LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

C/C++ BIBLIOTĒKU SASKARŅU IZVEIDE PYTHON VALODAI

BAKALAURA DARBS

Autors: **Alberts Glagoļevs**
Studenta apliecības Nr.: ag12082
Darba vadītājs: Asoc. prof., Dr. dat. Jānis Zuters

RĪGA 2017

ANOTĀCIJA

Šajā darbā tiek pētīta C/C++ bibliotēku saskarņu veidošana Python valodai. Saskarnes ļauj izmantot C/C++ bibliotēkas kodu no Python. Manuālā saskarņu rakstīšana nav praktiskā dažādu iemeslu dēļ. Daudzas reālas Python bibliotēkas, kuras izveidotas no C/C++ bibliotēkām, izmanto dažādus rīkus – bibliotēkas un koda ģeneratorus, kuri atvieglo saskarņu veidošanu. Tieši šiem rīkiem ir veltīts darbs. Darba mērķis ir salīdzināt pēc iespējas visus aktuālus rīkus. Ievērojama daļa no darba ir, patstāvīgi izveidoti, piemēri saskarņu veidošanai, izvēlētai reālai C un C++ bibliotēkai. Balstoties uz izveidotiem piemēriem, un rīku dokumentāciju ir veikta rīku analīze un salīdzināšana.

Atslēgvārdi: C/C++ bibliotēku saskarnes Python valodai, Python paplašinājumi, ctypes, cffi, Cython, PyBind11, Boost Python, SWIG, SIP, Shiboken

ABSTRACT

C/C++ LIBRARY INTERFACE CREATION FOR PYTHON

This work studies C/C++ library interface creation for Python language. Interfaces let use C/C++ library code from Python. Manual interface writing is not practical for variety of reasons. Many real Python libraries which created from C/C++ libraries, are using different tools – libraries and code generators, which simplify interface creation. Work is dedicated to this tools. Goal of this work is to compare most of the known tools. Significant amount of work consists of self created examples of interface creation to real C and C++ libraries. Analysis and comparison of tools is done based on created examples and documentation.

Keywords: C/C++ library interface for Python, Python extensions, ctypes, cffi, Cython, PyBind11, Boost Python, SWIG, SIP, Shiboken

SATURS

APZĪMĒJUMU SARAKSTS

API (Application Programmers Interface) – iepriekš definētu klašu, procedūru, funkciju, struktūru un konstanšu kopums, kas tiek pasniegts kā pielikums (bibliotēkas, servisi), kuru iespējams izmantot ārējiem programmatūras produktiem.

ABI (Application Binary Interface) – saskarne starp bibliotēku un lietojumprogrammu binārā līmenī.

CPython – vispopulārākā Python valodas implementācija.

Python/C API – lietojumprogrammas saskarne uz Python, lai piekļūtu interpretatoram dažādos līmeņos.

.so – koplietojuma bibliotēka no Linux platformas.

garbage collector (atkritumu savācējs) – automātiskā programmas atmiņas pārvaldīšana, kad neizmantoti objekti tiek nodzēsti.

segmentācijas kļūda (segmentation fault) – notiek, kad programma mēģina piekļūt aizliegtai atmiņas zonai.

PyObject – Python objekts C līmenī. Jebkurš Python objekts paplašina PyObject tipu.

ietinums (wrapper) – kods apkārt citam kodam (funkcijai, objektam vai bibliotēkai). Piemēram, Python objekts, kurš glabā un izmanto C objektu vai Python funkcija, kura izsauc C funkciju.

bibliotēkas piesaiste (library binding) – speciāli modificēta bibliotēka, lai to varētu izmantot no citas programmēšanas valodas.

iterators – konteinera objekts, caur kuru var iterēt (piekļūt tā elementiem), piemēram, saraksts.

Python pakete – direktorija kur glabājas pirmkoda un citi faili. Satur citas Python paketes vai moduļus.

STL (Standard Template Library) – C++ klašu kolekcija no standarta bibliotēkas.

IEVADS

Viens no iemesliem Python valodas popularitātei ir liels skaits pieejamu bibliotēku. Lielā daļa no nozīmīgām bibliotēkām nāk no C un C++ valodas. Tas ir iespējams pateicoties CPython implementācijai, kurai var izveidot paplašinājumus, un caur tiem izveidot saskarnes C/C++ bibliotēkām. Ir pieejami daudzi rīki, kuri atvieglo saskarņu izveidi, ģenerējot C/C++ saskarnes kodu vai pa tiešo izsaucot C funkcijas no Python koda. Šiem rīkiem ir lielā nozīme, ko apliecina to plaša izmantošana. Piemēram, viena no pazīstamākām grafiskās saskarnes bibliotēkām ir wxPython, kurā nāk no wxWidgets C++ bibliotēkas, kurai tika izveidota saskarne ar SWIG rīku. Vai PyQt bibliotēka, kura izveidota ar SIP rīku no Qt ietvara. C/C++ bibliotēku saskarnes var izmantot ne tikai, lai piesaistītu bibliotēku Python, bet arī lai uzrakstītu ātrāku kodu, vai arī lai piekļūtu zema līmeņa C kodam.

Šī darba mērķis ir salīdzināt pēc iespējas visus aktuālus rīkus saskarņu izveidei. Lai labāk noskaidrot katra rīka stipras un vājās puses, ir izmantota reālā C un C++ bibliotēka un tai tiek izveidota saskarne ar katru no apskatītiem rīkiem. Pētījumā rezultātā tiks noskaidrots, kādas iespējas ir katram rīkam, cik vienkārši tas ir lietojams un kādās situācijās ir labāk izmantot vienus vai citus rīkus.

Darbs sākas ar ievadu par Python paplašinājumiem. 2. nodaļa apraksta piemēros izmantotu C un C++ bibliotēku, programmatūras versijas un ātruma noteikšanas metodi. Tālāk sekos nodaļas ar dažādiem rīkiem, kur katrā ir iekļauts apraksts, lietošanas un iespēju analīze, un piemērs. Beidzot pēdējā nodaļa ir visu rīku salīdzinājums un kopsavilkums.

1 PYTHON PAPLAŠINĀJUMI

CPython implementācijas viena no īpatnībām ir iespēja veidot paplašinājumus. Tos raksta C valodā, izmantojot, Python-C-API, kas ir iekļauts galvenē Python.h. Python paplašinājumus izmanto, lai izsauktu C bibliotēku funkcijas, lai realizētu jaunās funkcijas C līmenī vai arī, lai izveidotu jaunus objekta tipus. Paplašinājuma kodam jābūt speciāli rakstītam, lai to varētu izmantot Python kodā. Nākamajās apakšnodaļās ir aprakstītas C funkciju un datu struktūru saskarnes izveide caur Python paplašinājumiem, to būvēšanu un kopsavilkums.

1.1 C/C++ funkciju piesaiste

Izsaucamai C funkcijai no Python koda, jābūt ar Python tipa argumentiem. Šādai C funkcijai vienmēr ir divi argumenti ar tipu PyObject, pēc konvencijas nosaucami self un args. self arguments norāda uz moduļa objektu. Metodei tas norādītu uz objekta instanci. args arguments norāda uz Python korteža (tuple) objektu, kurš satur funkcijas argumentus, kādi tie bija Python kodā. Lai šos objektus izmantot C kodā – tie jākonvertē C datu tipos. Python iebūvētiem tipiem tas notiek automātiski, izmantojot, int PyArg_ParseTuple(PyObject *arg, char *format, ...); funkciju [1]. Šī funkcija konvertē PyObject tipus uz atbilstošiem C datu tipiem un saglabā norādītājos lokālajos C mainīgos, kuri nāk "..." vietā. *format* simbolu virkne apraksta datu tipus, piemēram, 's' ir Python *string* vai *unicode* simbolu virkne un tiek konvertēta uz *char tipu. Piemērs:

```
/* Iespējamais Python izsaukums: f('example string!') */
static PyObject * f(PyObject *self, PyObject *args)
{
    char *c_str;

    ok = PyArg_ParseTuple(args, "s", &c_str);      /* A string */
    ...
}
```

Izsaucamai no Python paplašinājuma funkcijai atgriežamam tipam jābūt Python objektam, tāpēc C funkcijā vajag uzbūvēt Python objektu. Iebūvētiem datu tipiem ir pieejamas funkcijas, kuras automātiski uzbūvē objektus.

Paplašinājuma modulī jābūt definētai speciālai struktūrai PyMethodDef, kur ir pārskaitītas visas paplašinājuma moduļa funkcijas, to nosaukumi C kodā un Python kodā. Modulī arī jābūt definētam moduļa inicializācijas blokam, kurā tiek uzrakstīts moduļa nosaukums [2].

Kā redzams, lai izveidotu paplašinājuma funkcijas, pietiek nodrošināt, ka funkcijas argumenti un atgriežamais tips ir Python objekti. Acīmredzami paplašinājuma funkcijas ir C funkcijas un no tām var izsaukt citas C funkcijas. Tādā veidā ir iespējams izveidot saskarni C bibliotēkas funkcijām.

1.2 C/C++ datu struktūru piesaiste

C/C++ datu struktūras nevar izmantot pa tiešo Python kodā. Tie speciāli jāpārveido par Python objektu tipiem, jo Python valodā visi datu tipi ir objekti. Python valodai ir iespēja izveidot paplašinājuma tipus. Tie ir jaunie Python tipi, uzrakstīti C līmenī, izmantojot Python/C API. Lai izveidotu saskarni uz kādu C/C++ datu struktūru, var uzrakstīt Python paplašinājuma tipu, kas atbilst C/C++ datu struktūrai. Jāņem vērā, ka šos paplašinājuma tipus vajadzēs konvertēt uz C/C++ atbilstošiem tipiem, lai tos varētu sūtīt C/C++ funkcijām, kurām tiek veidota saskarne. Un arī konvertēt otrādi, uz Python tipu, kad C/C++ bibliotēkas atdod vērtības, kuras jāizmanto Python kodā.

Jauna objekta tipa izveidošanai ir vairāki sīkumi, kurus ir jāievēro. Katram Python objektam tiek glabāts references skaits. Kad tas ir vienāds ar 0, objekts tiek automātiski nodzēsts. Rakstot paplašinājumus, ir jāpārvalda objekta references skaits, palielināt un samazināt to, kad norāde uz objektu tiek kopēta vai nodzēsta [3]. Tas attiecas uz visiem PyObject* objektiem, piemēram, objekta atribūtiem vai metožu argumentiem.

Ja paplašinājuma tipam ir vajadzīga kontrole objekta izveidošanas laikā, tad jāimplementē objekta izveidošanas funkciju - tp_new. Tā atbilst __new__ statiskai Python metodei un atgriež izveidotu objektu. Šī funkcija izpildās objekta izveidošanas laikā un tiek izmantota, lai inicializētu objekta atribūtus. Ir vēl cita līdzīga funkcija – tp_init. Tā arī tiek izmantota inicializācijai, bet neatgriež jaunu objektu. Tā atbilst __init__ Python metodei. Ir arī funkcija objekta iznīcināšanai - tp_dealloc. Paplašinājuma tipa metodes tiek izveidotas līdzīgi, kā moduļa funkcijas (aprakstīts 1.1 nodaļā). Speciālā masīvā tiek norādītas visas paplašinājuma tipa metodes, to nosaukumi un norādes uz funkcijām. Paplašinājuma tipa atribūtu vērtības mainīšanai caur = operatoru, arī ir jāraksta funkcijas. Un pat lai nolasītu vērtību, ir jāraksta get funkciju [3].

Viens veids, kā atvieglot paplašinājumu rakstīšanu, ir paplašināt python iebūvētus tipus, piem., sarakstu, tādējādi mantojot funkcionalitāti. Ja papildus funkcionalitāte nav vajadzīga, tad var arī nerakstīt paplašinājuma tipu, un izmantot Python iebūvētos tipus, piem., sarakstu. Tad to var manuāli konvertēt uz C masīvu vai citu datu tipu.

1.3 Objektu pārvaldīšana

Viena no lietām, par kuru ir jārūpējas, rakstot saskarnes, ir, kā C/C++ objekti ir pārvaldīti. Tas ir, kura puse – Python vai C/C++ bibliotēka ir atbildīga par objektu nodzēšanu, jeb atmiņas atbrīvošanu. Kad Python objekta references skaits kļūst 0, atkritumu savācējs automātiski nodzēš objektu. Ja C/C++ kods izmantos šī objekta datus, tad notiks segmentācijas kļūda. Tā var notikt arī, ja Python programma mēģinās piekļūt C/C++ datiem, kuri C/C++ kodā jau ir nodzēsti.

Viens no veidiem, kā izvairīties no šīm problēmām, ir kopēt sūtamus datus no C/C++ uz Python. Tad dati no abām pusēm paliek neatkarīgi. Tas ir drošs veids, bet ātruma ziņā neefektīvs, un turklāt var radīt atmiņas noplūdes, ja C/C++ neatbrīvos atmiņu. Efektīvāks veids, kā pārvaldīt objektus, ir glabāt C/C++ objektus Python objektā kā referenci. To sauc par ietinuma objektu. Tad ir jānodrošina, ka C/C++ references dati tiks atbrīvoti, kad Python objekta references skaits kļūst 0. Vai, ja C/C++ puse ir atbildīga par atbrīvošanu, tad nenodzēst. Tas nozīmē, ka vajag noskaidrot, vai dati tiek atbrīvoti C/C++ kodā.

1.4 C/C++ paplašinājumu būvēšana

Paplašinājuma būvēšanas mērķis ir dinamiskā bibliotēka (piem., .so paplašinājums Linux sistēmā, .pyd Windows sistēmā). Paplašinājuma moduļus var uzbūvēt ar distutils rīku, kas ir iekļauts Python instalācijā. distutils izmanto skriptu setup.py, kurā tiks uzrakstīti paplašinājuma moduļu nosaukumi, izejas failu nosaukumi, un cita nepieciešama kompilācijai informācija. Skripta palaišana nokompilē C izejas failus un izveido paketi kopā ar paplašinājuma moduļiem [4].

1.5 Kopsavilkums

Manuālā Python paplašinājuma rakstīšana ir visgrūtākais veids, kā izveidot saskarni C/C++ bibliotēkai, bet tajā paša laikā ar vislielāko kontroli. Šī kontrole var dot vislielāko ātrdarbību. Ja kaut kas nestrādā pareizi, tad programmētājs zina, kā kļūda ir noteikti viņa uzrakstīta kodā, nevis koda ģeneratora kļūda. Pie trūkumiem var atnest liela koda daudzums saskarnes izveidei. Arī būtisks trūkums ir Python versiju atbalsts. Python 2 un 3 versijai ir atšķirības, un lai atbalstītu abas, būs jāraksta papildus kods.

Manuāla Python/C API priekšrocības:

- daudz zemā līmeņa kontroles

- nav bibliotēkas vai rīka atkarības

Manuāla Python/C API trūkumi:

- jāraksta daudz C/C++ koda
- papildus pūles Python versiju atbalstām
- viegli ieviest kļūdas
- ir jāzina Python/C API

2 TESTĒŠANAS VIDE

Darbā tiek izmantotas reālas C un C++ bibliotēka ar atvērto pirmkodu, lai uz piemēra parādīt, kā notiek saskarnes izveidošana ar dažādiem rīkiem, un arī, lai notestētu saskarņu ātrdarbību un ģenerēta koda daudzumu. Bibliotēkas ir brīvi pieejamas GitHub vietnē. Gan C, gan C++ bibliotēka ir JSON parsētāji. Gan C, gan C++ piemēriem tiks izmantots viens un tas pats JSON teksts. Šajā nodaļā ir arī aprakstīta ātruma noteikšanas metode, datora konfigurācija un izmantotas programmatūras versijas.

2.1 jsmn C bibliotēka

Bibliotēkai ir tikai viens C pirmkoda un viens galvenes fails. Bibliotēkas darbība ir bāzēta uz marķieriem (tokens). Katram marķiera objektam ir tips:

```
typedef enum {
    JSMN_UNDEFINED = 0,
    JSMN_OBJECT = 1,
    JSMN_ARRAY = 2,
    JSMN_STRING = 3,
    JSMN_PRIMITIVE = 4
} jsmntype_t;
```

Marķiera objektam ir arī starta un beigas pozīcijas, kuras norāda attiecīgi uz JSON simbolu virknes pozīcijām. Tā izskatās token datu struktūra:

```
typedef struct {
    jsmntype_t type; // Token type
    int start;       // Token start position
    int end;         // Token end position
    int size;        // Number of child (nested) tokens
} jsmntok_t;
```

Bibliotēkas API ir tikai divas funkcijas:

```
void jsmn_init(jsmn_parser *parser);

int jsmn_parse(jsmn_parser *parser, const char *js, size_t len,

            jsmntok_t *tokens, unsigned int num_tokens);
```

`jsmn_init` funkcija inicializē parseri. `jsmn_parse` funkcija parsē JSON simbolu virkni js ar garumu len. Pēc funkcijas izpildes tokens objekti tiek aizpildīti ar datiem.

Bibliotēka tiek nokompilēta ar sekojošu komandu:

```
gcc -Wall -Wextra -O -fpic -shared jsmn.c -o jsmn.so
```

Katram rīkam bibliotēka tiks izmantota vienā un tā pašā veidā, lai maksimāli tuvu novērtētu ātrdarbību. Tiks izveidota funkcija get_tokens(json_string), kura paņem JSON simbolu virkni

un atgriež tokens objektus. Šie objekti tiks izmantoti, lai nolasītu “nm” atribūta vērtības no JSON teksta.

JSON teksts sastāv no masīva ar vairākiem objektiem. Objekta piemērs:

```
{
   "nm": "Peter, Apostle",
   "yrs": "33-67",
   "snt": "Saint"
}
```

JSON teksts pieejams vietnē: http://mysafeinfo.com/api/data?list=popes&format=json

2.2 SimpleJSON C++ bibliotēka

Bibliotēkai ir 2 klases – JSON un JSONValue. JSON klase tiek izmantota kā statiskā klase, lai parsēt JSON simbolu virkni ar metodi:

```
static JSONValue* Parse(const wchar_t *data);
```

Parse metode atgriež JSONValue klases objektu, kurš reprezentē kādu no JSON tipa vērtībām.

Piemēros tiks izmantots masīva tips:

```
typedef std::vector<JSONValue*> JSONArray;
```

kuru iegūst ar metodi AsArray.

Kā arī tiks izmantots objekta tips:

```
typedef std::map<std::wstring, JSONValue*> JSONObject;
```

kuru iegūst ar metodi AsObject. Šīs objekts ir vārdnīca, kurai atslēga ir JSON objekta atribūta nosaukums.

Piemēros ir izmantots JSON teksts no C bibliotēkas piemēriem, un mērķis ir tāds pats kā C bibliotēkai, proti, savākt visas atribūta “nm” vērtības.

Bibliotēka tiek nokompilēta ar sekojošu komandu:

g++ -Wall -Wextra -O -fpic -shared JSON.cpp JSONValue.cpp -o simplejson.so

2.3 Datora konfigurācija

Linux: 4.4.62-18.6

Distributīvs: OpenSUSE 42.2

Procesors: i5 3360m

RAM: 8GB

2.4 Ātruma noteikšanas metode

Programmu izpildes ātrums tiek pārbaudīts, izmantojot *timeit* Python moduli no standarta bibliotēkas. Šeit ir piemērs, kā tas tiek izmantots:

```
import timeit

setup = """
from ctypes_example import get_tokens

f = open("json_sample.txt", "r")
json_string = f.read()
f.close()
"""

program = """
tokens = get_tokens(json_string)

names = []
for i in range(len(tokens)):    # for every token
    if tokens[i].type == 3:     # if token type is string
        # if token string is "nm" then append to list next token
        if json_string[tokens[i].start : tokens[i].end] == "nm":
            names.append(json_string[tokens[i+1].start : tokens[i+1].end])
print(names)
"""

print(timeit.timeit(program, setup=setup, number=15000))
```

timeit izsauc *setup* mainīga kodu vienu reizi un *program* mainīga kodu 15 000 reižu. Tādā veidā laiks faila nolasīšanai netiek ņemts vērā.

2.5 Programmatūru versijas

Šeit ir uzskaitītas programmatūras versijas, kuras tiek izmantotas piemēros.

Python: 3.4.5

Cython: 0.25.2

cffi: 1.10

SWIG: 3.0.10

SIP: 4.16.9

boost: 1.63

pybind11: 2.1.1

PyBindGen: 0.17

Shiboken: 1.2.2

3 RĪKI

Šajā nodaļā atrodas dažādu rīku un bibliotēku analīze. Katra rīka sadaļa sākas ar vispārēju rīka aprakstu. Tad seko rīka lietošanas īpatnību un iespēju analīze, kur tiek aprakstīts, kā notiek saskarnes izveidošana C funkcijām, kā tiek konvertēti datu tipi no Python uz C/C++ un otrādi, vai ir atbalsts C/C++ datu struktūrām, masīviem un norādēm. Labākai rīku saprašanai, tiek izveidoti arī piemēri saskarņu izveidošanai C un C++ bibliotēkai.

3.1 ctypes

ctypes ir bibliotēka, kas ir iekļauta Python standarta bibliotēkā un ļauj izsaukt C funkcijas no koplietojamas bibliotēkas (shared library). Funkciju izsaukšana notiek no paša Python koda, un tam nav vajadzīgs speciāli rakstīts C kods, interfeisa izveidošanai, un nav vajadzīga kompilācija. Tas ir priekšrocība salīdzinājumā ar manuālu paplašinājumu izveidošanu un atklāj iespējas izmantot lielu skaitu gatavu bibliotēku [5]. ctypes modulis ir uzrakstīts, izmantojot libffi bibliotēku, ar kuru var izsaukt tikai C funkcijas, tāpēc C++ izmantot nevar. Tā kā ctypes ir standarta bibliotēkas modulis, tas ir implementēts arī citās Python implementācijās - Jython, IronPython un PyPy.

3.1.1 Lietošana

Pirms C funkciju izsaukšanas ir jāielādē vajadzīga C bibliotēka. Bibliotēkai jābūt nokompilētai. Tad var izsaukt funkciju, padodot speciālus ctypes tipus, kuri atbilst C datu tipiem un izsaukšanas laikā tiek konvertēti. Katram C primitīvam datu tipam atbilst kāds speciāls ctypes objekts, kuram var piešķirt un mainīt vērtību, piemēram, šeit tiek izveidots

```
>>> from ctypes import *
>>> i = c_int(77)
>>> print(i.value)
77
>>> i.value = -99
>>> print(i.value)
-99
```

vesela skaitļa ctype tips un mainīta vērtība:

Primitīvus datu tipus ir viegli konvertēt no ctypes objektiem uz Python objektiem, piemēram, int(ctype_object) konvertē no ctypes.c_int tipa uz Python int tipu. To var pielietot, kad vajag saņemt un izmantot C funkcijas vērtības [6].

Ar ctypes ir iespēja izveidot struktūras datu tipu, norādot lauku nosaukumus un tipus, piemēram, šeit ir struktūra ar diviem laukiem:

```
from ctypes import *
```

```
class Point(Structure):
    _fields_ = [("x", c_int), ("y", c_int)]
```

Tad tiem var piešķirt vērtības un izmantot C funkcijā, kā argumentus. Kā struktūras lauku var izmantot arī citu struktūru, tādā veidā uzbūvējot sarežģītākas struktūras. Ar speciālu metodi in_dll() var piekļūt globālajiem mainīgajiem [6].

ctypes objektus var sūtīt C funkcijām arī kā references tipus, vienkārši izmantojot funkciju byref(<ctype objekts>). Lai izveidotu norādi (pointer) uz kādu objektu, pietiek izsaukt funkciju pointer(<ctype objekts>). Vēl viens svarīgais C datu tips ir masīvs, kuru var izveidot, vienkārši pareizinot ctype objektu ar skaitli, kas nozīmē masīva garumu [6]. Šī masīvu var izmantot arī Python kodā, lai piekļūtu tā elementiem vai noskaidrot elementu skaitu.

3.1.2 Lietošanas piemērs

ctypes_example.py:

```
import ctypes

class jsmn_parser(ctypes.Structure):
    _fields_ = [("pos", ctypes.c_uint),
                ("toknext", ctypes.c_uint),
                ("toksuper", ctypes.c_int)]

class jsmntok_t(ctypes.Structure):
    _fields_ = [("type", ctypes.c_uint),
                ("start", ctypes.c_int),
                ("end", ctypes.c_int),
                ("size", ctypes.c_int)]

def get_tokens(json_string):
    jsmn = ctypes.CDLL("./jsmn.so")   # load library
    parser = jsmn_parser()
    jsmn.jsmn_init(ctypes.byref(parser))

    # creates C like array of chars
    c_string = ctypes.create_string_buffer(str.encode(json_string))

    # determine needed tokens count
    tokens_count = jsmn.jsmn_parse(ctypes.byref(parser), c_string,
        ctypes.c_size_t(len(json_string)), None, 0)

    # refresh parser
    jsmn.jsmn_init(ctypes.byref(parser))

    tokens = (jsmntok_t * tokens_count)()   # tokens array
    # parse json string
    jsmn.jsmn_parse(ctypes.byref(parser), c_string,
        ctypes.c_size_t(len(json_string)), tokens, tokens_count)

    return tokens
```

Vislielākās grūtības sagādā C funkciju pareizu argumentu tipu lietošana, jo ir jāzina, kā ctypes objekti tiek konvertēti C datu tipos. Piemēram, šeit sūtot json simbolu virkni vajadzēja ar speciālu ctypes metodi izveidot objektu, kas atbilst C simbolu virknei char*.

usage_example.py:

```
from ctypes_example import get_tokens

f = open("json_sample.txt", "r")
json_string = f.read()
f.close()

tokens = get_tokens(json_string)

names = []
for i in range(len(tokens)):    # for every token
    if tokens[i].type == 3:     # if token type is string
        # if token string is "nm" then append to list next token
        if json_string[tokens[i].start : tokens[i].end] == "nm":
            names.append(json_string[tokens[i+1].start : tokens[i+1].end])
print(names)
```

Kā var redzēt šajā piemērā, ctypes masīva (tokens) elementiem var piekļūt ar indeksa palīdzību. jsmntok_t objekta atribūti arī ir pieejami ar parasto sintaksi. Šajā piemērā var arī redzēt, ka ar ctypes var ne tikai izsaukt C funkcijas, bet arī izmantot C objektus. Masīvs ar `jsmntok_t` tipa objektiem tika nosūtīts C funkcijai, kura aizpildīja to ar datiem un atgrieza Python skriptam.

3.1.3 Kopsavilkums

ctypes bibliotēkas priekšrocības:

- nāk ar Python instalāciju, kas nozīmē, ka lietotājiem nevajag neko papildus instalēt
- labs atbalsts no izstrādātājiem
- nav vajadzīga kompilācija, un saskarnes izveide notiek Python kodā, kas atvieglo izstrādes procesu
- ir pieejama citās Python implementācijās

ctypes bibliotēkas trūkumi:

- nav C++ atbalsta
- datu tipu konvertācijai jāizmanto speciāli bibliotēkas datu tipi un funkcijas

3.2 CFFI

CFFI līdzīgi kā ctypes ir C ārējo funkciju saskarne (C Foreign Function Interface), kas ļauj izmantot C kodu no koplietojamam bibliotēkām. Galvenā atšķirība ir, ka ar CFFI nav jādeklarē izmantotie tipi Python kodā, kā tas ir ar ctypes. To vietā CFFI izmanto C galvenes deklarācijas. Tādā veidā lietotājam gandrīz nav jāmācas jauno API. Vēl var pieminēt, ka CFFI ir pieejams gan ar CPython, gan ar PyPy Python implementāciju [7].

3.2.1 Lietošana

CFFI ir divi lietošanas režīmi – ABI mode un API mode. ABI režīms piekļūst bibliotēkām binārā līmenī, savukārt API režīms piekļūst tiem, izmantojot C kompilatoru. API režīmā tiks izveidots Python paplašinājuma modulis, ar kuru palīdzību var izsaukt C funkcijas. Abos režīmos Python kodā jāuzraksta C datu tipu un funkciju deklarācijas, kuras var vienkārši nokopēt no galvenes failiem, bet API režīmā deklarācijas var izlaist, un tad kompilators tas pabeigs. Pēc bibliotēkas ielādes un deklarācijas uzrakstīšanas var izsaukt C funkcijas un izveidot C datu tipus. Salīdzinot ar ctypes, viss notiek ļoti līdzīgi, izņemot to, ka nevajag deklarēt C datu tipus ar ctypes sintaksi, bet tikai galvenes deklarācijas [7].

Ar CFFI var izveidot un izmantot C primitīvus datu tipus, masīvus un struktūras. Tāpat kā ar ctypes – šos datu tipus var inicializēt Python kodā un piešķirt tam vērtības, un izmantot funkciju izsaukumos.

3.2.2 Uzbūvēšana

API režīmā tiek uzbūvēts paplašinājuma modulis. Lai atvieglotu būvēšanas procesu, CFFI nodrošina integrēšanu ar Python distutils un setuptools rīkiem. Šajā darbā tiek izmantots distutils un būvēšanas skripts – setup.py ir sekojošs:

```
from distutils.core import setup
import cffi_build

setup (
    ext_modules=[cffi_build.ffi.distutils_extension()]
)
```

Šeit cffi_build ir python modulis, kuru var redzēt tālāk, lietošanas piemērā.

Palaišana ar komandu: python3 setup.py build_ext --inplace

Izpildes rezultātā vispirms tiek palaists cffi_build skripts, kurš uzģenerē C pirmkoda failu. Tad tas tiek kompilēts kā Python paplašinājuma modulis un kompilēšanas laikā tiek saistīts ar C bibliotēku, šajā gadījumā ar libjsmn.so.

3.2.3 Lietošanas piemērs

Tika izveidoti 2 piemēri – ABI un API režīmā:

ABI režīms

cffi_build.py:

```
from cffi import FFI

ffi = FFI()
ffi.set_source("cffi_example", None)

ffi.cdef("""
    typedef enum {
        JSMN_UNDEFINED = 0,
        JSMN_OBJECT = 1,
        JSMN_ARRAY = 2,
        JSMN_STRING = 3,
        JSMN_PRIMITIVE = 4
    } jsmntype_t;
    typedef struct {
        jsmntype_t type;
        int start;
        int end;
        int size;
    } jsmntok_t;
    typedef struct {
        unsigned int pos; /* offset in the JSON string */
        unsigned int toknext; /* next token to allocate */
        int toksuper; /* superior token node, e.g parent object or array */
    } jsmn_parser;
    void jsmn_init(jsmn_parser *parser);
    int jsmn_parse(jsmn_parser *parser, const char *js, size_t len,
        jsmntok_t *tokens, unsigned int num_tokens);
""")

if __name__ == "__main__":
    ffi.compile()
```

cffi_build.py satur no C galvenes faila paņemtas deklarācijas struktūrām un funkcijām, kuras tiks izmantotas cffi_usage.py modulī.

cffi_usage.py:

```
from cffi_example import ffi

def get_tokens(json_string):
    lib = ffi.dlopen("./jsmn.so")   # load library
    parser = ffi.new("jsmn_parser*")
    lib.jsmn_init(parser)

    # determine needed tokens count
    tokens_count = lib.jsmn_parse(parser, str.encode(json_string),
        len(json_string), ffi.NULL, 0)
    # refresh parser
    lib.jsmn_init(parser)
```

```
    tokens = ffi.new("jsmntok_t[]", tokens_count)   # tokens array
    # parse json string
    lib.jsmn_parse(parser, str.encode(json_string),
        len(json_string), tokens, tokens_count)

    return tokens
```

cffi_usage.py modulī tika izsauktas C funkcijas līdzīgi kā tas bija ar ctypes.

usage_example.py:

```
from cffi_usage import get_tokens

f = open("json_sample.txt", "r")
json_string = f.read()
f.close()

tokens = get_tokens(json_string)

names = []
for i in range(len(tokens)):    # for every token
    if tokens[i].type == 3:     # if token type is string
        # if token string is "nm" then append to list next token
        if json_string[tokens[i].start : tokens[i].end] == "nm":
            names.append(json_string[tokens[i+1].start : tokens[i+1].end])
print(names)
```

tokens objekta izmantošana ir identiskā ar ctypes piemēru.

API režīms

cffi_build.py:

```
from cffi import FFI

ffi = FFI()

ffi.set_source("cffi_example",
    """ // passed to the real C compiler
        #include "jsmn.h"
    """,
    include_dirs=["./"],
    libraries=["./jsmn"])

ffi.cdef("""
    typedef enum {
        JSMN_UNDEFINED = 0,
        ...
    } jsmntype_t;
    typedef struct {
        jsmntype_t type;
        int start;
        int end;
        int size;
    } jsmntok_t;
    typedef struct {
        unsigned int pos; /* offset in the JSON string */
```

```
        ...;
    } jsmn_parser;
    void jsmn_init(jsmn_parser *parser);
    int jsmn_parse(jsmn_parser *parser, const char *js, size_t len,
        jsmntok_t *tokens, unsigned int num_tokens);
""")

if __name__ == "__main__":
    ffi.compile()
```

API režīmā ir divas atšķirības ar ABI piemēru. ffi.set_source() metode sūta C kompilatoram vajadzīgu informāciju kompilēšanai – galvenes faila un bibliotēkas nosaukumu un atrašanas vietu. Otrā atšķirība ir ka deklarācijas jsmntype_t enum un jsmn_parser struct netika uzrakstītas līdz galam. Šīs deklarācijas pabeigs kompilators. Lauku deklarācijas var izlaist tikai, ja tie netiek izmantoti pa tiešo Python kodā, tāpēc jsmn_parser struktūrai nevar izlaist lauku deklarācijas.

cffi_usage.py un usage_example.py tādi paši kā ABI režīmā.

3.2.4 Kopsavilkums

cffi rīka priekšrocības:

- visa saskarnes izveidošana notiek Python kodā
- atšķirībā no ctypes - gandrīz nav jāmācas jaunu API
- atbalsta CPython un PyPy implementācijas
- divi lietošanas režīmi – ABI un API

cffi rīka trūkumi:

- nav C++ atbalsta

3.3 Cython

Cython ir programmēšanas valoda, kas ļauj rakstīt Python valodas paplašinājumus, izmantojot Python līdzīgu valodu (Python virskopa). Cython ir translators no Python koda uz C kodu. Ar to var deklarēt statiskus tipus funkciju parametriem, lokālajiem mainīgajiem un klašu atribūtiem, kas tiek izmantots, lai nodrošinātu saskarni ar C/C++ funkcijām un bibliotēkām. Pēc būtības Cython translē Python kodu uz C kodu, integrējoties ar CPython interpretatoru, kurš ir uzrakstīts C valodā [8]. Bez Cython, lai uzrakstītu paplašinājumu, būtu jāzina un jāizmanto Python/C API.

3.3.1 Lietošana

Lietojot Cython, C bibliotēkas kods kļūst pieejams Python kodā caur paplašinājuma veidošanu. Cython uzģenerē Python paplašinājumu (C kodu) no uzrakstīta Cython koda. C bibliotēka ir pieejama Cython kodā, kur tiek rakstīts piesaistes kods un kuru tad var izmantot Python kodā.

Lai piekļūtu C bibliotēkas datiem un funkcijām, tos vajag nodeklarēt, līdzīgi kā C galvenēs, bet tikai ar nedaudz atšķirīgu sintaksi. Parasti šīs deklarācijas tiek novietotas failā .pxd, no kurienes dažādi Cython moduļi var importēt nodeklarētas C funkcijas [9].

C datu tipu deklarēšana Cython kodā notiek ar *cdef* atslēgvārdu. Var izmantot jebkuru mainīgo tipu no C valodas. Ir pieejami C masīvi un norādes, “struct”, “union” un “enum” datu tipi. C mainīgie var būt izmantoti, kā funkciju parametri gan C funkcijām, gan Python funkcijām. Ja funkcijas izsaukumā argumenti ir Python objekti un funkcijas parametri ir nodeklarēti, kā C datu tipi, tad Cython izdara tipu pārveidošanu no Python uz C. Tā notiks, ja starp tipiem ir atbilstība, piemēram, no Python float tipa uz C float, double un long double. Jāpiemin, ka C skaitļu tipiem, atšķirībā no Python ir diapazonu ierobežojumi (izņemot int objektu Python 2 versijā, kur iekšēji tas glabājas, kā C long tips), tāpēc, ja tos pārsniegt, būs kļūda. Automātiskā tipu pārveidošana notiek arī, ja sajaukt kādā izteiksmē Python un C mainīgus [9].

3.3.2 C++ atbalsts

Cython atbalsta C++ bibliotēkas saskarni izveidi. Lai to izdarīt, vajag nodeklarēt C++ klases, līdzīgi kā C++ galvenēs. Tad klases var importēt un izmantot Cython kodā. Lai klases varētu izmantot arī Python kodā, tiem vajag izveidot ietinumus – Python klases veidā. Cython atbalsta C++ metožu pārslogošanu (overloading), šablonus (templates), un citas C++ iespējas.

Cython arī ļauj konvertēt dažādus datu tipus no C++ standarta bibliotēkas uz Python

datu tipiem un otrādi, automātiski kopējot konteinera datus. Piemēram, no C++ saraksta vai vektora uz Python sarakstu [10]. Attēlā *Attēls 3-1* redzami C++ standarta bibliotēkas konteineru atbilstības ar Python datu tipiem.

Python type =>	*C++ type*	=> Python type
bytes	std::string	bytes
iterable	std::vector	list
iterable	std::list	list
iterable	std::set	set
iterable (len 2)	std::pair	tuple (len 2)

Attēls 3-1: **Cython automātiskā konteineru konvertācija starp Python un C++.** Attēls no [10]

3.3.3 Cython programmu būvēšana

Cython ģenerē paplašinājuma moduļus, kurus var izmantot Python kods, tāpat kā parastos Python moduļus ar *import* izteiksmi. Cython programmas izveides laikā moduļi ar .pyx paplašinājumu, izmantojot Cython, tiek kompilēti .c failā, kas satur Python paplašinājuma kodu, kuru vajag nokompilēt un tiek izveidots .so (Linux) vai .pyd (Windows) fails [11]. To var izdarīt dažādos veidos. Šajā darbā tas tiek darīts ar *distutils* moduļa palīdzību un setup.py priekš C piemēra:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

setup(
    ext_modules = cythonize([Extension("cython_example",
        ["cython_example.pyx"],
        runtime_library_dirs=['./'],
        libraries=["jsmn"],
        library_dirs = ["./"])])
)
```

Šeit tika norādīts arī bibliotēkas nosaukums un direktorija. Šie parametri tiks izmantoti kompilēšanas laikā, lai savienotājs (linker) atrastu mūsu bibliotēku - libjsmn.so.

Būvēšana ar komandu: `python setup.py build_ext --inplace`

setup.py priekš C++ piemēra:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

setup(
```

```
    ext_modules = cythonize([Extension("cython_example",
["cython_example.pyx"],
        runtime_library_dirs=['./'],
        libraries=["simplejson"],
        library_dirs = ["./"],
        language="c++")])
)
```

3.3.4 Lietošanas piemērs C bibliotēkai

cython_example.pyx:

```
from libc.stdlib cimport malloc, free

cdef extern from "jsmn.h":
    ctypedef struct jsmntok_t:
        unsigned int type   # enumeration in C
        int start
        int end
        int size

    ctypedef struct jsmn_parser:
        unsigned int pos
        unsigned int toknext
        int toksuper

    void jsmn_init(jsmn_parser *parser)
    int jsmn_parse(jsmn_parser *parser, const char *js, size_t len,
        jsmntok_t *tokens, unsigned int num_tokens)

def get_tokens(json_string):
    cdef jsmn_parser parser
    jsmn_init(&parser)
    # determine needed tokens count
    cdef int tokens_used = jsmn_parse(&parser, str.encode(json_string),
        len(json_string), NULL, 0)

    # refresh parser
    jsmn_init(&parser)
    # tokens array
    cdef jsmntok_t *tokens = <jsmntok_t *>malloc(tokens_used *
sizeof(jsmntok_t))
    if not tokens:
        raise MemoryError()
    # parse json string
    jsmn_parse(&parser, str.encode(json_string),
        len(json_string), tokens, tokens_used)

    tokens_list = []
    for i in range(tokens_used):
        tokens_list.append(tokens[i])

    free(tokens)
    return tokens_list
```

Cython kodā C masīvs - tokens tiek pārveidots par Python sarakstu, lai to varētu nosūtīt Python modulim - usage_example.py

usage_example.py:

```
from cython_example import get_tokens

f = open("json_sample.txt", "r")
json_string = f.read()
f.close()

tokens = get_tokens(json_string)

names = []
for i in range(len(tokens)):    # for every token
   if tokens[i]["type"] == 3:    # if token type is string
      # if token string is "nm" then append to list next token
      if json_string[tokens[i]["start"] : tokens[i]["end"]] == "nm":
         names.append(json_string[tokens[i+1]["start"] : tokens[i+1]["end"]])

print(names)
```

Šeit var redzēt, ka Cython automātiski konvertēja C struktūras objektus par Python vārdnīcas objektiem. Protams, nav grūti uzrakstīt Cython kodā savu ietinumu apkārt struktūras objektam, piemēram, Python klases veidā, lai objekta atribūtiem piekļūt caur punktu, nevis caur vārdnīcas indeksu.

3.3.5 Lietošanas piemērs C++ bibliotēkai

cython_example.pyx:

```
from libcpp.vector cimport vector
from libcpp.map cimport map
from libc.stddef cimport wchar_t, size_t
from libcpp cimport bool
from cython.operator cimport dereference, preincrement

cdef extern from "<Python.h>":
    wchar_t* PyUnicode_AsWideCharString(object, Py_ssize_t *)
    object PyUnicode_FromWideChar(const wchar_t *w, Py_ssize_t size)

cdef extern from "string" namespace "std":
    cdef cppclass wstring:
        wchar_t* c_str()

cdef extern from "JSON.h":
    ctypedef vector[JSONValue*] JSONArray
    ctypedef map[wstring, JSONValue*] JSONObject

    cdef cppclass JSON:
        @staticmethod
        JSONValue* Parse(char *data)
        @staticmethod
        JSONValue* Parse(wchar_t *data)
```

```
cdef extern from "JSONValue.h":
    cdef cppclass JSONValue:
        JSONValue()
        JSONValue(wchar_t *m_char_value)
        JSONValue(wstring *m_string_value)

        bool IsString()
        bool IsArray()
        bool IsObject()

        wstring &AsString()
        JSONArray &AsArray()
        JSONObject &AsObject()

cdef class PyJSON:
    @staticmethod
    def Parse(json_string):
        py_v = PyJSONValue(json_string)
        return py_v

cdef class PyJSONValue:
    cdef JSONValue *c_json_value # hold a C++ instance which we're wrapping
    def __cinit__(self, json_string):
        cdef Py_ssize_t length
        cdef wchar_t* c_string
        if json_string != "":
            c_string = PyUnicode_AsWideCharString(json_string, &length)
            self.c_json_value = JSON.Parse(c_string)

    def PyAsArray(self):
        cdef JSONArray json_array = self.c_json_value.AsArray()
        py_list = []
        cdef unsigned int i
        for i in range(json_array.size()):
            py_json_value = PyJSONValue("")
            py_json_value.c_json_value = json_array[i]
            py_list.append(py_json_value)
        return py_list

    def PyAsObject(self):
        cdef JSONObject json_object = self.c_json_value.AsObject()
        py_dict = {}
        cdef map[wstring, JSONValue*].iterator end = json_object.end()
        cdef map[wstring, JSONValue*].iterator it = json_object.begin()
        cdef const wchar_t* c_string
        while it != end:
            py_json_value = PyJSONValue("")
            py_json_value.c_json_value = dereference(it).second
            c_string = dereference(it).first.c_str()
            py_string = PyUnicode_FromWideChar(c_string, -1)
            py_dict[py_string] = py_json_value
            preincrement(it)

        return py_dict

    def PyAsString(self):
        cdef const wchar_t* c_string = self.c_json_value.AsString().c_str()
        return PyUnicode_FromWideChar(c_string, -1)

    def PyIsArray(self):
```

```
        return self.c_json_value.IsArray()

    def PyIsObject(self):
        return self.c_json_value.IsObject()
```

Kā redzams, C++ piemērs ir lielāks un sarežģītāks. Šeit bibliotēka izmanto vairākus C++ datu tipus. Importēti tipi no libcpp un libc ir Cython paketē nedeklarēti tipi. wchar_t tipu Cython nemāk automātiski konvertēt uz Python tipu, tāpēc tiek izmantotas funkcijas no Python.h: PyUnicode_AsWideCharString un PyUnicode_FromWideChar. Tips, kuru bija papildus jādeklarē, ir wstring klase no std:string un metode c_str(), kura atdod wchar_t simbolu virkni.

Tad seko C++ bibliotēkas klašu deklarācijas. PyJSON un PyJSONValue ir Cython klases, kuras var izmantot Python kodā. Tie kalpo par ietinumiem C++ klasēm. Tur notiek visa manuāla tipu pārveidošana. PyJSONValue klase tur JSONValue C++ instanci atribūta veidā. Konstruktorā __cinit__ tas tiek aizpildīts ar datiem. PyAsArray metode pārveido C++ vektoru ar JSONValue objektiem uz Python sarakstu ar PyJSONValue objektiem un atgriež to.

PyAsObject metode pārveido JSONObject objektu par Python vārdnīcu. Lai dabūtu C++ vārdnīcas vērtības, tiek izmantots iterators un speciāla Cython funkcija *dereference*, kura atgriež iteratora vērtību.

usage_example.py:

```
from cython_example import PyJSON, PyJSONValue

f = open("json_sample.txt", "r")
json_string = f.read()
f.close()

names = []
json_value = PyJSON.Parse(json_string)
if json_value.PyIsArray():
    json_array = json_value.PyAsArray()
    for object in json_array:
        if object.PyIsObject():
            o = object.PyAsObject()
            names.append(o["nm"].PyAsString())

print(names)
```

3.3.6 Kopsavilkums

Cython priekšrocības:

- Python līdzīga valoda paplašinājumu rakstīšanai
- auto-ģenerētais kods no Cython koda

- labākā kontrole nekā pilnīgi automatizētiem rīkiem
- labs C un C++ atbalsts
- Cython kompilators paziņo par dažādam kļūdām kodā
- iekļauj GNU atkļūdotāja paplašinājumu, kas ļauj atkļūdot Cython kodu
- ļoti laba dokumentācija

Cython trūkumi:

- jāmācas jaunu valodu
- C++ klasēm ir jāraksta Cython ietinumi, lai tos varētu izmantot Python kodā
- vairākiem C/C++ datu tipiem no standarta bibliotēkas ir jāraksta deklarācijas, lai tos varētu izmantot Cython kodā
- ir tomēr labi jāzina C/C++ valodu, lai pareizi rakstīt Cython kodu, kas bieži satur norādes un references tipus, ar kuriem viegli radīt kļūdas

3.4 SWIG

SWIG (Simplified Wrapper Interface Generator) ir saskarnes ģenerators, kurš savieno programmas, uzrakstītas C/C++ valodās ar dažādam skriptu valodām, tai skaitā Python. Lai uzģenerētu saskarnes kodu, SWIG izmanto C/C++ galvenes failus un speciālu saskarnes failu [12].

Lai uzbūvētu Python paplašinājuma moduļus, SWIG izmanto metodi, kurā daļas no paplašinājuma moduļa definēti C valodā un citas daļas Python valodā. Šīs C kods satur zema līmeņa ietinumus, turpretim Python kods satur augstā līmeņa ietinumus [13]. Tas ļauj vairāk izmantot Python valodas īpatnības, tai skaitā objektorientētu pieeju C struktūrām un C++ klasēm.

3.4.1 Lietošana

Lai uzģenerētu saskarnes kodu, vispirms ir jāuzraksta speciālu saskarnes failu (ar .i paplašinājumu), kurš satur tādu informāciju, kā paplašinājuma moduļa nosaukumu un C/C++ galvenes failu deklarācijas. SWIG vispirms uzģenerē C vai C++ failu, tad to var nokompilēt, piemēram ar distutils rīku, kurš zina kā darboties ar SWIG. Rezultātā tiek izveidots paplašinājuma modulis un Python modulis, kurš to izmanto [13].

C funkcijas SWIG vienkārši pārvērš par Python funkcijām, saglabājot to nosaukumus. C globālajiem mainīgajiem var piekļūt caur speciālu moduļa objektu *cvar,* piem., cvar.GLOBAL_VAR. SWIG atbalsta arī C/C++ norādes. Funkcijai vai metodei ar norādes tipu parametru var padot Python objektu vai vērtību ar atbilstošo tipu. Padodot SWIG C struktūras datu tipu, SWIG uzģenerē Python klasi, ar kuru palīdzību var piekļūt struktūras atribūtiem [13], piemēram:

```
struct Vector {
   double x,y,z;
};
```

Python kodā:

```
v = example.Vector()
v.x = 3.3
v.y = 7.7
```

C līmenī SWIG piekļūst struktūras vai klases atribūtiem ar C funkciju palīdzību, kas nav visātrākais veids.

Priekš Python nav atšķirības starp short un int datu tipiem, bet tas neapgrūtina

lietošanu, jo Python kodā viss notiek kā parasti, t.i., tiek izmantots Python integer tips. Zema līmenī SWIG nodrošina, kā Python tips tiks konvertēts uz atbilstošo C datu tipu.

SWIG atbalsta C masīvu izmantošanu. Ar speciālu direktīvu %array_class var pateikt SWIG, lai tiktu uzģenerēta saskarne uz C masīvu ar noteiktu tipu elementiem. Tad Python kodā iespējams izveidot šo masīvu un sūtīt C funkcijām, kā arī piekļūt masīva elementiem. Ja kādai funkcijai arguments ir norāde, tad Python kodā var vienkārši nosūtīt attiecīgu vērtību un SWIG nokonvertēs to par norādes tipu. Piemēram, char* tipam var nosūtīt Python simbolu virkni. Līdzīgi ir arī ar referencēm.

Ne visus datu tipus SWIG var automātiski nokonvertēt. Šādiem datu tipiem vajadzēs rakstīt manuālu konvertācijas kodu. To var izdarīt SWIG saskarnes failā, ar speciālu direktīvu.

Piemērs no SWIG dokumentācijas:

```
/* Convert from Python --> C */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}

/* Convert from C --> Python */
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

Šeit, izmantojot speciālu sintaksi, tiek pateikts, kā konvertēt int tipa mainīgus.

3.4.2 C++ atbalsts

SWIG dot iespēju izveidot saskarni arī C++ kodam. Lai nodrošinātu saskarni ar C++ klasēm, vajag uzrakstīt saskarnes failā C++ klases deklarācijas, tāpat kā C++ galvenes failā. Saskarnes izveidošanas rezultātā C++ klases tiek pārvērstas par Python klasēm. Arī klašu mantošana tiek pilnīgi atbalstīta. Pārslogošana gandrīz pilnībā tiek atbalstīta. Izņēmumi ir datu tipi, kuriem nav interpretācijas Python valodā. Un arī pārslogošanas operatori ir atbalstīti [13].

C++ nosaukumu telpas (namespaces) tiek ignorēti. Visbiežāk tie tiek izmantoti, lai atdalītu dažādu moduļu kodu, bet katru SWIG saskarnes failu var uzskatīt par vienu moduļu vai nosaukumu telpu. No saskarnes faila kods tiek izmantots Python modulī un šīs modulis pats kalpo par nosaukumu telpu. Lai automātiski izmantot C++ šablonu, vajag nodeklarēt to ar direktīvu %template.

3.4.3 Uzbūvēšana

`swig -python interface.i` komanda uzģenerē C pirmkoda failu un Python failu. C

fails satur zemā līmeņa saskarnes un Python fails augstā līmeņa. Tad ar distutils rīku var uzbūvēt paplašinājuma moduļu.

setup.py faila saturs C bibliotēkai:

```
from distutils.core import setup, Extension

setup ( ext_modules = [Extension('_swig_example',
     sources = ['interface_wrap.c'],
     libraries = ["jsmn"],
     library_dirs = ["./"],
     runtime_library_dirs = ['./']
)])
```

setup.py faila saturs C++ bibliotēkai:

```
from distutils.core import setup, Extension
setup (
    ext_modules=[Extension('_swig_example',
        sources=['interface_wrap.cxx'],
        libraries=["simplejson"],
        library_dirs = ["./"],
        runtime_library_dirs=['./']
)])
```

interface_wrap.cxx izveide ar komandu: swig -c++ -python interface.i

3.4.4 Lietošanas piemērs C bibliotēkai

interface.i fails:

```
%module swig_example
%{
#define SWIG_FILE_WITH_INIT
#include "jsmn.h"
%}
typedef enum {
    JSMN_UNDEFINED = 0,
    JSMN_OBJECT = 1,
    JSMN_ARRAY = 2,
    JSMN_STRING = 3,
    JSMN_PRIMITIVE = 4
} jsmntype_t;
typedef struct {
    jsmntype_t type;
    int start;
    int end;
    int size;
} jsmntok_t;
typedef struct {
    unsigned int pos;
    unsigned int toknext;
    int toksuper;
} jsmn_parser;

void jsmn_init(jsmn_parser *parser);
```

```
int jsmn_parse(jsmn_parser *parser, const char *js, size_t len,
    jsmntok_t *tokens, unsigned int num_tokens);

%include "carrays.i"
%array_class(jsmntok_t, jsmntok_tArray);
```

SWIG interfeisa fails satur C deklarācijas, kā arī `%array_class` direktīvu, kura izveido saskarni C masīvām, ar tipu `jsmntok_t`.

swig_usage.py:

```
from swig_example import jsmn_parser, jsmntok_tArray, jsmn_init, jsmn_parse

def get_tokens(json_string):
    parser = jsmn_parser()
    jsmn_init(parser)

    # determine needed tokens count
    tokens_count = jsmn_parse(parser, json_string,
        len(json_string), None, 0)
    # refresh parser
    jsmn_init(parser)

    tokens =  jsmntok_tArray(tokens_count)  # tokens array
    # parse json string
    jsmn_parse(parser, json_string,
        len(json_string), tokens, tokens_count)

    return tokens, tokens_count
```

tokens masīvām nevar noskaidrot elementu skaitu, tāpēc funkcija `get_tokens` atdod arī tokens_count mainīgu.

usage_example.py:

```
from swig_usage import get_tokens

f = open("json_sample.txt", "r")
json_string = f.read()
f.close()

tokens, tokens_count = get_tokens(json_string)

names = []
for i in range(tokens_count):    # for every token
    if tokens[i].type == 3:     # if token type is string
        # if token string is "nm" then append to list next token
        if json_string[tokens[i].start : tokens[i].end] == "nm":
            names.append(json_string[tokens[i+1].start : tokens[i+1].end])
print(names)
```

3.4.5 Lietošanas piemērs C++ bibliotēkai

interface.i saturs:

```
%module swig_example
%{
#define SWIG_FILE_WITH_INIT
#include "JSON.h"
#include "JSONValue.h"
typedef std::vector<JSONValue*> JSONArray;
typedef std::map<std::wstring, JSONValue*> JSONObject;
%}
typedef std::vector<JSONValue*> JSONArray;
typedef std::map<std::wstring, JSONValue*> JSONObject;
%include std_vector.i
%include std_map.i
%include std_wstring.i
%template(JSONArray) std::vector<JSONValue*>;
%template(JSONObject) std::map<std::wstring, JSONValue*>;

class JSON
{
    friend class JSONValue;

    public:
        static JSONValue* Parse(const char *data);
        static JSONValue* Parse(const wchar_t *data);
    private:
        JSON();
};

class JSONValue
{
    friend class JSON;

    public:
        JSONValue(/*NULL*/);

        bool IsString() const;
        bool IsArray() const;
        bool IsObject() const;

        const std::wstring &AsString() const;
        const JSONArray &AsArray() const;
        const JSONObject &AsObject() const;
    protected:
        static JSONValue *Parse(const wchar_t **data);
};
```

Kā redzams, saskarnes fails satur galveņu deklarācijas klasēm un typedef tipiem. Lai varētu izmantot vector, map un wstring tipus no standarta bibliotēkas, ir iekļauti attiecīgie SWIG saskarnes faili – std_vector.i, std_map.i un std_wstring.i.

Lai SWIG izveidotu saskarnes typedef tipiem - `JSONArray un JSONObject`, tika uzrakstītas SWIG šablonu direktīvas (`%template`).

swig_usage.py saturs:

```
from swig_example import JSON

f = open("json_sample.txt", "r")
json_string = f.read()
f.close()

names = []

json_value = JSON.Parse(json_string)   #  parse json string
if json_value.IsArray():
    json_array = json_value.AsArray()  # in C++ vector type
    for object in json_array:
        if object.IsObject():
            o = object.AsObject()   # object in C++ is map type
            names.append(o["nm"].AsString())

print(names)
```

3.4.6 Kopsavilkums

SWIG priekšrocības:

- automātiski uzģenerē saskarnes kodu no C/C++ galvenēm
- var izmantot arī vairākām citām valodām
- vecs projekts ar labu atbalstu
- ir iespēja rakstīt tipu konvertācijas

SWIG trūkumi:

- uzģenerētais saskarnes kods ir ļoti liels un nepiemērots lasīšanai
- grūti atkļūdot
- izveido ietinumus arī Python kodā, kas samazina ātrdarbību

3.5 SIP

SIP ir rīks automātiskai C/C++ bibliotēku piesaistījumu ģenerēšanai priekš Python. Tas tika izstrādāts 1998. gadā, lai izveidotu PyQt bibliotēku - Python piesaistes Qt bibliotēkai, bet to var izmantot jebkurai citai C/C++ bibliotēkai [14]. Līdzīgi kā SWIG rīkā, tiek izmantots speciālais saskarnes fails, kurš satur C/C++ galveņu kodu, dažādas direktīvas un konvertācijas kodu datu tipiem, kurus SIP nespēj automātiski konvertēt. Atšķirībā no SWIG SIP tika izstrādāts, lai veidotu saskarni tieši un tikai Python valodai. No sākuma SIP izstrādāja tieši C++ bibliotēku saskarņu veidošanai, ar domu, kā SWIG atbalsts C++ valodai tajā laikā bija slikts.

3.5.1 Lietošana

SIP lietošana ir ļoti līdzīga SWIG lietošanai. Šeit arī tiek izmantots saskarnes fails, bet tikai ar paplašinājumu .sip, un nedaudz atšķirīgu sintaksi. Arī šeit vajag kopēt C/C++ deklarācijas funkcijām, struktūrām un klasēm, lai SIP uzģenerētu saskarnes kodu [15].

Atšķirībā no SWIG SIP rikām nav iebūvēta atbalsta C++ standarta bibliotēkas tipiem, piemēram, wstring, konteinera tipiem – vector, map un citiem. Priekš tiem vajadzēs rakstīt vai meklēt internetā kodu, lai konvertētu tipus no Python uz C++ un otrādi. Šīs kods tiek rakstīts speciālos blokos .sip failā. Vienā blokā tiek konvertēts Python tips uz C++ un otrajā blokā C++ tips uz Python tipu. Blokos jau ir pieejams mainīgais, kuru jākonvertē, un tad atliek tikai nokonvertēt un atgriezt to. Šīs koda gabalus SIP spēj integrēt ar uzģenerētu kodu. Daudziem tipiem konvertācijas kodu var atrast internetā. Tad to var iekļaut .sip failā ar *include* direktīvu. Tas būs redzams C++ piemērā.

Lai izveidotu saskarni C/C++ masīvām, ir pieejama speciālā C funkcija:

```
PyObject *sipConvertToTypedArray(void *data, const sipTypeDef *td,
      const char *format, size_t stride, SIP_SSIZE_T len)
```

Šī funkcija konvertē C masīvu par Python objektu, caur kuru var piekļūt masīva elementiem.

3.5.2 Uzbūvēšana

SIP rīkam ir integrācija ar distutils.

setup.py saturs C bibliotēkai:

```
from distutils.core import setup, Extension
import sipdistutils
```

```
setup(
  ext_modules=[
    Extension("sip_example", ["interface.sip"],
                             include_dirs=["./"],
                             libraries=["jsmn"],
                             library_dirs = ["./"],
                             runtime_library_dirs=['./'])],

  cmdclass = {'build_ext': sipdistutils.build_ext}
)
```

C++ bibliotēkai tāds pats setup.py, tikai bibliotēkas nosaukums – simplejson.

3.5.3 Lietošanas piemērs C bibliotēkai

SIP saskarnes fails:

```
%Module(name=sip_example, language="C")

%ModuleHeaderCode
#include "jsmn.h"
%End

struct jsmntok_t {
    unsigned int type;
    int start;
    int end;
    int size;
};

struct jsmn_parser {
    unsigned int pos;
    unsigned int toknext;
    int toksuper;
};

void jsmn_init(jsmn_parser *parser);
int jsmn_parse(jsmn_parser *parser, const char *js, int len,
    jsmntok_t *tokens, unsigned int num_tokens);
```

Var redzēt, ka typedef struct vietā izmantots C++ deklarācijas veids – vienkārši struct. SIP neatbalsta typedef struct deklarāciju, turklāt izmantojot šo C++ sintaksi, SIP uzģenerē nedaudz nepareizu C kodu. Lai tas kompilētos un strādātu, vajadzēja noņemt uzģenerētā C failā struct atslēgvārdu, vietās, kur tiek deklarēti `jsmntok_t` un `jsmn_parser` tipi.

Var secināt, ka C valodas atbalsts SIP rīkam ir slikts.

Izmantošana Python kodā:

```
from sip_example import jsmn_parser, jsmn_init, jsmn_parse,
get_jsmntok_t_C_array, get_jsmntok_t_array

def get_tokens(json_string):
    parser = jsmn_parser()
    jsmn_init(parser)
```

```
    # determine needed tokens count
    tokens_count = jsmn_parse(parser, str.encode(json_string),
        len(json_string), None, 0)
    # refresh parser
    jsmn_init(parser)

    tokens =  get_jsmntok_t_C_array(tokens_count)   # tokens array
    # parse json string
    jsmn_parse(parser, str.encode(json_string),
        len(json_string), tokens, tokens_count)

    # convert to usable python array
    return get_jsmntok_t_array(tokens, tokens_count)
```

Kā redzams, Python kodā viss notiek tāpat kā SWIG piemērā, izņemot to, ka darbam ar C masīvu tiek izmantotas speciālas C funkcijas. Šīs funkcijas šajā darbā netiek apskatītas.

3.5.4 Lietošanas piemērs C++ bibliotēkai

interface.sip saturs:

```
%Module(name=sip_example, language="C++")

%ModuleHeaderCode
#include "JSON.h"
#include "JSONValue.h"
%End

%Include wstring.sip
%Include vector.sip
%Include map.sip

typedef std::vector<JSONValue*> JSONArray;
typedef std::map<std::wstring, JSONValue*> JSONObject;

class JSON
{
    %TypeHeaderCode
    #include "JSON.h"
    %End

    public:
        static JSONValue* Parse(const char *data);
        static JSONValue* Parse(const wchar_t *data);
    private:
        JSON();
};

class JSONValue
{
    %TypeHeaderCode
    #include "JSONValue.h"
    %End

    public:
        bool IsString() const;
```

```
        bool IsArray() const;
        bool IsObject() const;

        const std::wstring &AsString() const;
        const JSONArray &AsArray() const;
        const JSONObject &AsObject() const;

};
```

Saskarnes fails satur vajadzīgas C++ deklarācijas, kā arī tiek iekļauti papildus SIP saskarnes faili – wstring.sip, map.sip un vector.sip. Tie satur saskarnes uz JSONArray un JSONObject tipiem. SIP nevar automātiski darboties ar wstring, map un vector datu tipiem, tāpēc ir nepieciešams uzrakstīt, kā konvertēt C++ objektu uz Python objektu un otrādi. Tad Python kodā nekas papildus rakstīt nevajag. Šie saskarnes faili netiek iekļauti darbā, lielā apjoma dēļ (katrs fails satur vairāk nekā 50 koda rindiņas).

3.5.5 Kopsavilkums

SIP priekšrocības:

- automātiski uzģenerē saskarnes kodu no C/C++ galvenēm
- uzģenerētais saskarnes kods ir kompakts un piemērots lasīšanai
- labs C++ valodas atbalsts
- ātrs saskarnes kods

SIP trūkumi:

- C++ standarta bibliotēkas datu tipu izmantošanai jāraksta speciālie saskarnes faili
- slikts C valodas atbalsts

3.6 Boost Python

Boost Python ir bibliotēka, kura ļauj saskarties Python ar C++. Tas neizmanto speciālus rīkus – tikai C++ kompilatoru, un šī bibliotēka ir daļa no C++ Boost bibliotēkas kolekcijas. Boost Python tika izstrādāts tādā veidā, lai nepieciešamas izmaiņas C++ kodā būtu minimālas. Bibliotēka izmanto meta-programmēšanas tehnikas, kuras vienkāršo sintaksi lietotājam [16]. Izmantojot šo C++ kodu, tiek definēts ietinums C++ bibliotēkai.

3.6.1 Lietošana

Boost Python bibliotēka dot iespēju piesaistīt C++ klases vai struktūras Python klasēm. Lai to izdarīt, nepieciešams uzrakstīt speciālu C++ kodu, kurš apraksta ietinumu un satur tādu informāciju kā, moduļa un klases nosaukumu, klases atribūtus un metodes, un citu informāciju. Nav obligāti jāraksta visas klases metodes un atribūti, bet tikai tie, kuri vajadzīgi Python modulī. Boost Python atbalsta arī C++ mantošanu, operatoru pārslogošanu, funkciju un metožu pārslogošanu, enum tipus, noklusējuma argumentus (default arguments) [17].

Boost Python izmanto object klasi, lai ietināt (to wrap) PyObject*. Šī klase nodrošina dažādu funkcionalitāti ar PyObjekt*, piemēram, referenču skaitīšanu. Šis klases objektus var izmantot C++ kodā, lai izveidotu Python objektus no C++ objektiem, vienkārši norādot objekt tipu C++ mainīgajiem. Boost Python bibliotēkai ir pieejami dažādi C++ tipi, kas atbilst Python iebūvētiem tipiem. Tie ir list, dict, tuple, str, long_, enum. Tos var izmantot C++ kodā kā atbilstošus Python tipus. No object tipa C++ vērtības tiek dabūtas ar extract<T> funkciju [18]. Boost Python arī atbalsta C++ iteratoru saskarņu izveidošanu.

Boost Python ļauj kontrolēt objektu dzīvošanas ilgumu ar referencēm un norādēm, pateicoties izsaukumu noteikumiem (call policies). Dažādi noteikumi nosaka, kā tiks pārvaldīti objekti, izsaucot funkcijas vai metodes.

with_custodian_and_ward pasaka, lai viens funkcijas arguments dzīvotu tik, cik otrs arguments.

with_custodian_and_ward_postcall noteikums tura argumentu un funkcijas rezultātu. return_by_value noteikums nosaka, ka atgriežama vērtība tiek kopēta uz jauno Python objektu. return_internal_reference gadījumā Python objekts tiek būvēts apkārt atgriežamai vērtībai un C++ objekts pastāv, kamēr pastāv Python objekts [19].

3.6.2 Uzbūvēšana

setup.py:

```
from distutils.core import setup, Extension

setup (
    ext_modules=[Extension('boost_example',
        sources=['boost_example.cpp'],
        libraries=['simplejson', 'boost_python3'],
        library_dirs = ['.', '/usr/local/lib'],
        runtime_library_dirs=['./'],
        include_dirs=['.', '/usr/local/include', '/usr/include/python3.4m']
    )]
)
```

Šeit papildus tiek norādīta boost python bibliotēkas atrašanas vieta un nosaukums, kā arī galveņu direktorijas. Šajā gadījumā arī vajadzēja norādīt Python galveņu direktoriju. Atšķirībā no Linux distributīva un no tā, kā tika uzbūvēta boost bibliotēka, direktorijas var atšķirties.

3.6.3 Lietošanas piemērs

boost_example.cpp:

```
#include "JSON.h"
#include "JSONValue.h"
#include <boost/python.hpp>

using namespace std;
using namespace boost::python;

struct JSONValueToPython
{
    static PyObject* convert(JSONValue* const& my_obj)
    {
        object o = object(*my_obj);
        return incref(o.ptr());
    }
};

struct JSONObjectToPython
{
    static PyObject* convert(JSONObject const& map)
    {
        typename JSONObject::const_iterator iter;
        dict dictionary;

        for (iter = map.begin(); iter != map.end(); ++iter) {
            dictionary[iter->first] = iter->second;
        }

        return incref(dictionary.ptr());
    }
};

BOOST_PYTHON_FUNCTION_OVERLOADS(JSON_overloads, JSON::Parse, 1, 1)

BOOST_PYTHON_MODULE(boost_example)
{
    to_python_converter<JSONValue*, JSONValueToPython>();
```

```
    to_python_converter<JSONObject, JSONObjectToPython>();

    class_<JSON>("JSON", no_init)
        .def("Parse", static_cast<JSONValue* (*) (const char*)>(&JSON::Parse),
            return_internal_reference<>(), JSON_overloads())
        .staticmethod("Parse")
    ;

    class_<JSONValue>("JSONValue", no_init)
        .def("IsString", &JSONValue::IsString)
        .def("IsObject", &JSONValue::IsObject)
        .def("IsArray", &JSONValue::IsArray)
        .def("AsArray", &JSONValue::AsArray, return_internal_reference<>())
        .def("AsObject", &JSONValue::AsObject,
             return_value_policy<return_by_value>())
        .def("AsString", &JSONValue::AsString,
             return_value_policy<return_by_value>())
    ;

    class_<JSONArray>("JSONArray")
        .def("__iter__", boost::python::iterator<vector<JSONValue*> >())
    ;
}
```

`BOOST_PYTHON_MODULE(boost_example)` definē boost_example Python moduli ar saskarnēm uz JSON un JSONValue klasēm. JSON klasei tiek norādīta statiskā metode Parse. Tā ir pārslogojama metode. Lai pārslogošana darbotos, tiek izmantots makro – `BOOST_PYTHON_FUNCTION_OVERLOADS`. Tas tiek izmantots parastām funkcijām un statiskām metodēm. Instances metodēm ir cits makro def() metodei otrais arguments ir norāde uz funkciju. Izmantojot šo norādi boost python noskaidro vajadzīgu informāciju par funkciju, kurai tiek veidota saskarne. JSONValue klasei līdzīgi tiek norādītas metodes.

Grūtības sagādā tipi JSONArray un JSONObject (vector un map). Boost automātiski nevar izveidot tiem saskarni. JSONArray tika nodefinēts kā klase ar __iter__ atribūtu. Python valodā šīs atribūts nozīmē, ka caur objektu var iterēt un saņemt vērtības. Tas ir tieši tas, kas vajadzīgs. Iteratora atgriežama vērtība ir ar JSONValue* tipu un boost nezināja, kā to pārveidot par Python tipu, tāpēc tika nodefinēta statiskā metode `JSONValueToPython::convert`, kura pārveido JSONValue* par Python objektu. Ar tādu pašu paņēmienu tika pateikts, kā pārveidot JSONObject tipu par Python. `JSONObjectToPython::convert` saņem JSONObject objektu un pārveido to par Python vārdnīcu, izmantojot boost::python::dict tipu. wstring tipu boost spēja automātiski konvertēt.

3.6.4 Kopsavilkums

Boost Python priekšrocības:

- labs C++ valodas atbalsts
- vecā un zināmā bibliotēka

Boost Python trūkumi:

- izmantošanai vajadzīgas labas C++ valodas zināšanas
- lielā izmēra bibliotēka

3.7 PyBind11

PyBind11 ir neliela, tikai galveņu bibliotēka, speciāli izveidota, lai veidotu C++ koda piesaistes. Tā padara C++ tipus pieejamus priekš Python. Bibliotēkas mērķi un sintakse ir līdzīgi Boost Python. Iemesls pybind11 izveidošanai ir, kā Boost ir ļoti liela un sarežģīta bibliotēka, kura atbalsta gandrīz visus kompilatorus. PyBind11 izmanto C++ 11 standarta iespējas, kas samazināja koda daudzumu. Bibliotēka ir atkarīga tikai no paša Python un C++ standarta bibliotēkas. Atšķirībā no Boost Python, tā atbalsta arī PyPy2.7 implementāciju. Kā uzrakstīts oficiālajā mājaslapā, bibliotēka izauga pāri Boost Python un ļauj rakstīt vienkāršāku saskarnes kodu [20].

3.7.1 Lietošana

Saskarņu kods ir ļoti līdzīgs boost python kodam. Viena no atšķirībām ir, ka pybind11 spēj automātiski konvertēt C++ standarta bibliotēkas konteineru tipus uz Python iebūvētiem tipiem - list, set un dict un otrādi. Ir atbalstīti gandrīz visi C++ konteineri, tikai vajag iekļaut galveni #include <pybind11/stl.h>. Automātiski konvertējot konteinerus, pybind11 izveido datu kopijas, kas var samazināt ātrumu, un arī padara neiespējamu mainīt konteinera datus pa tiešo. Lai to izlabotu, var izmantot speciālus necaurredzamus tipus (opaque types). Tos deklarē ar galveni <pybind11/stl_bind.h> un makrosu PYBIND11_MAKE_OPAQUE [21].

Tāpat kā ar boost python, var izmantot dažādus izsaukumu noteikumus atgriežamiem objektiem no C++:

return_value_policy::take_ownership – atdod objektu kā referenci un objekta pārvaldīšana notiek Python pusē. Jā C++ pusē objekta dati tiks nodzēsti, tad var iestāties neparedzamas kļūdas.

return_value_policy::copy – atdod objekta kopiju. Python un C++ puses neietekmē viens otru.

return_value_policy::reference – atdod objekta referenci, bet par objekta pārvaldīšanu atbildīgs C++. Ir vēl citi dažādi noteikumi [22].

Pybind11 atbalsta saskarņu veidošanu dažādām modernām C++ konstrukcijām,

piemēram, anonīmām funkcijām, dažiem C++ 17 konteineriem – optional, variant.

3.7.2 Uzbūvēšana

setup.py:

```
from distutils.core import setup, Extension

setup (
    ext_modules=[Extension('pybind11_example',
        sources=['pybind11_example.cpp'],
        libraries=['simplejson'],
        library_dirs = ['./'],
        runtime_library_dirs=['./'],
        include_dirs=['./', '/usr/include/python3.4m'],
        extra_compile_args=['-std=c++11']
    )]
)
```

3.7.3 Lietošanas piemērs

```
#include "JSON.h"
#include "JSONValue.h"
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

using namespace std;
namespace py = pybind11;

PYBIND11_PLUGIN(pybind11_example) {
    py::module m("pybind11_example");

    py::class_<JSON>(m, "JSON")
        .def("Parse", static_cast<JSONValue* (*) (const char *)>(&JSON::Parse),
             py::return_value_policy::take_ownership)
        .def("Parse", static_cast<JSONValue* (*) (const wchar_t *)>(&JSON::Parse),
             py::return_value_policy::take_ownership)
    ;

    py::class_<JSONValue>(m, "JSONValue")
        .def("IsString", &JSONValue::IsString)
        .def("IsObject", &JSONValue::IsObject)
        .def("IsArray", &JSONValue::IsArray)
        .def("AsArray", &JSONValue::AsArray, py::return_value_policy::reference)
        .def("AsObject", &JSONValue::AsObject, py::return_value_policy::reference)
        .def("AsString", &JSONValue::AsString, py::return_value_policy::reference)
    ;

    return m.ptr();
}
```

Piemērs ir gandrīz identisks ar boost python piemēru. Vislielākā atšķirība ir, ka šeit nebija vajadzīga manuāla konvertācija JSONValue * un JSONObject tipam. Izmantošana Python kodā ir pilnīgi identiskā ar boost python piemēru.

3.7.4 Kopsavilkums

pybind11 priekšrocības:

- labs C++ valodas atbalsts
- mazā izmēra bibliotēka
- ļoti laba dokumentācija
- automātiskā konvertācija starp vairākiem datu tipiem
- iespēja kontrolēt objektu pārvaldīšanu caur izsaukumu noteikumiem
- bibliotēka aktīvi attīstās

pybind11 trūkumi:

- vajadzīgs C++ 11 kompilators

3.8 PyBindGen

PyBindGen ir rīks, ar kuru var uzģenerēt piesaistes priekš C/C++ APIs. PyBindGen izstrādāts Python valodā un tiek kontrolēts caur Python. Atšķirībā no citiem līdzīgiem rīkiem, tas uzģenerē lasāmu C/C++ kodu.

3.8.1 Lietošana

Lai uzģenerētu C/C++ saskarnes kodu, ir jāuzraksta Python skripts, kurā, izmantojot, pybindgen moduļa funkcijas, tiek uzrādīta nepieciešama informācija paplašinājuma ģenerācijai. Palaižot šo skriptu, tiek uzģenerēts piesaistes C vai C++ kods. Tad šīs kods tiek kompilēts un savienots ar piesaistītu bibliotēku. Rezultāts ir Python paplašinājuma modulis, gatavais importēšanai Python kodā [23]. PyBindGen iekļauj arī rīku, lai automātiski uzģenerētu skriptu no bibliotēkas galveņu failiem. Tas izmanto gccxml un pygccxml bibliotēkas.

Lai ietināt C funkcijas, pietiek tikai norādīt parametru un atgriežamo vērtību tipus Python skriptā. Pēc paplašinājuma moduļa izveides, funkcijas var izsaukt kā parastas Python funkcijas. Padoto argumentu datu tipu konvertācija notiek automātiski. Kā funkcijas argumentus un atgriežamo vērtību var lietot arī C struktūras un norādes [23].

PyBindGen ļauj kontrolēt objekta pārvaldību - funkciju argumentiem un atgriežamām vērtībām. Var norādīt, kura puse – Python vai C/C++ ir atbildīga par objekta nodzēšanu.

Būtiskais ierobežojums ir saistīts ar C masīva lietošanu. Ar bibliotēkas palīdzību nav iespējams izveidot masīvu Python kodā, lai nosūtītu to C funkcijai. Arī ja C funkcija atgriež kādu masīvu, tiek atgriezts norādes tips, ar kuru nevar nolasīt masīva vērtības.

Modulis, atbildīgais par automātisko Python koda ģenerēšanu - pybindgen.gccxmlparser neatbalsta Python 3, kas tika atklāts piemēra veidošanas laikā. Modulis arī neatbalsta jaunas pygccxml versijas.

3.8.2 Uzbūvēšana

Vispirms ģeneratora skripts uzģenerē C++ kodu, kas satur saskarnes kodu ar komandu:

```
python3 generator_script.py > pybindgen_example.cpp
```

setup.py saturs:

```
from distutils.core import setup, Extension

demo = Extension('pybindgen_example', sources = ['pybindgen_example.cpp'],
    include_dirs = ['./'],
    runtime_library_dirs=['./'],
    library_dirs = ["./"],
    libraries=["jsmn"])

setup(ext_modules = [demo])
```

3.8.3 Lietošanas piemērs C bibliotēkai

generator_script.py:

```
import pybindgen
import sys

mod = pybindgen.Module('pybindgen_example')
mod.add_include('"jsmn.h"')

token_struct = mod.add_struct('jsmntok_t')
token_struct.add_instance_attribute('type', 'int')
token_struct.add_instance_attribute('start', 'int')
token_struct.add_instance_attribute('end', 'int')
mod.add_typedef(token_struct, 'jsmntok_t')
parser_struct = mod.add_struct('jsmn_parser')
mod.add_typedef(parser_struct, 'jsmn_parser')
mod.add_function('jsmn_init', None, [pybindgen.param ('jsmn_parser *',
    'parser', transfer_ownership=False)])
mod.add_function('jsmn_parse', pybindgen.retval ('int'),
    [pybindgen.param ('jsmn_parser *', 'parser', transfer_ownership=False),
    pybindgen.param ('const char *', 'js', transfer_ownership=False),
    pybindgen.param ('size_t', 'len'),
    pybindgen.param ('jsmntok_t *', 'tokens', transfer_ownership=False),
    pybindgen.param ('unsigned int', 'num_tokens')])

mod.generate(sys.stdout)
```

Piemērs nav pabeigts, jo ar pybindgen nav iespējams izveidot masīvu ar jsmntok_t tipa elementiem.

3.8.4 Lietošanas piemērs C++ bibliotēkai

generator_script.py:

```
import pybindgen

mod = pybindgen.Module('pybindgen_example')
mod.add_include('"JSON.h"')
mod.add_include('"JSONValue.h"')
JSON = mod.add_class('JSON')
JSON = mod.add_class('JSONValue')
JSONArray = mod.add_container('std::vector<JSONValue*>',
    pybindgen.ReturnValue.new('JSONValue*',
      reference_existing_object=True),
    'vector', custom_name="JSONArray")
JSONObject = mod.add_container('std::map<wstring, JSONValue*>',
      ('std::wstring', pybindgen.ReturnValue.new('JSONValue*',
      reference_existing_object=True)),
'map', custom_name="JSONObject")
```

Piemērs nav pabeigts, jo PyBindGen nemāk konvertēt std:wstring tipu.

3.8.5 Kopsavilkums

PyBindGen priekšrocības:

- saskarnes definēšana notiek Python kodā
- uzģenerētais saskarnes kods ir kompakts un piemērots lasīšanai
- pieejams modulis gccxmlparser, ar kuru palīdzību var automātiski uzģenerēt saskarni

PyBindGen trūkumi:

- nav automātiskā C masīva saskarņu izveides
- modulis gccxmlparser neatbalsta Python 3, kā arī jaunās pygccxml versijas

3.9 Shiboken

Shiboken ir saskarņu ģenerators, kurš parsē C++ bibliotēkas galvenes un, balstoties uz speciālu XML failu, veic izmaiņas. Šīs rīks tika izmantots PySide bibliotēkas izstrādei. Tas ir Qt bibliotēkas saskarnes. No sākuma PySide izmantoja boost python, bet tas ģenerēja pārāk lielus binārus failus [24].

3.9.1 Lietošana

Shiboken ģenerē saskarnes kodu, izmantojot bibliotēkas galveņu failus un tā saucamu sistēmas tipu (typesystem) xml failu. Tā kā visa uzmanība ir ap xml failu. Tajā tiek norādīti visi tipi, kuriem vajadzīga konvertācija. Shiboken spēj automātiski nokonvertēt dažādus primitīvus tipus, norādes, references, objektus. Ir datu tipi, kurus Shiboken nemāk konvertēt. Tad ir jāraksta konvertācijas kods speciālā xml tagā. Lai atvieglotu šī koda rakstīšanu, kā arī padarīt to vispārīgāko, Shiboken ļauj izmantot speciālus mainīgus, kuri ir aizvietoti ar pareizām vērtībām. Piemēram, konvertējot kādu C++ tipu uz Python, %in mainīgais ir C++ vērtība, kuru ir jākonvertē, un %INTYPE ir mainīga tips [25].

Shiboken ļauj modificēt ģenerējamo kodu, izmantojot inject-code tagu. Šajā tagā tiek rakstīts kods, kuru Shiboken iekļauj noteiktās vietās. Piemēram, var iekļaut kodu pirms klases ietinuma (Python paplašinājuma tipa) inicializācijas, lai modificētu to struktūru. Kodu var iekļaut arī pirms vai pēc Python moduļa inicializācijas, pirms vai pēc C++ metodes izsaukuma, un dažas citās vietās [26].

Lai nodrošinātu drošu objektu pārvaldīšanu, Shiboken dažās situācijās padara neiespējamu sūtīt objektu kā argumentu un izsaukt objekta metodes. Piemēram, tas notiek, kad objekts tiek sūtīts C++ funkcijai, tad ir pieņemts, ka šī funkcija atbild par objektu [27].

3.9.2 Uzbūvēšana

setup.py:

```
from distutils.core import setup, Extension

setup (
    ext_modules=[Extension('shiboken_example',
        sources=['./out/shiboken_example/json_wrapper.cpp',
            './out/shiboken_example/jsonvalue_wrapper.cpp',
            './out/shiboken_example/shiboken_example_module_wrapper.cpp'],
        libraries=['simplejson', 'shiboken.cpython-34m'],
        library_dirs = ['./', '/usr/lib64/python3.4/site-packages/Shiboken'],
        runtime_library_dirs=['./', '/usr/lib64/python3.4/site-packages/Shiboken'],
        include_dirs=['.',
            '/usr/lib64/python3.4/site-packages/Shiboken/include/shiboken',
            '/usr/include/python3.4m']
```

```
    )]
)
```

Shiboken katram galveņu failam ģenerē atsevišķu izejas failu. Moduļa definīcijai arī tiek uzģenerēts atsevišķs fails. Kompilēšanai nepieciešama Shiboken bibliotēka, tāpēc ir norādīts tas nosaukums un atrašanas vieta.

3.9.3 Lietošanas piemērs

```
<?xml version="1.0"?>
<typesystem package="shiboken_example">
    <primitive-type name="bool"/>
    <primitive-type name="char"/>
    <primitive-type name="std::wstring">
        <conversion-rule>
            <native-to-target>
            return PyUnicode_FromWideChar(%in.c_str(), -1);
            </native-to-target>
        </conversion-rule>
    </primitive-type>
    <object-type name="JSON" />
    <object-type name="JSONValue" />

    <rejection class="JSON" function-name="Stringify"/>

    <container-type name="std::map" type="map">
        <include file-name="map" location="global"/>
        <conversion-rule>
            <native-to-target>
            PyObject* %out = PyDict_New();
            %INTYPE::const_iterator it = %in.begin();
            for (; it != %in.end(); ++it) {
                %INTYPE_0 key = it->first;
                %INTYPE_1 value = it->second;
                PyDict_SetItem(%out,
                               %CONVERTTOPYTHON[%INTYPE_0](key),
                               %CONVERTTOPYTHON[%INTYPE_1](value));
            }
            return %out;
            </native-to-target>
            <target-to-native>
                <add-conversion type="PyDict">
                PyObject* key;
                PyObject* value;
                Py_ssize_t pos = 0;
                while (PyDict_Next(%in, &amp;pos, &amp;key, &amp;value)) {
                    %OUTTYPE_0 cppKey = %CONVERTTOCPP[%OUTTYPE_0](key);
                    %OUTTYPE_1 cppValue = %CONVERTTOCPP[%OUTTYPE_1](value);
                    %out.insert(%OUTTYPE::value_type(cppKey, cppValue));
                }
                </add-conversion>
            </target-to-native>
        </conversion-rule>
    </container-type>

    <container-type name="std::vector" type="vector">
        <include file-name="vector" location="global"/>
        <conversion-rule>
            <native-to-target>
            %INTYPE::size_type vectorSize = %in.size();
            PyObject* %out = PyList_New((int) vectorSize);
            for (%INTYPE::size_type idx = 0; idx &lt; vectorSize; ++idx) {
                %INTYPE_0 cppItem(%in[idx]);
                PyList_SET_ITEM(%out, idx, %CONVERTTOPYTHON[%INTYPE_0](cppItem));
```

```
            }
            return %out;
            </native-to-target>
            <target-to-native>
                <add-conversion type="PySequence">
                Shiboken::AutoDecRef seq(PySequence_Fast(%in, 0));
                int vectorSize = PySequence_Fast_GET_SIZE(seq.object());
                %out.reserve(vectorSize);
                for (int idx = 0; idx &lt; vectorSize; ++idx ) {
                    PyObject* pyItem = PySequence_Fast_GET_ITEM(seq.object(), idx);
                    %OUTTYPE_0 cppItem = %CONVERTTOCPP[%OUTTYPE_0](pyItem);
                    %out.push_back(cppItem);
                }
                </add-conversion>
            </target-to-native>
        </conversion-rule>
    </container-type>
</typesystem>
```

Shiboken saskarnes xml failā ir nodefinēti visi tipi, kuriem vajadzīga konvertācija no C++ uz Python un otrādi. Arī primitīvus tipus, tādus kā bool vai char vajag norādīt, bet konvertācija tiem notiek automātiski. wstring tipu shiboken nemāk automātiski nokonvertēt, tāpēc ir uzrakstīts konvertācijas kods no C++ uz Python. Tāpat ir ar konteinera tipiem map un vector, tikai tiem konvertācijas kods ir abpusējs. Īstenībā konvertācija šeit vajadzīga tikai uz Python, bet piemērs nekompilējas bez Python uz C++ konvertācijas definīcijas. Konvertācijas kods vector un map tipiem tika paņemts no piemēriem, no Shiboken paketes.

3.9.4 Kopsavilkums

Shiboken priekšrocības:

- ģenerēta koda kontrole
- automātiski uzģenerē saskarnes kodu no C++ hederiem
- uzģenerē kompaktu un lasāmu kodu

Shiboken trūkumi:

- nav iebūvētā atbalsta STL konteinera tipiem

4 REZULTĀTU APKOPOJUMS

Šajā darbā iegūtā informācija par bibliotēkām un koda ģeneratoriem ir sagrupēta un apkopota šajā nodaļā. Šī nodaļa satur arī visu apskatīto bibliotēku un koda ģeneratoru salīdzinājumu. Spriežot pēc iepriekšējam nodaļām, bibliotēkas un koda ģeneratorus var iedalīt četrās grupās.

4.1 Python bibliotēkas izsauc C funkcijas pa tiešo (ctypes, cffi ABI)

Abas bibliotēkas ļauj izsaukt C funkcijas no koplietojuma bibliotēkām, izmantojot libffi. Tas ir vienīgas bibliotēkas, kur saskarnēm nav vajadzīga kompilēšana, jo tā tiek rakstīta Python kodā. Abas bibliotēkas ir pazīstamas un efektīvas. Ar tam var padot dažāda tipa argumentus C funkcijām, tai skaitā norādes un struktūras. Saņemot vērtības no C funkcijām, tas tiek pārveidotas par Python tipiem, no kuriem var nolasīt vērtības. Vienīga pamanīta ctypes priekšrocība pāri cffi, ir, ka ctypes nāk no Python standarta bibliotēkas. Citām Python implementācijām – PyPy, Jython, IronPython arī ir pieejama ctypes bibliotēka. cffi ir pieejama no CPython un PyPy implementācijām. cffi priekšrocība ir vienkāršākā lietošana. Ar cffi ir mazāk jālieto bibliotēkas API nekā ar ctypes, jo cffi paņem vajadzīgu informāciju no C bibliotēkas galveņu koda. Ātruma ziņā, cffi arī bija labāks. cffi piemērs izpildījās 21.3 sekundēs un ctypes 26.3 sekundēs.

4.2 Koda ģeneratori (cffi API, SWIG, SIP, PyBindGen, Shiboken)

Koda ģeneratori uzģenerē Python paplašinājumus, izmantojot galveņu failus un speciālu saskarnes kodu. cffi atbalsta tikai C, Shiboken atbalsta tikai C++, pārējie atbalsta gan C, gan C++ valodas. Visiem rīkiem var kontrolēt, kādu daļu no bibliotēkas piesaistīt. cffi rikām ir visvienkāršākā lietošana, salīdzinot ar citiem. Viņš pilnīgi automātiski uzģenerē kodu, ar kuru caur vienkāršu API var saskarties Python kodā. Uzģenerētais kods ir kompakts un ātrs. SWIG uzģenerētais kods bija vislielākais un arī vislēnākais no visiem rīkiem. Ar to var ģenerēt saskarnes kodu ne tikai Python, bet arī citām valodām. SIP rīks ir ļoti līdzīgs SWIG, bet izveidots tieši Python valodai. Tas ģenerē kompaktu un ātru kodu. SIP būtiskais trūkums ir slikts C valodas atbalsts. Ar to neizdevās izveidot strādājošo piemēru C bibliotēkai. Shiboken ģenerēta koda lielums un ātrums gandrīz identisks ar SIP. Atšķirībā no SWIG un SIP, Shiboken ļauj vairāk ietekmēt ģenerēto kodu. PyBindGen izcēlās ar to, ka ir uzrakstīts un lietojams pilnīgi Python valodā. Ģeneratoram ir būtiskie trūkumi, kuri neļāva pabeigt piemērus C un C++ bibliotēkai. Tas neļauj izveidot saskarni C masīvām ar patvaļīgiem tipiem

un tas nemāk konvertēt wstring tipu. Citi rīki arī nemāk konvertēt šo tipu, bet tiem varēja atrast un iekļaut speciālu saskarnes kodu.

4.3 C++ bibliotēkas (boost python, pybind11)

Šis bibliotēkas palīdz izveidot saskarni uz C++ bibliotēkām, izmantojot tikai C++ kodu. Abas bibliotēkas strādā pēc līdzīgā principa, un abām ir līdzīga sintakse. PyBind11 ir jaunā bibliotēka, kura izveidota, lai novērstu boost python trūkumus. Atšķirībā no boost python, pybind11 ir tikai galveņu bibliotēka, kas nozīmē, ka to atsevišķi nevajag būvēt. pybind11 izmanto C++11 standarta jaunās iespējas, kuras ļāva samazināt bibliotēkas izmēru. Boost Python bieži pieminēts trūkums ir liels izmērs. PyBind11 bibliotēkai ir arī citas priekšrocības – automātiskā STL konteineru konvertācija, dažādu jaunu C++ iespēju atbalsts un aktīvā projekta attīstība. Vienīgais pybind11 trūkums ir nepieciešamība pēc C++11 kompilatora. Abām bibliotēkām ir vairāk iespēju kontrolēt objektu pārvaldību, nekā koda ģeneratoriem.

4.4 Cython valoda un koda ģenerators

Cython varēja pieskaitīt pie ģeneratoriem, bet tā darbība un lietošana pārāk atšķirās. Cython ir valoda, ar kuru var rakstīt Python paplašinājumus. Tas galvenais uzdevums ir jauno paplašinājumu rakstīšana, bet to var arī izmantot, lai veidotu saskarni uz C/C++ bibliotēkām. Cython sintakse ir hibrīds no Python un C/C++ sintakses. Šis rīks visvairāk ļauj kontrolēt saskarnes kodu. Tas ir pluss. Mīnuss ir, ka var nākties rakstīt vairāk koda, nekā ar citiem rīkiem. Cython rīkam ir vairāki plusi - ģenerētais kods ir labi optimizēts, Cython ir plaši izmantots un ātri attīstās, un tam ir laba dokumentācija.

4.5 Labākie rīki

Pēc darba autora viedokļa vislabākais rīks C bibliotēku saskarņu izveidei ir cffi API. Tam bija visvienkāršākā lietošana, kā arī uzģenerētais kods bija viskompaktākais un visātrākais. Ja vajadzīga labākā saskarnes koda kontrole, tad labāk izmantot Cython.

Priekš C++ bibliotēkām, pēc autora viedokļa vislabākais ir PyBind11. Tam sanāca visvienkāršākais saskarnes kods. Ātrums tādā pašā līmenī, kā koda ģeneratoriem (pat nedaudz ātrāks). Ja vajadzīga saskarnes koda kontrole, tad atkal, Cython ir labāks. No koda ģeneratoriem, labākie ir SIP un Shiboken. Tiem ir daudz mazākais ģenerētā koda apjoms nekā SWIG, kā arī ātrums ievērojami labāks.

4.6 Rīku ātrdarbību rezultāti

Rīku ātrdarbības noteikšanas metode ir aprakstīta “TESTĒŠANAS VIDE” nodaļā.

Tabulās 4-1 un 4-2 ir arī iekļauts uzģenerētā koda daudzums katram rīkam un nozīmīgi rīka lietotāji.

Tabula 4-1

Rīki C bibliotēkām

Rīks	Ātrums(s)	Uzģenerēta koda daudzums (rindas)	Nozīmīgi rīka lietotāji (bibliotēkas)
ctypes	26.3	-	PyOpenGL
cffi ABI	21.3	-	cryptography
cffi API	20.8	693	
Cython	22.8	2584	Kivy (multi-platformu lietotāja interfeisa ietvars), gevent (tīklojuma bibliotēka)
SWIG	239.9	4861	

Tabula 4-2

Rīki C++ bibliotēkām

Rīks	Ātrums (s)	Uzģenerēta koda daudzums (rindas)	Nozīmīgi rīka lietotāji (bibliotēkas)
Cython	17.6	3494	SAGE (matemātikas programmatūra)
SWIG	40	10535	wxPython
SIP	23.3	1171	PyQT, PyKDE
Boost Python	23.9	-	dlib, graph-tool
PyBind11	23.2	-	PyRosetta (molekulu modelēšana)
Shiboken	25.9	1148	PySide (QT ietvara saskarnes)

Pētījumā laikā tika savākta informācija par STL konteineru atbalstu katram rīkam, kura redzama 4-3 tabulā.

Tabula 4-3

C++ STL konteineru atbalsts

Rīks	vector	map	list	set	pair
Cython	jā	jā	jā	jā	jā
SWIG	jā	jā	jā	jā	jā
SIP	jā, ar neoficiālu saskarnes failu	jā, ar neoficiālu saskarnes failu	jā, ar neoficiālu saskarnes failu	jā, ar neoficiālu saskarnes failu	jā, ar neoficiālu saskarnes failu
Boost Python	nē	nē	nē	nē	nē
PyBind11	jā	jā	jā	jā	jā
Shiboken	jā	jā	jā	nē	jā
PyBindGen	jā	jā	jā	jā	nē

SECINĀJUMI

Darba sākumā tika izpētīti Python paplašinājumu rakstīšanas pamati un kā caur tiem var izveidot saskarnes C/C++ bibliotēkām. Tas ir sarežģīts process, kur viegli radīt kļūdas. Ir labi jāzina C/C++, Python/C API, ir jārūpējas par objektu referenču skaitu, datu tipu pārveidošanu, Python versiju atbalstu.

Darbā ir izpētīti visi aktuālie rīki, kuri palīdz izveidot C/C++ bibliotēku saskarnes Python valodai. Katram rīkam tika izveidots un palaists piemērs ar reālu C un C++ bibliotēku. Gandrīz katram rīkam izdevās strādājošs piemērs. Tiem, kuri nestrādā, ir paskaidrots, kāpēc tas neizdevās. Katrs rīks tika analizēts, balstoties uz rīku dokumentācijām un izveidotiem piemēriem. Beigās ir veikta visu rīku salīdzināšana un kopsavilkums. Salīdzinot, galvenais, kas tika ņemts vērā, ir lietošanas vienkāršība, rīka iespējas un saskarnes ātrums.

Pētījumā rezultātā rīki tika sagrupēti – Python bibliotēkas, kuras izsauc C funkcijas pa tiešo (ctypes, cffi), koda ģeneratori (cffi API, SWIG, SIP, PyBindGen, Shiboken), C++ bibliotēkas (boost python, pybind11) un Cython. ctypes un cffi ir vienīgi, kuriem nav vajadzīga saskarnes koda kompilācija. Tika noskaidrots, ka cffi bibliotēkai ir vienkāršākā lietošana nekā ctypes, un arī ātrums bija labāks. Koda ģeneratori priekš C bibliotēkām, vislabāk sevi parādīja cffi API režīmā. Tam ir viskompaktākais un visātrākais saskarnes kods, kā arī vienkāršākā lietošana. C++ bibliotēkām Shiboken un SIP ģeneratori ir labāki. Tiem ir ātrāks un kompaktāks saskarnes kods nekā SWIG. boost python un pybind11 nav koda ģeneratori, bet tikai C++ bibliotēkas. Starp tiem pybind11 ir labāks vairākos parametros. Cython valoda, paplašinājumu rakstīšanai, ir vislabākā, lai kontrolētu saskarnes kodu.

Tālākais pētījums var būt zemā līmeņa saskarnes koda pētīšana vai kāda no rīkiem pilnveidošana. Cits virziens varētu būt citu programmēšanas valodu saskarņu pētīšana.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] Extending Python with C or C++

[https://docs.python.org/3.4/extending/extending.html]

[2] The Module’s Method Table and Initialization Function

[https://docs.python.org/3.4/extending/extending.html#the-module-s-method-table-and-initialization-function]

[3] Defining New Types

[https://docs.python.org/3.4/extending/newtypes.html]

[4] Building C and C++ Extensions

[https://docs.python.org/3.4/extending/building.html#building]

[5] [http://docs.scipy.org/doc/numpy-1.10.1/user/c-info.python-as-glue.html#index-3]

[6] ctypes — A foreign function library for Python

[https://docs.python.org/3.4/library/ctypes.html]

[7] Eli Bendersky, Python FFI with ctypes and cffi
[http://eli.thegreenplace.net/2013/03/09/python-ffi-with-ctypes-and-cffi]

[8] Cython - an overview

[http://docs.cython.org/src/quickstart/overview.html]

[9] Language Basics [http://docs.cython.org/src/reference/language_basics.html]

[10] Using C++ in Cython [http://docs.cython.org/src/userguide/wrapping_CPlusPlus.html]

[11] Building Cython code, http://docs.cython.org/src/quickstart/build.html

[12] EXECUTIVE SUMMARY [http://www.swig.org/exec.html]

[13] SWIG and Python [http://www.swig.org/Doc3.0/Python.html#Python_nn2]

[14] What is SIP? [https://www.riverbankcomputing.com/software/sip/intro]

[15] Using SIP [http://pyqt.sourceforge.net/Docs/sip4/using.html]

[16] Joel de Guzman, David Abrahams, QuickStart
[http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/tutorial/index.html#tutorial.quickstart]

[17] Joel de Guzman, David Abrahams, Exposing Classes [http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/tutorial/tutorial/exposing.html]

[18] Joel de Guzman, David Abrahams, Object Interface [http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/tutorial/tutorial/object.html]

[19] CallPolicy [https://wiki.python.org/moin/boost.python/CallPolicy]

[20] Wenzel Jakob, About this project [http://pybind11.readthedocs.io/en/master/intro.html]

[21] STL containers [http://pybind11.readthedocs.io/en/master/advanced/cast/stl.html]

[22] Functions [http://pybind11.readthedocs.io/en/master/advanced/functions.html]

[23] PyBindGen Tutorial [https://pythonhosted.org/PyBindGen/tutorial.html]

[24] 1. Frequently Asked Questions [http://pyside.github.io/docs/shiboken/faq.html]

[25] 6. User Defined Type Conversion [http://pyside.github.io/docs/shiboken/typeconverters.html]

[26] 7. Code Injection Semantics [http://pyside.github.io/docs/shiboken/codeinjectionsemantics.html]

[27] 9. Object ownership [http://pyside.github.io/docs/shiboken/ownership.html]

Bakalaura darbs „C/C++ bibliotēku saskarņu izveide Python valodai” 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.

Autors: _____ Alberts Glagoļevs

Rekomendēju darbu aizstāvēšanai

Vadītājs: Asoc. prof., Dr. dat. Jānis Zuters _____ 29.05.2017.

Recenzents: Doc., Dr. dat. Viesturs Vēzis

Darbs iesniegts Datorikas fakultātē 29.05.2017.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

_____

Komisijas sekretārs: _____