametra in kot rezultat vrne vsoto teh dveh parametrov. Sedaj pa moramo iz te datoteke ustvariti knjižnico, ki jo bomo poimenovali `libSestej.so`. To storimo z ukazom `g++` na naslednji način:

```
# g++ -fPIC Sestej.cpp -shared -o libSestej.so
```

Vse opcije iz tega ukaza so nam znane, razen opcije `-fPIC`. Tukaj gre za kodo, ki je pozicijsko neodvisna, kar pomeni, da bo delovala ne glede na to, na katerem naslovu se nahaja. Ko imamo ustvarjeno knjižnico, lahko pogledamo, katere simbole vsebuje. To storimo z naslednjim ukazom:

```
# nm libSestej.so
```

Problem je v tem, da nikjer ne vidimo simbola `sestej` brez nenavadnih črk zraven. Najverjetneje bomo videli nekaj v tem smislu:

```
# 00000000000007a8 T _Z6sestejii
```

Dejansko vidimo, da vsebuje simbol `sestej` ampak zraven vsebuje še neke črke. Iz naše funkcije `sestej` moramo narediti C-funkcijo oziroma narediti, da bo vsaj navzven izgledala kot C-funkcija. V C++ lahko to naredimo z ukazom `extern "C"`. Ampak ta ukaz lahko uporabimo samo, kadar uporabljamo programski jezik C++. Zato bomo našo datoteko `Sestej.cpp` spremenili, da bo izgledala tako:

Izsek kode 4.5: Popravek datoteke Sestej.cpp

```
#include <iostream>
#ifdef __cplusplus
extern "C" {
#endif

int sestej(int a, int b){
   return a + b;
}

#ifdef __cplusplus
}
#endif
```

Če sedaj ponovno naredimo knjižnico imenovano `libSestej.so` iz popravljene datoteke in če še enkrat pogledamo simbole, ki se nahajajo v tej knjižnici, bi morali zagledati simbol `sestej`. Sedaj je na vrsti glavni program, v katerega bomo naložili knjižnico in poklicali funkcijo `sestej`. Knjižnico lahko odpremo z ukazom `dlopen`. Če ni bilo težav pri odpiranju knjižnice, moramo naložiti simbol funkcije, ki jo mislimo uporabiti. V našem primeru je to simbol `sestej`. Simbol naložimo z ukazom `dlsym`. Ta ukaz nam vrne rezultat tipa `void*`, to pa zato, ker ne more vedeti, kakšen tip vrača funkcija. Za pravilno pretvorbo moramo poskrbeti sami. Naš program bi izgledal približno tako:

Izsek kode 4.6: Uporaba funkcije, ki se nahaja v dinamični knjižnici

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

int main(int argc, const char *argv[]){
   int (*sestej)(int,int);
   void *handle;
   int a = 5;
   int b = 3;
```

```
    handle = dlopen("./libSestej.so", RTLD_LAZY);
    if (handle)
    {
        sestej = (int(*)(int,int))dlsym(handle, "sestej");
        std::cout << (*sestej)(a, b) << std::endl;
    }
    return 0;
}
```

Kot je napisano zgoraj, odpremo knjižnico z ukazom `dlopen`. Ta ukaz oziroma funkcija prejme dva parametra, in sicer ime knjižnice in pa neko opcijo, iz nabora opcij, ki so že vnaprej določene. V našem primeru smo izbrali opcijo `RTLD_LAZY`. Poglejmo, kaj točno naredi opcija `RTLD_LAZY`. Recimo, da imamo izvršljiv program A in dve dinamični knjižnici. Knjižnica P je recimo primarna in knjižnica S sekundarna. Program A z ukazom `dlopen` naloži knjižnico P in knjižnica P naloži knjižnico S. Recimo, da ima knjižnica P referenco na funkcijo `moja_funkcija()`, ki je definirana v knjižnici S. Če A odpre P brez opcije `RTLD_LAZY`, potem je simbol te funkcije nerazrešen. Če pa knjižnico P naložimo z opcijo `RTLD_LAZY`, potem pa povezovalnik, med samim izvajanjem programa ne poskuša razrešiti simbola za funkcijo `moja_funkcija()`. Tako imamo potem priložnost naložiti knjižnico S preden kličemo funkcijo `moja_funkcija()` [12]. Sedaj, ko je knjižnica naložena, moramo samo preveriti, ali je prišlo do kakšne napake med odpiranjem le-te. Če ni prišlo do nobene napake, potem naložimo simbol s funkcijo `dlsym`, ki mu kot prvemu parametru podamo `handle` in ime simbola. V našem primeru gre tukaj za simbol `sestej`. Kot je bilo že prej napisano, nam funkcija `dlsym` vrne podatkovni tip `void*`. Zato moramo rezultat funkcije `dlsym` pretvoriti v takšen tip, kot je naša funkcija seštej. Na koncu še pokličemo funkcijo `sestej`, ki ji podamo naša dva parametra. Paziti moramo pri prevajanju programa, če ga prevedemo z osnovnim ukazom kot je

```
# g++ -o program_sestej main.cpp
```

bomo najverjetneje dobili sledeče obvestilo o napaki, in sicer, da ima program

nedefinirane reference na funkciji `dlopen` in `dlsym`. Da se tega problema znebimo, moramo uporabiti statično knjižnico imenovano `dl`. Torej pravilni ukaz za ustvarjanje izvršljivega programa iz naše izvorne kode je sledeči

```
# g++ -o program_sestej main.cpp -ldl
```

Če sedaj zaženemo program imenovam `program_sestej`, bomo na ekranu videli vsoto števil 5 in 3.

Poglavje 5

Autoconf in Automake

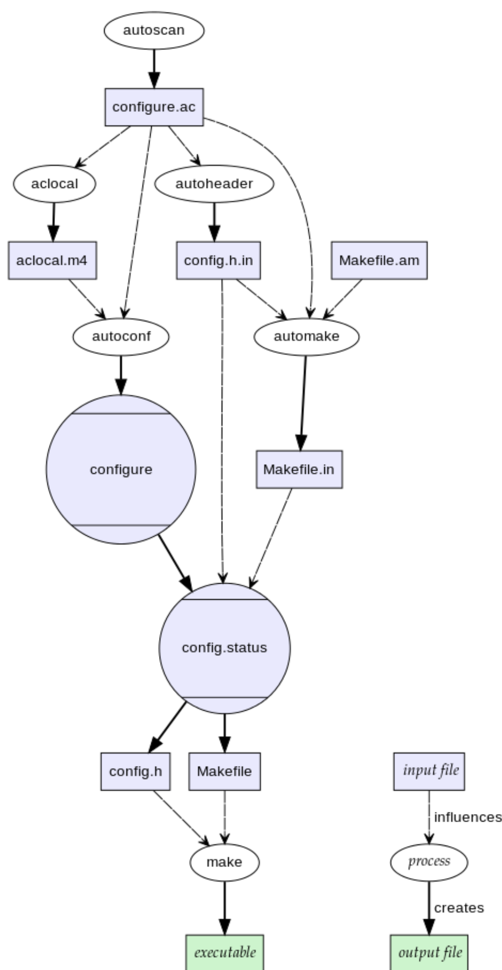
Automake je orodje za samodejno generiranje `Makefile.in` datoteke, ki so skladne z GNU standardi kodiranja. Orodje Automake zahteva uporabo orodja Autoconf [9]. Automake je programsko orodje za avtomatizacijo prevajanja programske kode. To orodje, samodejno ustvari datoteko `Makefile.in` in vhodne datoteke imenovane `Makefile.am`. Vsaka vhodna datoteka `Makefile.in` med drugimi vsebuje tudi zastavice za prevajalnik, povezovalnik in druge. Ustvarjene datoteke `Makefile.in` so prenosljive in v skladu z GNU standardi. Te datoteke se prav tako lahko uporabi pri konfiguracijskih skriptah za generiranje datoteke `Makefile` [2].

Autoconf je orodje za izdelavo konfiguracijske skripte, iz vhodne datoteke `configure.ac` [8], kjer je na voljo Bournova lupina [17]. Autoconf se zelo pogosto uporablja za projekte, ki so napisani v programskih jezikih C, C++, Fortran, Erlang, Objective-C ... Konfiguracijska skripta konfigurira programski paket za namestitev na določenem ciljnem sistemu. Konfiguracijska skripta preverja, ali na ciljnem sistemu obstajajo določeni programi, knjižnice, itd. Ta skripta prav tako ustvari datoteki `Makefile` in `config.status` (Slika 5.1) [2].

Autoconf in Automake se največkrat uporabljata skupaj in sicer za namestitev programa na OS. Tukaj gre predvsem za UNIX platforme. Vsak program, ki je ustvarjen z orodjem Autoconf in Automake se namesti s kom-

binacijo sledečih ukazov

```
# ./configure && make && make install
```



Slika 5.1: Diagram poteka za autoconf in automake. Vir: [https://en.wikipedia.org/wiki/Automake]

5.1 Primer uporabe

Recimo, da bi želeli napisati preprosto igro, kjer bi moral uporabnik ugotavljati, katero število med 1 in 10 si je naš program izbral. Prav tako bi radi,

da je možno naš program namestiti na naš OS.

Najprej napišemo programsko kodo za naš program in kodo shranimo v datoteko `Ugani.cpp`. Programska koda bi izgledala takole:

Izsek kode 5.1: Programska koda za igro Ugani število

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
int main(int argc, const char *argv[]){
   srand(time(NULL));
   int stevilo = rand() % 10 + 1;
   int vnos;
   int cnt = 0;
   do{
      std::cout << "Vnesi stevilo med 1 in 10: > ";
      std::cin >> vnos;
      if (stevilo == vnos){
         std::cout << "Vnesli ste pravilno stevilo" << std::endl;
         break;
      }else{
         std::cout << "Vnesli ste napacno stevilo" << std::endl;
      }
      cnt++;
   }while(cnt < 3);
   return 0;
}
```

Sedaj moramo narediti skripto `autoconf` imenovano `configure.ac`. To storimo z ukazom `autoscan`. Ta ukaz nam bo ustvaril dve datoteki, in sicer `configure.scan` in `autoscan.log`. Datoteko `configure.scan` moramo preimenovati v `configure.ac`. Vse to naredimo z naslednjim ukazom:

```
# autoscan && mv configure.scan configure.ac
```

Ko imamo datoteko `configure.ac`, jo moramo še popraviti. Popravimo sledeče vrstice: Makro `AC_INIT`, moramo spremeniti tako, da namesto po-

lja FULL_PACKAGE_NAME vstavimo ime našega programa, torej ugani, polje VERSION spremenimo v verzijo našega programa, torej 1.0 in spremenimo še polje BUG_REPORT_ADDRESS v naš elektronski naslov. V mojem primeru bo to email@gmail.com. Prav tako moramo zakomentirati vrstico AC_CONFIG_HEADERS([config.h]), saj nismo ustvarili nobene konfiguracijske datoteke [6]. Prav tako moramo preveriti ali obstaja ukaz AC_CONFIG_FILES([Makefile]) v naši datoteki configure.ac. V primeru da ne obstaja, ga moramo dodati. Ta ukaz pove orodju autoconf kaj so izhodne datoteke [7]. V našem primeru je izhodna datoteka Makefile. Prav tako moramo dodati ukaz AM_INIT_AUTOMAKE. Vsebina datoteke bi sedaj morala biti sledeča:

Izsek kode 5.2: Vsebina datoteke configure.ac

```
AC_PREREQ([2.69])
AC_INIT([ugani], [1.0], [email@gmail.com])
AC_CONFIG_SRCDIR([Ugani.cpp])
#AC_CONFIG_HEADERS([config.h])
AM_INIT_AUTOMAKE
# Checks for programs.
AC_PROG_CXX
# Checks for libraries.

# Checks for header files.
AC_CHECK_HEADERS([stdlib.h])
# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.
AC_CONFIG_FILES([Makefile])

AC_OUTPUT
```

Sedaj moramo še ustvariti datoteko Makefile.am. V istem direktoriju kot smo ustvarili datoteko configure.ac, ustvarimo še datoteko Makefile.am, jo odpremo in napišemo sledeče:

Izsek kode 5.3: Vsebina datoteke Makefile.am

```
AUTOMAKE_OPTIONS = foreign

AM_CXXFLAGS=-Wall -std=gnu++11 -DVERSION=\"$(VERSION)\"
   -DPROG="\"$(PACKAGE)\""

bin_PROGRAMS = ugani
ugani_SOURCES = Ugani.cpp
```

Z ukazom v prvi vrstici, povemo orodju `automake`, da to ni standardni paket GNU. Standardni paket GNU zahteva določene direktorije in datoteke, kot so doc, src, INSTALL, COPYING itd.

Z ukazom `AM_CXXFLAGS`, nastavimo zastavice, kot bi jih nastavili, če bi program prevedli z ukazom `g++`. Tukaj dodamo še dve dodatni zastavici, in sicer `-DVERSION` in `-DPROG`. Z ukazom `bin_PROGRAMS = ugani` povemo, da je ime našega programa `ugani`. V naslednji vrstici oziroma v zadnji vrstici, pa povemo, katere so izvorne datoteke našega programa. V našem primeru je to datoteka `Ugani.cpp`. Če bi v datoteki `Makefile.am` uporabili knjižnice ali datoteke v katerem imamo deklarirane razrede, funkcije in vse ostale strukture in spremenljivke, potem bi se te datoteke ob namestitvi prekopirale v `/usr/local/lib` oz. v `/usr/local/include`.

Sedaj, ko smo napisali vsebino datoteke `configure.ac`, in vsebino datoteke `automake` imenovane `Makefile.am`, je potrebno samo še pognati sledeče ukaze:

```
# libtoolize && aclocal && automake -a && autoconf
```

Ukaz `aclocal` je orodje, ki zgenerira datoteko `aclocal.m4` na podlagi datoteke `configure.ac`, ki je uporabljena kot vhodna datoteka za orodje `autoconf`. `autoconf` zgenerira konfiguracijsko skripto, ukaz `automake` pa na podlagi datoteke `Makefile.am` zgenerira datoteko `Makefile.in`

Sedaj lahko program prevedemo in namestimo na naš OS z naslednjimi ukazi:

```
# ./configure && make && sudo make install
```

S temi ukazi smo prevedli in namestili program na naš OS. Sedaj lahko v

ukazno vrstico vpišemo ukaz `ugani` in naš program se bo zagnal.

5.2 Navodila za uporabo

Večina konzolnih programov, ima tudi navodila za uporabo. Na sistemih UNIX lahko do teh navodil pridemo z ukazom `man`, temu pa sledi ime ukaza/programa za katerega bi radi videli navodila za uporabo. Če želimo videti navodila za uporabo ukaza `echo`, vpišemo v terminal sledeče:

```
# man echo
```

5.2.1 Oddelki

Navodila za uporabo so razdeljena po oddelkih [3].

V **oddelku 1** se nahajajo navodila za uporabniške ukaze in orodja kot so orodja za manipulacijo z datotekami, lupine, prevajalniki, itd. Ko se ukaz konča, vrne izhodni status, ki je navadno celo število. V večini lupin lahko vidimo ta status z ukazom `$?`. S tem ukazom lahko vidimo, ali se je prejšnji ukaz/program uspešno zaključil. Če ukaz `$?` vrne kot rezultat 0, potem se je program uspešno zaključil, v nasprotnem primeru je prišlo do napake. Status vsakega programa se lahko giblje med 0 in 255 [3].

V **oddelku 2** se nahajajo sistemski klici. Sistemski klic je vstopna točka v jedro Linuxa. Navadno sistemskih klicev ne kličemo direktno. Večina sistemskih klicev ima ustrezno C-knjižnico, ki poskrbi za določene korake, potrebne pri klicu. V primeru napake večina sistemskih klicev vrne negativno številko. Če se klic uspešno izvede, je vse odvisno od samega klica. Večina sistemskih klicev vrne kot rezultat 0, če se je klic uspešno izvedel, nekateri klici pa lahko vrnejo tudi druge vrednosti (različne od 0) [3].

V **oddelku 3** se nahajajo vse funkcije, ki se nahajajo v knjižnicah, razen tistih, ki so zadolžene za sistemske klice, kot je opisano v prejšnjih dveh odstavkih. [3]

Vse posebne datoteke se nahajajo v **oddelku 4** [3]. Tipično se tukaj nahajajo datoteke, ki so povezane s strojno opremo.

V **oddelku 5** lahko najdemo navodila za uporabo različnih protokolov in datotečnih formatov ter ustreznih struktur, napisanih v programskem jeziku C [3].

Vsa navodila za uporabo iger, ki so trenutno nameščene na našem OS, se nahajajo v **oddelku 6** [3].

V **oddelku 7** lahko najdemo pregled različnih tem, opise protokolov, nabore znakov in razne druge stvari [3]

V **oddelku 8** lahko najdemo ukaze, ki jih lahko ali pa jih lahko samo uporabijo super uporabniki, kot so sistemski administratorji in podobni [3].

5.2.2 Zgradba

Navodila za uporabo imajo tudi določen format. Ko pogledamo navodila za uporabo za kateri koli program, bi običajno videli nekatera zelo pogosta poglavja. Ta poglavja so napisana krepko in z velikimi tiskanimi črkami. Ta poglavja so naslednja:

- NAME
- SYNOPSIS
- DESCRIPTION
- OPTIONS
- BUGS
- AUTHOR
- SEE ALSO

Ta poglavja se pojavijo pri približno 90 % vseh navodil za uporabo [13]. Lahko najdemo tudi druga poglavja, vendar ostala poglavja niso tako pogosta in so dodana zaradi dodatnih informacij o programu. Poglavje **NAME** nam

pove, kakšno je ime programa, sledi kratek opis česa je ta program sposoben oziroma kaj pomeni ime programa, če je ime kratica. Ta del bodo uporabili ukazi kot so `man -k`, `whatis` in podobni. Poglavje **SYNOPSIS** nam pove, kakšna je pravilna sintaksa za zagon programa iz ukazne vrstice. Primer: `ls [OPTION]... [FILE]...` V poglavju **DESCRIPTION** imamo podan opis programa oziroma kakšne so funkcije tega programa. Ker je to tisti del, ki ga večina uporabnikov najprej pogleda, je zelo pomembno, da je to poglavje napisano zelo razumljivo in natančno. Če program uporablja kakšna stikala, potem je to napisano v poglavju **OPTIONS**. Tukaj je opisano kaj vsaka možnost naredi in kakšen parameter sprejme. Če ima program slučajno kakšne znane hrošče v kodi, potem je to zapisano v poglavju **BUGS**. V poglavju **AUTHOR** je zapisano ime avtorja in njegov elektronski naslov. Njegov elektronski naslov je pomemben iz mnogih razlogov. Eden je ta, da lahko uporabniki prijavijo hrošče. Če je program slučajno na nekakšen način povezan s katerim drugim programom, potem je ime programa, s katerim je povezan, zapisan v poglavju **SEE ALSO**. Recimo, da imamo program `foo`, ki je urejevalnik besedila. Potem bi verjetno imeli v tem poglavju referenco na program `vi(1)`.

5.2.3 Namestitev navodila za uporabo z orodjem automake

Spremenimo program 5.1 tako, da bo uporabnik lahko iz ukazne vrstice podal opcijo `-p` tej pa bo sledila številka. Ta številka pomeni kolikokrat lahko uporabnik poizkusi uganiti število oziroma največ kolikokrat se lahko zanka `for` izvede. Program bo izgledal tako:

Izsek kode 5.4: Nadgradnja igre Ugani

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <cstring>
```

```
int main(int argc, const char *argv[]){
   int st_poizkusov = 3;
   if (argc == 3 && strcmp(argv[1], "-p") == 0) {
     st_poizkusov = std::stoi(argv[2]);
   }
   srand(time(NULL));
   int stevilo = rand() % 10 + 1;
   int vnos;
   int cnt = 0;
   do {
      std::cout << "Vnesi stevilo med 1 in 10: > ";
      std::cin >> vnos;
      if (stevilo == vnos) {
      std::cout << "Vnesli ste pravilno stevilo" << std::endl;
         break;
      }else {
         std::cout << "Vnesli ste napacno stevilo" << std::endl;
      }
      cnt++;
   }while(cnt < st_poizkusov);
   return 0;
}
```

Sedaj lahko program namestimo na računalnik z ukazi:

```
# ./configure && make && sudo make install
```

Problem je v tem, da najverjetneje noben uporabnik ne bo vedel, da lahko programu podamo opcijo `-p`. Zato je priporočljivo, da napišemo tudi navodila za uporabo. To naredimo tako, da naredimo novo datoteko, ki jo poimenujemo `ugani.1`. Ta datoteka ima končnico `.1` zato, ker program spada v oddelek 1.

Izsek kode 5.5: Navodilo za uporabo programa Ugani

```
.TH ugani 1 "1. Januar 2015" "verzija 1.0"
.SH NAME
.B ugani
- preprosta igra v kateri mora uporabnik ugotoviti stevilo
```

```
.SH SYNOPSIS
.B ugani
.PP
.B ugani
.I [stikalo stevilo_poizkusov]
.SH DESCRIPTION
.B ugani
je preprosta igra v kateri mora uporabnik ugotoviti stevilo med
   1 do 10, katero je izbral racunalnik
.PP
Uporabnik ima na voljo tri poizkuse, lahko pa tudi vec ce poda
   stikalo -p.
.SH OPTIONS
Mozne so naslednje opcije:
.PP
.B -p
.I stevilo_poizkusov
to stikalo doloci stevilo poizkusov kolikor lahko uporabnik
   ugotavlja stevilo.
.SH EXIT STATUS
.B ugani
vedno vrne 0.
.SH AUTHOR
Ime Priimek, email@gmail.com
```

Definicija makra `.TH` je sledeča: `.TH [ime_programa] [oddelek] [sredina_noge] [levi_del_noge] [sredina_glave]`. Pod ime programa napišemo ime našega programa, pod oddelek napišemo številko oddelka, v katerega spada naš program. Nato določimo komponente, ki jih želimo na dnu strani na sredini, na spodnjem levem robu in na sredini na vrhu strani.

Makro `.SH` sprejme samo en parameter in to je ime poglavja. Imena poglavij si lahko ogledate v poglavju 5.2.2.

Potem imamo makre za krepko pisavo, ležeče in za odstavek. Makro `.PP` se uporablja za odstavek in ne prejme nobenega parametra, medtem ko makra `.B` za krepko in `.I` za ležeče lahko prejmeta poljubno število parametrov.

Sedaj moramo samo še spremeniti našo automake skripto (5.3). Skripti moramo povedati, da imamo za program tudi navodila za uporabo in pa oddelek, kamor bo to navodilo spadalo. Vse ostalo bo automake naredil sam [11].

Izsek kode 5.6: Automake in navodila za uporabo programa

```
AUTOMAKE_OPTIONS = foreign

AM_CXXFLAGS=-Wall -std=gnu++11 -DVERSION=\"$(VERSION)\"
   -DPROG="\"$(PACKAGE)\""

bin_PROGRAMS = ugani
ugani_SOURCES = Ugani.cpp
man1_MANS = ugani.1
```

Če sedaj namestimo program `ugani` na naš OS, lahko v terminal vpišemo

```
# man ugani
```

in dobili bomo navodila za uporabo programa `ugani`.

Poglavje 6

Primer uporabe programa ALGatorC

V tem primeru bomo izdelali projekt, ki ga bomo poimenovali `Sorting`. Znotraj tega projekta bomo implementirali algoritem, imenovan `BubbleSort`, njegova naloga pa je, da naraščajoče uredi tabelo števil.

Najprej moramo najprej ustvariti nov projekt. To bomo prepustili samemu programu. Tako bomo v ukazno vrstico napisali sledeči ukaz:

```
# algatorc -n Sorting
```

Prav tako moramo znotraj ustvarjenega projekta dodati tudi algoritem imenovan `BubbleSort`:

```
# algatorc -a Sorting BubbleSort QuickSort
```

6.1 Testni primeri

Najprej bomo napisali testne primere za naš problem. Pri urejanju števil imam navadno tri skupine testnih primerov. Ena skupina je tista, ki že ima urejena števila po velikost (od najmanjšega do največjega), druga skupina je tista, ki ima nasprotno urejena števila (od največjega do najmanjšega), in zadnja skupina je tista, kjer so števila urejena naključno. Recimo, da

bo testni primer zapisan na sledeči način: `test#:st_stevil:skupina`. Skupine so lahko naslednje:

- **SORTED** - program bo zgeneriral testni primer, pri čemer bodo števila urejena naraščajoče,
- **INVERSED** - program bo zgeneriral testni primer, pri čemer bodo števila urejena padajoče,
- **RND** - program bo zgeneriral testni primer, pri čemer bodo števila urejena naključno,
- **INLINE** - števila so podana za parametrom INLINE,

Recimo da bo naša datoteka `testset1.txt` izgledala takole:

Izsek kode 6.1: Vsebina datoteke testset1.txt

```
test1:10:INLINE:5 4 9 1 34 2 76 1 90 10
test2:10:INLINE:9 1 2 9 3 6 1 3 8 3
test3:10:RND
test4:100:RND
test5:1000:RND
test6:10:SORTED
test7:100:SORTED
test8:1000:SORTED
test9:10:INVERSED
test10:100:INVERSED
test11:1000:INVERSED
```

Sedaj moramo še dopolniti datoteko `TestSet1.atts`. V polje za vrednost `ShortName`, vpišemo kratko ime naše testne množice, pod polje za vrednost `Description` pa lahko napišemo malo daljši opis testne množice. Ostali podatki, ki so pomembni v tem dokumentu so še vrednost za polje `N`, ki nam pove koliko testnih primerov vsebuje testna množica, vrednost za polje `TestRepeat`, ki nam pove kolikokrat izvedemo algoritem na enem testnem primeru in pa vrednost za polje `DescriptionFile`, saj nam ta vrednost pove, kje se nahaja datoteka s testno množico.

Izsek kode 6.2: Primer opisa testne množice

```
{
   "TestSet": {
      "ShortName"    : "Testna mnozica1",
      "Description"  : "Primer mozne testne mnozice",
      "HTMLDescFile" : "",
      "N"            : 11
      "TestRepeat"   : 10,
      "DescriptionFile" : "testset1.txt"
  }
}
```

6.2 Izvorna koda projekta

Znotraj našega projekta imamo prav tako direktorij z imenom `proj`. V tem direktoriju se nahajata dve datoteki in direktorij `src`. Znotraj tega direktorija imamo datoteko s končnico `.atp`, ki opisuje trenutni projekt. Znotraj te datoteke navedemo ime avtorja, datum nastanka projekta, testne množice, algoritme, programski jezik (JAVA ali C++).

Izsek kode 6.3: Vsebina datoteke Sorting.atp

```
{
   "Project" : {
      "Description"     : "Testiranje razlicnih algoritmov za
         urejanje",
      "Author"          : "Tadej",
      "Date"            : "27/08/2015",
      "Algorithms"      : ["QuickSort", "BubbleSort"],
      "TestSets"        : ["TestSet1", "TestSet2"],
      "Language"        : "C++",
      "ExecuteSignature" : "int *tab, int size"
   }
}
```

Prav tako je potrebno v datoteko `Sorting-em.atrd` napisati, v kakšnem formatu naj bodo rezultati, kaj naj rezultati vsebujejo in kaj naj program meri. Dovolj je, da ta datoteka vsebuje vrstice, ki so zapisane v izseku kode 6.4

Izsek kode 6.4: Vsebina datoteke Sorting-em.atrd

```
{
   "ResultDescription":
   {
      "Format"       : "CSV",
      "Delimiter"    : ";",
      "TestParameters" : ["Test", "Group", "N"],
      "ResultParameters" : ["Tmax", "Tmin", "Tavg", "Tsum",
         "Check"]
   }
}
```

Opis posameznih parametrov se nahaja v poglavju 2.2.3. Sedaj je potrebno dopolniti razrede, ki se nahajajo v datotekah `SortingAbsAlgorithm.cpp`, `SortingTestSet.cpp`, `SortingTestSetIterator.cpp`. Vse tri datoteke se nahajajo v poddirektoriju `src`, direktorija `proj`.

Razred `SortingTestCase` mora opisati en testni primer znotraj tega projekta. Ustvariti moramo dve spremenljivki, in sicer ena spremenljivka bo služila kot tabela in bo kazalec, ki kaže na podatkovni tip int in v drugi spremenljivki pa bo zapisana velikost tabele. Ustvarimo še privzeti konstruktor in metodo za inicializacijo tabele.

Izsek kode 6.5: Primer razred SortingTestCase

```
class SortingTestCase : public TestCase {
public:
   int size;
   int *array_to_sort;

   SortingTestCase() {
      size = 0;
```

```
        array_to_sort = nullptr;
    }

    void init_array(int tab[], int s) {
        size = s;
        array_to_sort = new int[size];
        for (int i = 0; i<size; i++)
            array_to_sort[i] = tab[i];
    }
    //ostale potrebne metode
}
```

Prav tako moramo dopolniti razred `SortingTestSetIterator`. V metodi `init_iterator()` moramo samo definirati nov testni primer. Metoda `get_current()` pa mora iz testne množice vrniti en testni primer.

Izsek kode 6.6: Primer razreda SortingTestSetIterator

```
class SortingTestSetIterator : public TestSetIterator {
private:
    SortingTestCase *t_case;
    std::string file_path;
    std::string test_file_name;

    void report_invalid_data_format(const std::string &note) {
        std::string msg = "Invalid input data in file " +
            test_file_name + " in line " +
            std::to_string(line_number);
        if (!note.empty())
            msg += "( " + note + " )";
        LOG(ERROR) << msg;
    }
public:
    bool init_iterator() {
        TestSetIterator::init_iterator();
        t_case = new SortingTestCase();
    }
```

```
SortingTestCase *get_current() {
   if (current_input_line.empty())
      return nullptr;

   std::vector<std::string> fields;
   std::string token;
   std::stringstream str(current_input_line);
   while ( getline(str, token, ':') )
      fields.push_back(token);
   str.clear();

   if (fields.size() < 3) {
      report_invalid_data_format("to few fields");
      return nullptr;
   }

   std::string test_name = fields.at(0);
   int prob_size;
   try {
      prob_size = std::atoi(fields.at(1).c_str());
   } catch (...)
      report_invalid_data_format("'n' is not a number");

   std::string group = fields.at(2);
   std::string test_id = "Test-" +
       std::to_string(line_number);
   EParameter test_id_par = EParameter("TestID", "Test
       identificator", test_id);
   EParameter parameter1 = EParameter("Test", "Test name",
       test_name);
   EParameter parameter2 = EParameter("N", "Number of
       elements", std::to_string(prob_size));
   EParameter parameter3 = EParameter("Group", "A name of a
       group of tests", group);

   t_case->add_parameter(test_id_par);
   t_case->add_parameter(parameter1);
   t_case->add_parameter(parameter2);
```

```
t_case->add_parameter(parameter3);

int arr[prob_size];
int i = 0;
if (group == "INLINE") {
   if (fields.size() < 4) {
      report_invalid_data_format("to few fields");
      return nullptr;
   }
   std::vector<std::string> data;
   std::stringstream ss(fields.at(3));
   while (getline(ss, token, ' ') )
      data.push_back(token);

   if (data.size() != prob_size) {
      report_invalid_data_format("invalid number of inline
         data");
      return nullptr;
   }

   try {
      for (i = 0; i<prob_size; i++)
         arr[i] = std::atoi(data.at(i).c_str());
   } catch (...) {
      report_invalid_data_format("invalid type of inline
         data - data " + std::to_string((i+1)));
      return nullptr;
   }
}
else if (group == "RND") {
   srand(time(NULL));
   for (i = 0; i<prob_size; i++)
      arr[i] = rand() % prob_size + 1000;
}
else if (group == "SORTED") {
   for (i = 0; i<prob_size; i++)
      arr[i] = i;
}
```

```
      else if (group == "INVERSED") {
         for (i = 0; i<prob_size; i++)
            arr[i] = prob_size - i;
      }
      t_case->init_array(arr, prob_size);
      return t_case;
   }
};
```

Sedaj nam preostane samo še to, da dopolnimo razred `SortingAbsAlgorithm`.

Izsek kode 6.7: Primer razreda SortingAbsAlgorithm

```
class SortingAbsAlgorithm : public AbsAlgorithm {
private:
   SortingTestCase *sorting_test_case;
public:
   bool init (TestCase *test) {
      sorting_test_case = dynamic_cast<SortingTestCase*>(test);
      return true;
   }

   void run() {
      execute(sorting_test_case->array_to_sort,
          sorting_test_case->size);
   }

   ParameterSet* done() {
      ParameterSet *result = sorting_test_case->get_parameters();
      EParameter passPar;
      if (check_algorithm())
         passPar = EParameter("Check", "If algorithm is ok",
             "OK");
      else
         passPar = EParameter("Check", "If algorithm is ok",
             "NOK");
      result->add_parameter(passPar, true);
      return result;
   }
```

```
   bool check_algorithm() {
      for (int i = 0; i<sorting_test_case->size-1; i++) {
         if (sorting_test_case->array_to_sort[i] >
             sorting_test_case->array_to_sort[i+1])
            return false;
      }
      return true;
   }

   virtual void execute(int tab[], int size) = 0;
};
```

V metodi `init()` moramo dobljen test, pretvoriti v točno specifičen test. V našem primeru je to `SortingTestCase`. V metodi `run()`, moramo poklicati metodo `execute()`, ki jo bo definiral raziskovalec. Prav tako moramo definirati še metodo `done()`, ki jo program `algatorc` pokliče, kadar algoritem izvrši testni primer. V tej metodi lahko preverjamo, ali je algoritem pravilno uredil testni primer in dodamo še kakšen parameter (v našem primeru smo dodali parameter `Check`, ki nam pove ali, je algoritem uspešno uredil tabelo števil). Prav tako moramo narediti še virtualno metodo `execute()`, saj bi drugače dobili napako v metodi `run()`, ker program ne bi poznal metode `execute()`.

6.3 Implementacija algoritma BubbleSort

Sedaj je na vrsti to, kar mora narediti raziskovalec. Implementirati moramo algoritem `BubbleSort`. Znotraj direktorija `algs/ALG-Sorting/` imamo datoteko, imenovano `BubbleSort.atal`, ki opisuje algoritem. Datoteka je podobna ostalim opisnim datotekam znotraj tega projekta in je dovolj, če napišemo samo to kar piše v izseku kode 6.8.

Izsek kode 6.8: Vsebina datoteke BubbleSort.atal

```
{
   "Algorithm" :
   {
      "ShortName"   : "BubbleSort",
      "Description" : "Osnovni BubbleSort algoritem",
      "Author"      : "Tadej",
      "Date"        : "27/08/2015"
   }
}
```

Sedaj je potrebno samo še definirati metodo `execute()` znotraj razreda `BubbleSortAlgorithm`. Tukaj definiramo algoritem (v našem primeru `BubbleSort`.

Izsek kode 6.9: Definicija razreda BubbleSortAlgorithm

```
class BubbleSortAlgorithm : public SortingAbsAlgorithm {
public:
   void execute(int *tab, int size) {
      for (int i = 0; i < size; i++) {
         for (int j = 0; j < size-1; j++) {
            if (tab[j] > tab[j+1]) {
               int tmp = tab[j];
               tab[j] = tab[j+1];
               tab[j+1] = tmp;
            }
         }
      }
   }
};
```

6.4 Rezultati

Če želimo dobiti rezultate projekta, ki smo ga ustvarili v prejšnjem podpoglavju, `Sorting`, potem moramo izvesti naslednji ukaz:

```
# algatorc -e Sorting BubbleSort TestSet1
```

Ko se program uspešno konča, dobimo nov direktorij, imenovan `results`, znotraj projekta. Ta direktorij vsebuje izhodne datoteke programa oziroma rezultate testiranja programa. Datoteka za ta naš primer se imenuje `BubbleSort-TestSet1.em`. Primer datoteke:

```
BubbleSort;TestSet1;Test-1;DONE;test1;INLINE;10;7.003000;
5.730000;6.083020;304.151000;OK;
BubbleSort;TestSet1;Test-2;DONE;test2;INLINE;10;87.427000;
5.211000;7.348960;367.448000;OK;
BubbleSort;TestSet1;Test-3;DONE;test3;RND;10;16.222000;5.220000;
5.960940;298.047000;OK;
BubbleSort;TestSet1;Test-4;DONE;test4;RND;100;75.490000;
34.949000;37.797540;1889.877000;OK;
BubbleSort;TestSet1;Test-5;DONE;test5;RND;1000;4886.288000;
2866.118000;3400.823580;170041.179000;OK;
BubbleSort;TestSet1;Test-6;DONE;test6;SORTED;10;25.150000;
5.313000;6.763000;338.150000;OK;
BubbleSort;TestSet1;Test-7;DONE;test7;SORTED;100;82.244000;
27.888000;32.331380;1616.569000;OK;
BubbleSort;TestSet1;Test-8;DONE;test8;SORTED;1000;3363.833000;
1731.452000;2562.576500;128128.825000;OK;
BubbleSort;TestSet1;Test-9;DONE;test9;INVERSED;10;48.089000;
5.415000;7.534980;376.749000;OK;
BubbleSort;TestSet1;Test-10;DONE;test10;INVERSED;100;72.871000;
38.824000;42.696240;2134.812000;OK;
BubbleSort;TestSet1;Test-11;DONE;test11;RND;10000;404006.198000;
375647.957000;386525.961840;19326298.092000;OK;
```

Kot vidimo je prvo polje ime algoritma, drugo polje vsebuje ime testne množice, tretje polje je univerzalen identifikator testnega primera, ki je definiran znotraj metode `SortingTestSetIterator::get_current()`, četrto polje pa nam pove, da se je algoritem izvedel brez problema. Vsebino petega, šestega in sedmega polja smo prav tako definirali znotraj metode `SortingTestSetIterator::get_current()`. Peto polje predstavlja ime testa, šesto polje predstavlja skupino (RND, SORTED, INVERSED),

sedmo polje pa predstavlja velikost tabele oziroma testnega primera. Potem sledijo parametri, ki so zapisani v datoteki `Sorting-em.atrd` pri parametru `ResultParameters`. Velikosti testnega primera sledi najdaljši čas izvajanja, nato najkrajši čas izvajanja, nato povprečni čas izvajanja in nazadnje skupni čas izvajanja. Zadnji parameter je `Check`, ki smo ga definirali v metodi `SortingTestSetIterator::done()`, in nam izpiše `OK`, če je algoritem pravilno uredil testni primer oziroma nam v nasprotnem primeru izpiše `NOK`.

Poglavje 7

Sklepne ugotovitve

V okviru diplomske naloge smo ustvarili zelo uporaben program. Ne samo da program preverja pravilnost delovanja algoritma napisanega v programskem jeziku C++, ampak nam prav tako lahko izpiše najdaljši čas izvajanja algoritma, najkrajši čas izvajanja algoritma, povprečni čas izvajanja algoritma, skupni čas izvajanja algoritma in nenazadnje seveda tudi pravilnost delovanja algoritma.

Pri samem programiranju nam je zelo velik problem predstavljalo generiranje dimanične knjižnice iz statične. Prav tako smo imeli kar nekaj problemov pri pridobivanju simbolov iz dinamične knjižnice, saj preden izvedemo program ALGatorC, ne poznamo imena projekta in ostalih zadev, ki so specifične za projekt.

Program ALGatorC je šele nastal in ima še mnogo prostora za razširitev. Želimo si, da bi program nekoč deloval tudi na sistemih OS X in da bi bil program zmožen šteti več zadev, na primer število rekurzivnih klicev, primerjav in podobno.Program bi bilo mogoče razširiti na način, da bi lahko hkrati testirali več algoritmov. Ena od možnih nadgradenj omogoča zapis rezultatov v drugem formatu, na primer v formatu JSON.

Prav tako pa obstaja možnost razširitve sistema ALGator na še kakšen programski jezik, naprimer Python.

Literatura

[1] Goran Bervar. *C++ na kolenih*. Študentska založba, Ljubljana, 2008. [Dostopano: 11. 8. 2015].

[2] John Calcote. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, 2010. [Dostopano: 11. 8. 2015].

[3] Die.net. Linux man pages. `http://linux.die.net/man/`, 2014. [Dostopano: 12. 8. 2015].

[4] Tomaž Dobravec. Algator. `https://github.com/ALGatorDevel/Algator`, 2014. [Dostopano 27. 8. 2015].

[5] Keith G. Erickson. NSTX-U Advances in Real-Time C++11 on Linux. *IEEE Transactions on Nuclear Science*, 62(4):1758 – 1765, 2015.

[6] Inc Free Software Foundation. Configuration header files. `https://www.gnu.org/software/autoconf/manual/autoconf-2.65/html_node/Configuration-Headers.html`, 2009. [Dostopano: 5. 8. 2015].

[7] Inc Free Software Foundation. Creating configuration files. `https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.69/html_node/Configuration-Files.html`, 2009. [Dostopano: 5. 8. 2015].

[8] Inc Free Software Foundation. Autoconf - gnu project - free software foundation (fsf). http://www.gnu.org/software/autoconf/autoconf.html, 2011. [Dostopano: 11. 8. 2015].

[9] Inc Free Software Foundation. Automake - GNU Project - Free Software Foundation (FSF). http://www.gnu.org/software/automake/#TOCintroduction, 2013. [Dostopano: 11. 8. 2015].

[10] Inc Free Software Foundation. Gdb: The gnu project debugger. http://www.gnu.org/software/gdb/, 2015. [Dostopano 15. 8. 2015].

[11] Inc Free Software Foundation. Man pages. https://www.gnu.org/software/automake/manual/html_node/Man-Pages.html, 2015. [Dostopano: 12. 8. 2015].

[12] QNX. RTLD_LAZY. http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fdevel_RTLD_LAZY.html, 2015. [Dostopano: 1. 8. 2015].

[13] Harold Rodriguez. Creating your own man page version 1.0. http://www.linuxhowtos.org/System/creatingman.htm, 2013. [Dostopano: 12. 8. 2015].

[14] Bjarne Stroustrup. *The C++ Programming Language (4th Edition)*. Pearson Education (US), New Jersey, 2013. [Dostopano: 11. 8. 2015].

[15] Bjarne Stroustrup. Bjarne Stroustrup FAQ. http://www.stroustrup.com/C++11FAQ.html, 2014. [Dostopano: 12. 8. 2015].

[16] David A. Wheeler. Dynamically loaded (dl) libraries. `http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html`, 2015. [Dostopano: 1. 8. 2015].

[17] Wikipedia. Bourne shell. `https://en.wikipedia.org/wiki/Bourne_shell`, 2015. [Dostopano: 11. 8. 2015].