**МIНIСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**Західноукраїнський національний університет**

**Факультет комп’ютерних інформаційних технологій**

Кафедра комп’ютерної інженерії

**Демчук Юлія Ігорівна**

**«Алгоритм генерації оптимального машинного коду для мов високого рівня/Algorithm for generating optimal machine code for high-level languages»**

спеціальність: 123 - Комп’ютерна інженерія

освітньо-професійна програма - Комп’ютерна інженерія

Кваліфікаційна робота

Виконала студентка групи КІм-21

Ю.І. Демчук

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Науковий керівник:

к.т.н., доц. Ю.М. Батько

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кваліфікаційну роботу допущено

до захисту:

"\_\_\_" \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ р.

Завідувач кафедри

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Л. О. Дубчак

**Тернопіль – 2022**

# Вступ

Актуальність роботи. Оптимізація сама по собі є позитивним словом, до чого прагне кожен у багатьох аспектах життя. І у випадку написання коду все так само.

Оптимізований код – це той, який працює краще, ніж його попередні версії. Оптимізація коду – це мистецтво написання або переписування коду таким чином, щоб він споживав найменшу кількість пам’яті чи дискового простору, мінімізував пропускну здатність мережі або процесорний час програми. Іншими словами, оптимізація означає написання тієї ж програми з використанням коротшого коду. Це буде найбільш корисно для людини, яка буде осягати ваш код і переробляти його. Існує тонка межа між зрозумілістю вашого коду та його оптимізацією. Також потрібно звернути увагу на читабельність вашого коду.

Готуючи алгоритм для програми, робіть це так, щоб програміст-початківець міг його легко зрозуміти. Простий алгоритм – запорука оптимальної програми. Розробляючи алгоритм, уникайте построкової оптимізації, оскільки якщо його замінять на пізнішому етапі, ваші зусилля щодо оптимізації нижчого рівня будуть марними.

Вимоги в ІТ-індустрії дуже динамічні. Вони обов’язково зміняться в майбутньому. Ваш код має впоратися з неминучими майбутніми змінами. Існують неоднозначні думки щодо того, якою мірою повинні перевірити свій код у майбутньому, і це рішення краще залишити за вами та компанією, у якій ви працюєте.

Програма має бути закодована таким чином, щоб її можна було переглядати з урахуванням змін. Оптимізований код має бути далекоглядним. Залишити можливість переробити код краще, ніж передбачити майбутні вимоги, оскільки прогнози не завжди відповідають дійсності, і зміни можна вносити на ходу. Не розробляйте код, який можна просто виконати за допомогою циклу. Використання циклів покращує читабельність вашого коду та допомагає в оптимізації.

У будь-якій мові оптимізація коду завжди є бажаною якістю. Завжди працюйте в порядку зверху вниз. Удосконалений код забезпечує виконання вимог із належним обсягом ресурсів, запобігаючи додатковому навантаженню на процесор або програмну пам’ять. Оптимізація завжди призводить тільки до покращення отриманих результатів, тому задача створення алгоритму детекції людських облич на основі виділення ключових точок є актуальною.

Метою роботи є розробка алгоритму генерації оптимального програмного коду мовою програмування високого рівня.

Для досягнення даної мети ставились наступні завдання:

* провести класифікацію програмних додатків на основі сфер використання;
* проаналізувати існуючі типи мов програмування;
* провести аналітичний огляд інтегрованих середовищ розробки програмних додатків;
* проаналізувати існуючі алгоритми трансляції програмнго коду на мову асемблер;
* розробити алгоритм трансляції програмного коду написаного на мові високого рівня на мову ассемблер;
* реалізувати програмний додаток трансляції програмного коду написаного на мові високого рівня на мову ассемблер, провести його тестування та порівняти з програмами-аналогами.

Об’єкт дослідження – процес розробки програмних додатків.

Предмет дослідження – методи і алгоритми перекодування програмного коду на мову асемблер.

Наукова новизна одержаних результатів визначається наступним чином:

* проведено комплексний аналіз та класифікацію алгоритмів трансляції програмних кодів написаних мовами високого рівня на мову асемблер, що надало можливість підкреслити їх переваги та недоліки, а також розробити власний алгоритм трансляції програмного коду на мову асемблер;
* розроблено алгоритм трансляції програмних кодів мовами високого рівня на мову асемблер, що дозволило отримати оптимізацію програмного коду та підвищити підвищити швидкість роботи кінцевого програмного додатку.

Практична цінність одержаних результатів полягає в тому, що:

* розроблено та проведено теоретичне дослідження програмного додатку трансляції програмного коду мовою програмування високого рівня на мову асемблер, що дозволило в подальшому програмно реалізувати та провести дослідження розроблених алгоритмів;
* реалізовано програмне забезпечення для генерації програмного коду мовою асемблер та з використанням масиву моделей трансляції для різних типів алгоритмів.

Публікації та апробація до випускної кваліфікаційної роботи. За результатами наукових досліджень, проведених у випускній кваліфікаційні роботі, підготовлено тези доповіді «Алгоритм конвертації програмних кодів з оптимізацією» обсягом 1 сторінка на VІ Науково-практичній конференції молодих вчених і студентів «Інтелектуальні комп’ютерні системи та мережі», а також «Алгоритм виділення окремих елементів стрічки програмного коду на основі семантичного аналізу» обсягом 1 сторінка на VІ Науково-практичній конференції молодих вчених і студентів «Інтелектуальні комп’ютерні системи та мережі».

# 1 Інтегровані системи розробки та тестування програмних додатків

## 1.1 Програмні додатки класифікація, структура та сфери застосування

Працюючи за комп’ютером, користувачі стикаються з різними комп’ютерними програмами, які допомагають полегшити наші завдання та підвищити загальну ефективність робочого процесу. Усе, від створення простого документа до перегляду Інтернету на комп’ютері, виконується за допомогою програмного забезпечення.

У цю цифрову чи технологічну еру ми оточені різним програмним забезпеченням, кількість якого з кожним днем стрімко зростає. Незалежно від того, яку операційну систему чи платформу ми використовуємо, ми використовуємо програмне забезпечення та полегшуємо собі життя. Без програмного забезпечення комп’ютер – це не що інше, як набір різних апаратних компонентів.

За визначенням, «програмне забезпечення визначається як набір даних, інструкцій, програм і правил, які наказують комп’ютерній системі чи іншому електронному пристрою працювати ефективно, виконувати та виконувати певні завдання. »

Іншими словами, програмне забезпечення – це загальний термін, який стосується будь-якої програми, програми або сценарію, що виконується на комп’ютерних пристроях, щоб допомогти їм виконувати певні завдання або обробляти їх. Вважається, що це змінна частина комп’ютерної системи, тоді як іншою основною частиною є апаратне забезпечення, яке є незмінною частиною.

Комп'ютерне програмне забезпечення - це не що інше, як окремий термін для комп'ютерної програми, який означає певний набір інструкцій. Інструкції написані різними мовами, зрозумілими комп’ютеру. Інструкції (зазвичай звані кодами) допомагають давати інструкції комп’ютеру для роботи та подавати дані для виконання роботи та задоволення потреб користувачів.

Основна логіка всього програмного забезпечення полягає в тому, щоб приймати вхідні дані, обробляти їх і надавати вихідні дані, тобто Введення > Процес > Вихід. Наприклад, можна натиснути кнопку Друк на комп’ютері. Комп’ютер приймає документ як вхідні дані, а потім просить користувачів налаштувати певні параметри. Після цього заданий набір правил обробляється і виводиться на принтер у вигляді друкованого документа на папері.

Існують різні типи програмного забезпечення на основі їх класифікації. Однак, якщо ми загально класифікуємо типи програмного забезпечення в комп’ютері, виділяємо два основні типи, які перераховані нижче:

* Системне програмне забезпечення.
* Прикладне програмне забезпечення.

Системне програмне забезпечення зазвичай визначається як набір різноманітних програм або інструкцій, які допомагають контролювати або керувати апаратними пристроями, підключеними до комп’ютера та іншими важливими ресурсами системи. Зокрема, це допомагає підвищити ефективність роботи комп’ютера, забезпечує більш високу продуктивність у безпечний спосіб. Крім того, системне програмне забезпечення також створює платформу для запуску прикладного програмного забезпечення.

Системне програмне забезпечення надає користувачам різноманітні параметри для взаємодії з основними параметрами та надання команд комп’ютеру за допомогою графічного інтерфейсу користувача. Користувачі можуть взаємодіяти лише із загальними параметрами системи та службами. Крім того, основні служби та програми системного програмного забезпечення автоматично працюють у фоновому режимі для керування системними процесами та продуктивністю.

Нижче наведено основні характеристики або особливості системного програмного забезпечення:

* тісніше пов’язане з компонентами системи чи обладнання, а не з користувачами.
* є набагато швидшим у продуктивності та обробці.
* зазвичай пишеться або розробляється мовою низького рівня, щоб воно могло взаємодіяти з машинами з максимально можливою швидкістю.
* мало взаємодіє з користувачами порівняно з прикладним програмним забезпеченням. Зазвичай вони працюють у фоновому режимі.

Порівняно важко маніпулювати системним програмним забезпеченням, оскільки воно складне для розуміння або розробки.

Проаналізуємо різні типи системного програмного забезпечення та прикладами, які допоможуть зрозуміти трохи більше про системне програмне забезпечення (рисунок 1.1).



Рисунок 1.1 – Приклади системного програмного забезпечення

Операційна система: операційна система, яку зазвичай називають ОС, є одним із основних типів або прикладів системного програмного забезпечення. Він складається з набору програмного забезпечення, призначеного для обробки комп’ютерних ресурсів і послуг, і забезпечує платформу для запуску інших програм/програмного забезпечення.

Хоча кожна операційна система працює на основі програмного коду та інструкцій, більшість операційних систем включають GUI (графічний інтерфейс користувача), який допомагає користувачам легко взаємодіяти з ОС без безпосереднього використання коду.

Кожен електронний пристрій, будь то настільний комп’ютер, ноутбук або мобільний телефон, повинен мати встановлену операційну систему, щоб функціонувати та забезпечувати базові функції. Операційна система — це перша програма, яка запускається після завантаження комп’ютерної системи. Операційна система контролює весь комп’ютер і ефективно керує всіма службами та ресурсами, такими як ЦП, пам’ять, пристрої зберігання (HDD або SSD), принтери тощо. Деякі з популярних прикладів операційних систем: MS-Windows, Android, macOS, iOS, Linux, Ubuntu, Unix, CentOS.

Драйвери пристроїв: драйвер пристроїв — це спеціалізоване комп’ютерне програмне забезпечення, призначене в основному для керування певним апаратним забезпеченням комп’ютерної системи. Комп’ютери використовують різні апаратні пристрої, і для належної роботи кожного пристрою потрібен драйвер.

Найпоширенішими апаратними пристроями, для яких потрібні драйвери, є дисплеї, графічні карти, звукові карти, жорсткі диски, принтери, миші та клавіатури. Користувачам може знадобитися вручну встановити драйвери для деяких апаратних пристроїв, а для інших операційна система автоматично встановить підтримуваний драйвер. Існує два різних типи драйверів пристроїв, а саме драйвери пристроїв ядра та драйвери пристроїв користувача. Деякі з популярних прикладів драйверів пристроїв: Драйвер BIOS, Драйвер материнської плати, Драйвер дисплея тощо.

Мікропрограмне забезпечення: мікропрограмне забезпечення відоме як постійне програмне забезпечення або інструкції, що зберігаються в пам’яті материнської плати або ПЗУ (постійній пам’яті) комп’ютерної системи. Однак насправді він не зберігається постійно, оскільки більшість сучасних комп’ютерних систем дозволяють користувачам оновлювати мікропрограму за допомогою програми оновлення мікропрограми.

Програмне забезпечення мікропрограми, як і інше програмне забезпечення, має набір інструкцій. На відміну від іншого програмного забезпечення, основною функцією мікропрограми є перевірка на наявність помилок і забезпечення належної роботи всіх необхідних апаратних пристроїв, щойно користувач увімкне комп’ютер. Якщо все працює, прошивка виводить систему з режиму сну та передає керування операційній системі.

Перекладачі мов програмування: Зокрема, системне програмне забезпечення допомагає перетворювати мовний код високого рівня (код мови, який читається людиною) у простий код машинного рівня і навпаки. Це перетворення зазвичай виконується за допомогою трансляторів мов програмування або процесорів.

Транслятори мов програмування допомагають перетворювати комп’ютерні програми, написані мовами програмування високого рівня, такими як Java, C, C++, Python тощо (їх також називають вихідним кодом), на інструкції, які можуть інтерпретуватися машинами як код об’єктного рівня або машинний код.

Незважаючи на спрощення коду, перекладачі мов програмування також допомагають виконувати наступні завдання: призначення сховища даних, отримання діагностичних звітів, отримання вихідного коду та інших деталей, пов’язаних з програмою, виявлення системних помилок під час виконання або виправлення тощо.

Утиліта: допоміжний засіб або програмне забезпечення в основному розроблено для аналізу, оптимізації, налаштування та обслуговування комп’ютерної системи або її ресурсів. Зазвичай це допомагає безперебійно підтримувати комп’ютерну інфраструктуру. Службове програмне забезпечення відстежує роботу операційної системи, а потім надає пропозиції або автоматично вносить зміни для покращення функціонування та загальної продуктивності комп’ютера.

Службове програмне забезпечення включає різні категорії комп’ютерного програмного забезпечення, такі як антивірусне програмне забезпечення, засоби очищення диска та інструменти керування, оптимізатори продуктивності, дефрагментатори, засоби стиснення тощо.

Прикладне програмне забезпечення відноситься до набору програм і інструкцій, які допомагають виконувати певні завдання в комп’ютерній системі. В основному він призначений для задоволення певних вимог конкретного середовища. Прикладне програмне забезпечення можна завантажити та встановити вручну на комп’ютерній системі, і це програмне забезпечення не має нічого спільного з основними функціями системи.

Прикладне програмне забезпечення зазвичай працює у інтерфейсі, що робить його більш доступним і зручним для користувачів. Важливо зазначити, що прикладне програмне забезпечення не може працювати самостійно; їм потрібна платформа, встановлена системним програмним забезпеченням. Наприклад, для роботи будь-якого програмного забезпечення веб-браузера, як-от Google Chrome, потрібна операційна система. На відміну від системного програмного забезпечення, прикладне програмне забезпечення не є обов’язковим для системи. Однак вони можуть допомогти виконувати різноманітні завдання на комп’ютері.

Характеристики прикладного програмного забезпечення:

* в основному розробляється або програмується для виконання конкретних завдань на комп’ютерах, таких як редагування зображень, обробка текстів, ігри тощо.
* переважно має більший розмір і, отже, займає більше пам’яті або місця на машинах/системах.
* зазвичай пишеться або розробляється на мовах високого рівня.
* більше взаємодіє з користувачами, допомагаючи їм легко використовувати програмне забезпечення.
* дещо легше проектувати та розробляти, ніж системне програмне забезпечення.

Типи прикладного програмного забезпечення наведено на рисуноку 1.2.



Рисунок 1.2 – Приклади прикладного програмного забезпечення

Текстові процесори: текстові процесори або програмне забезпечення для обробки текстів призначені в основному для створення документації на комп’ютерах або інших електронних пристроях. Крім того, програмне забезпечення текстового процесора також допомагає керувати, форматувати та друкувати документи. Деякі з популярних прикладів текстових процесорів: MS Word, Документи Google, OpenOffice Writer тощо.

Програмне забезпечення для баз даних: програмне забезпечення для баз даних – це корисне комп’ютерне програмне забезпечення, яке в основному допомагає створювати та керувати базами даних. Його також називають СУБД (Програмне забезпечення для керування базами даних). Програмне забезпечення баз даних відіграє важливу роль при організації даних на комп’ютері чи сервері. Приклади програмного забезпечення для баз даних: MS Access, MySQL, dBase.

Мультимедійне програмне забезпечення: як випливає з назви, мультимедійне програмне забезпечення в основному використовується для виконання завдань, пов’язаних із мультимедійними файлами, такими як аудіо- чи відеофайли. Це допомагає відтворювати, створювати або змінювати мультимедійні файли. Мультимедійне програмне забезпечення відіграє важливу роль у редагуванні зображень, відео, анімації та задачах, пов’язаних із графікою.

До мультимедійного програмного забезпечення належать: Windows Media Player, Windows Movie Maker, Adobe Photoshop тощо.

Графічне програмне забезпечення: графічне програмне забезпечення розроблено суворо з урахуванням завдань, пов’язаних із графікою. Це насамперед допомагає користувачам створювати логотипи, редагувати зображення та вносити інші зміни у візуальні дані. Графічні програми містять багато вбудованих інструментів і функцій, які полегшують редагування та ілюстрації.

Веб-браузери: веб-браузери в основному використовуються для пошуку в Інтернеті або перегляду веб-сторінок. Це спеціалізоване комп’ютерне програмне забезпечення, яке допомагає користувачам знаходити певні веб-адреси для отримання даних у мережі. Більшість комп’ютерів і електронних пристроїв постачаються з попередньо встановленим веб-браузером, який надається виробником. Однак користувачі завжди можуть встановити бажані сторонні веб-браузери з App Store за допомогою Інтернету на своїх пристроях.

Навчальне та довідкове програмне забезпечення: це комп’ютерне програмне забезпечення, призначене в першу чергу для того, щоб допомогти користувачам вивчити конкретні теми з конкретних предметів. В Інтернеті доступний широкий вибір освітнього програмного забезпечення, і користувачі можуть полегшити навчання за допомогою нього. Цей тип програмного забезпечення також відомий як академічне програмне забезпечення. Навчальне програмне забезпечення може бути безкоштовним або платним.

Спеціалізоване програмне забезпечення або програмне забезпечення спеціального призначення. Спеціалізоване програмне забезпечення або програмне забезпечення спеціального призначення – це спеціалізоване комп’ютерне програмне забезпечення, розроблене або запрограмоване переважно для виконання конкретних завдань/функцій або призначене для конкретних організацій. Усе таке програмне забезпечення, яке виконує певні цілі, відноситься до прикладного програмного забезпечення. Серед популярних програм спеціального призначення є система бронювання залізничних перевезень, система керування рахунками-фактурами, система бронювання авіакомпаній тощо.

Крім згаданої вище класифікації, існує ще одна популярна класифікація комп'ютерного програмного забезпечення. Ця класифікація, як правило, базується на доступності програмного забезпечення та можливості спільного використання, яке включає наступне:

* Безкоштовне програмне забезпечення;
* Умовно-безкоштовна програма;
* Відкритий код;
* Закритий код;
* Проміжне ПЗ.

## 1.2 Мови програмування та їх класифікація

Оскільки комп’ютер, автоматизація та робототехніка все більше залучаються до нашого повсякденного життя, програмування стає дуже необхідним для контролю над усіма ними. Щоб контролювати все це систем і машин і отримати бажаний результат за допомогою кваліфікованих мов програмування необхідно. Однак область мови програмування стає настільки широкою, як тільки це підпадатиме під одну з двох категорій мов програмування (тобто низькорівнева мова та мова високого рівня). На початку розвитку комп’ютерів мовний дизайн був сильно залежить від рішення використовувати компіляцію чи інтерпретацію як спосіб виконання.

Залежно від таких інструментів, як компіляція та інтерпретація, щоб отримати наш письмовий код у форму, яку може виконати комп’ютер. Код може бути виконаний у власному вигляді операційна система після її перетворення на машинний код (через компіляцію) або може бути оцінюється рядок за рядком за допомогою іншої програми, яка замість цього обробляє виконання коду самої операційної системи (через інтерпретацію).

Мови програмування в основному класифікуються на дві основні категорії – мова низького рівня та мова високого рівня. Кожна мова програмування належить до однієї з ці категорії та підкатегорії.

Мови низького рівня використовуються для написання програм, які відносяться до конкретної архітектури і апаратне забезпечення певного типу комп’ютера. Вони ближчі до рідної мови комп’ютер (двійковий), що ускладнює їх розуміння програмістами. Програми написані мовами низького рівня є швидкими та ефективними для пам’яті. Однак це викликає великі складності для програмістів для написання, налагодження та підтримки програм низького рівня. Вони в основному використовуються для розробки операційних систем, драйверів пристроїв, баз даних і програм, які вимагають прямого апаратний доступ. Мови низького рівня додатково класифікуються ще на дві категорії:

* Машинна мова.
* Мова асемблера.

Машинна мова: машинна мова є найближчою мовою до апаратного забезпечення. Це складається з набору команд, які виконуються безпосередньо комп’ютером. Ці інструкції – це послідовність двійкових бітів. Кожна інструкція виконує дуже специфічно і маленьке завдання. Інструкції, написані машинною мовою, залежать від машини залежить від комп’ютера.

Мова асемблера: мова асемблера є вдосконаленням машинної мови.

Подібно до машинної мови, мова асемблера також безпосередньо взаємодіє з обладнання. Замість використання необробленої двійкової послідовності для представлення набору інструкцій, мова асемблера використовує мнемотехніку. Мова асемблера використовує спеціальну програму називається асемблером. Асемблер перетворює мнемоніку в певний машинний код.

Переваги мов низького рівня:

* Програми, розроблені з використанням мов низького рівня, швидкі та ефективно використовують пам’ять.
* Програмісти можуть краще використовувати процесор і пам’ять, використовуючи низький рівень мову.
* Немає потреби в будь-якому компіляторі або інтерпретаторі для перекладу джерела на машину код. Таким чином, скорочується час компіляції та інтерпретації.
* Мови низького рівня забезпечують пряме маніпулювання комп’ютерними регістрами та пам’яттю.
* Вона може безпосередньо спілкуватися з апаратними пристроями.

Недоліки мов низького рівня:

* Програми, розроблені з використанням мов низького рівня, залежать від машини, а не є такими портативний.
* Важко розробляти, налагоджувати та підтримувати.
* Програми низького рівня більш схильні до помилок.
* Низькорівневе програмування зазвичай призводить до низької продуктивності програмування.
* Програміст повинен мати додаткові знання про архітектуру комп’ютера конкретної машини для програмування на мові низького рівня.

Мови високого рівня схожі на людську мову. мовами високого рівня є зручний для програмістів, простий у кодуванні, налагодженні та обслуговуванні. це забезпечує вищий рівень абстракція від машинної мови. Вони не взаємодіють безпосередньо з обладнанням.

Швидше, вони більше зосереджені на складних арифметичних операціях, оптимальній ефективності програми і легкість у кодуванні. Програми на мові високого рівня написані англійською мовою оператори (такі як Python, Java, C++ тощо). Вимагають програми високого рівня компілятори/інтерпретатори для перекладу вихідного коду на машинну мову. Програміст можемо скомпілювати вихідний код, написаний мовою високого рівня, на декілька машинних мов. Таким чином, вони є машинно незалежною мовою. Мови високого рівня згруповані в дві категорії на основі моделі виконання – скомпільовані або інтерпретовані мови.

Також можемо класифікувати мову високого рівня на кілька інших категорій на основі парадигма програмування.

Структуроване програмування (іноді відоме як модульне програмування) – це парадигма програмування, спрямована на підвищення чіткості, якості та часу розробки комп’ютерну програму, широко використовуючи конструкції структурованого потоку керування вибір (if/then/else) і повторення (while і for), блокові структури та підпрограми.

Отже, це робить його більш ефективним і легшим для розуміння та модифікації. Структурований програмування часто використовує низхідну модель проектування, за якою розробники планують загальну структуру програми на окремі підрозділи. Зауважте, це можна зробити структурованим програмування будь-якою мовою програмування.

Процедурне програмування є парадигмою програмування, похідною від структурованого програмування, засноване на концепції виклику процедури. Процедури, також відомі як підпрограми, підпрограми або функції, просто містять серію обчислювальних кроків здійснюється. Будь-яка дана процедура може бути викликана в будь-який момент під час виконання програми, в тому числі іншими процедурами або самостійно.

Об’єктно-орієнтоване програмування – це парадигма програмування, заснована на концепції "об'єкти", які можуть містити дані у формі полів, часто відомих як атрибути; і код у формі процедур, часто відомих як методи. Особливістю об'єктів є те, що процедури об'єкта можуть отримувати доступ і часто змінювати поля даних об'єкта, з яким вони пов'язані. Таким чином, програмісти визначають не тільки тип даних структури даних, але також типи операцій (функцій), які можна застосувати до структури даних. Таким чином, структура даних стає об'єктом, який включає як дані, так і функції. На додачу, програмісти можуть створювати зв’язки між одним об’єктом та іншим.

Переваги мови високого рівня:

* Мови високого рівня зручні для програміста. Їх легко писати, налагоджувати та підтримувати.
* Це забезпечує вищий рівень абстракції від машинних мов.
* Це машинно незалежна мова.
* Легко навчатися.
* Менш схильний до помилок, їх легко знайти та виправити.
* Високорівневе програмування забезпечує кращу продуктивність програмування.

Недоліки мови високого рівня

* Для перекладу вихідного коду в машинний код потрібен додатковий час перекладу.
* Програми високого рівня порівняно повільніші за програми низького рівня.
* Порівняно з низькорівневими програмами, вони, як правило, менш ефективні для пам’яті.
* Неможливо зв’язатися безпосередньо з обладнанням.

Інтерпретовані та скомпільовані мови. Під час написання програми розробник може вирішити, чи використовувати скомпільований мова або інтерпретована мова для вихідного коду програми. Обидва види мов мають свої сильні та слабкі сторони. Як правило, рішення використовувати мову перекладу базується на часових обмеженнях на розробку або для полегшення майбутніх змін у програмі.

Терміни інтерпретована мова та скомпільована мова не є чітко визначеними, оскільки згідно теорії, будь-яка мова програмування може бути інтерпретована або скомпільована. У сучасному реалізація мови програмування стає все більш популярною для платформи обидва варіанти.

Інтерпретована мова – це тип мови програмування, для якого більшість її реалізації виконують інструкції прямо і вільно, без попередньої компіляції програму в команди машинної мови. Інтерпретатор виконує програму безпосередньо, перекладаючи кожен оператор у послідовність однієї або кількох підпрограм, а потім на іншу мову (часто машинний код). Приклади деяких поширених інтерпретацій мови включають Ruby, JavaScript і Python.

Інтерпретатори приймають як вхідні дані абстрактне представлення вхідної програми (у джерелі мову) і оцінити його щодо додаткових вхідних даних. Вихід інтерпретатор – це результат, створений програмою, що виконується.

Переваги інтерпретованих мов:

* простий у вивченні та використанні;
* мінімальні знання або досвід програмування;
* дозволяє виконувати складні завдання за відносно невелику кількість кроків;
* дозволяє просте створення та редагування в різноманітних текстових редакторах;
* дозволяє додавати динамічні та інтерактивні дії до веб-сторінок;
* редагування та виконання коду відбувається швидко.

Недоліки мови перекладу

* зазвичай бігають досить повільно;
* обмежений доступ до коду оптимізації низького рівня та швидкості;
* обмежені команди для виконання детальних операцій над графікою.

Компілятор – це програма, яка перетворює зрозумілий людині код у зрозумілий для комп'ютера інструкції – процес, який відбувається лише один раз протягом життя цього коду. Спочатку це займає трохи більше часу, тому що компілятор повинен змінити порядок, оптимізувати або «компілювати» об’єкт код спочатку. Прикладами чисто скомпільованих мов є C, C++ і Go.

Компілятор перетворює та оптимізує програму, написану на вищому рівні абстракції (вихідна мова) у програму, написану мовою нижчого рівня абстракції (цільова мову). Цільова програма написана таким чином, щоб її могла виконувати машина чиї інструкції є дрібнішими та адаптованими до його внутрішніх структур даних.

Переваги компільованої мови:

* швидке виконання;
* оптимізовано для цільового обладнання;
* Недоліки компільованої мови:
* потрібен компілятор;
* редагування та розгортання коду набагато повільніше, ніж інтерпретатори.

На рисуноку 1.3 наведено порівняння мов високого рівня та низького.



Рисунок 1.3 – Порівняння мов високого та низького рівнів

## 1.3 Інтегровані середовища розробки

Інтегроване середовище розробки або IDE – це програмна платформа, яка полегшує створення інших програмних програм, надаючи простір для написання, компіляції та налагодження коду, іноді з додатковими інструментами, які зменшують зусилля розробки.

До появи інтегрованих середовищ розробки на початку 90-х розробникам програмного забезпечення доводилося писати свій код у текстовому редакторі, як-от Блокнот, а потім запускати його в окремому компіляторі. Потім їм доведеться помітити всі помилки, повернутися до текстового редактора та внести зміни в код. Це робило розробку програмного забезпечення надзвичайно громіздким процесом, оскільки кодування, компіляція та налагодження відбувалися в незв’язаних робочих процесах.

Поява IDE наприкінці 1980-х все змінила. Softlab Munich запустила перше в світі інтегроване середовище розробки Maestro I, яке встановили тисячі програмістів по всьому світу. Згодом Microsoft придумала власну IDE Visual Basic (VB), яка стала надзвичайно популярною. З Visual Basic IDE увійшли в основний технічний лексикон і стали невід’ємною частиною розробки та життєвого циклу DevOps.

Щоб повністю відповісти на запитання «що таке інтегроване середовище розробки» проведемо аналіз критичних компоненти IDE, які забезпечують її роботу (рисунок 1.4).

Внутрішні компоненти IDE

Інструменти налагодження

Автоматизація збірки

Переглядач класів і об’єктів

Редактор коду

Механізм компіляції

Рисунок 1.4 – Внутрішні компоненти IDE

Редактор коду: це основний компонент IDE, де програміст пише програмний код. Зовні він виглядає схожим на текстовий редактор, але під ним є кілька функцій, які спрощують процес написання коду. Наприклад, компонент редактора IDE може передбачити, що ви збираєтеся написати, і автоматично завершити команду.

Механізм компіляції: IDE мають вбудований компілятор, який запускає код у тому самому середовищі, щоб перевірити, чи він працює. Як правило, результати компіляції відкриваються в окремому інтерфейсі на одній платформі IDE, і можна легко перемикатися між редактором коду та компілятором.

Інструменти налагодження: IDE мають основні інструменти тестування, які виявляють помилки програми та рівень вихідного коду. Важливо, що він може не бачити логічних недоліків, але може позначати помилки написання коду, такі як неправильні команди, відсутні змінні, проблеми з синтаксисом тощо. Інтегроване середовище розробки підсвічує точне місце помилки, щоб програміст міг налагодити легко кодувати.

Переглядач класів і об’єктів: під час використання об’єктно-орієнтованого програмування (ООП) IDE може містити інструмент для перевірки об’єктів і об’єктно-орієнтованих класів у програмі. Він може навіть візуалізувати ієрархію класів і дозволити розробникам ефективніше повторно використовувати об’єкти.

Автоматизація збірки: автоматизація збірки – це процес підготовки програмного коду до виконання. Хоча можна мати окреме середовище збирання, деякі IDE надають готові до використання інструменти автоматизації збирання. Це допомагає запакувати код для реалізації після завершення написання.

Використання IDE є відносно простим процесом. Це не вимагає від розробника спеціальних навичок, крім навичок кодування, знання систем розробки та базового знайомства з функціями платформи IDE. Оскільки кожне інтегроване середовище розробки дещо відрізняється (крім пропозиції п’яти основних компонентів), слід знати про найкращі програми в цьому сегменті.

IDE існують багато років. З часом IDE перетворилися з простої платформи для налагодження та тестування на взаємопов’язаний програмний пакет, який функціонує як розширення розробника. «Інтегрований» компонент відрізняє найкраще програмне забезпечення IDE від стандартних інструментів редагування коду. Нижче наведено деякі з найкращих доступних на даний момент рішень для кодування, які є простими та багатими функціями:

Microsoft Visual Studio – це інтегроване середовище розробки (IDE) для створення програм із графічним інтерфейсом користувача та консолями. Він також підтримує веб-сайти, веб-програми, онлайн-сервіси та програми Windows Forms і WPF. У Visual Studio є редактор коду, який підтримує IntelliSense (компонент для завершення коду) і рефакторинг коду. Додаткові встановлені інструменти включають інтегрований налагоджувач, профайлер коду, дизайнер GUI, веб-розробник, дизайнер класів і дизайнер схеми бази даних.

Eclipse є одним з найвідоміших інтегрованих середовищ розробки (IDE) для Java. Це настільна програма, яка працює на кількох платформах. Інтерфейс користувача Eclipse є однією з найпривабливіших функцій. Він також підтримує можливості перетягування. Можна також виконати статичний аналіз вашого коду. Крім того, він підтримує налагодження та профілювання.

Netbeans – це безкоштовна IDE із відкритим кодом. Ідеально підходить для налаштування існуючих проектів або розробки з нуля, NetBeans має інтуїтивно зрозумілий інтерфейс перетягування та безліч корисних шаблонів проектів. Зазвичай він використовується для розробки програм Java, хоча ви можете придбати пакети, які підтримують інші мови.

Amethyst 2 – це інтегроване середовище розробки (IDE), створене Г’ю Коллінгборном і Дермотом Хоганом у 2006 році. Amethyst 2 доступний у двох версіях: Amethyst Ultimate, платна версія, і Amethyst Personal, безкоштовна версія. Amethyst Ultimate пропонує кілька інструментів, таких як Amethyst Designer, налагоджувач, рефакторинг коду тощо.

Android Studio, офіційна IDE Android, пропонує найпростіші інструменти для розробки програм для всіх пристроїв Android. Поєднання настроюваної системи збірки та системи швидкого збирання/розгортання дозволяє розробникам зосередитися на створенні багатофункціональних високоякісних програм і швидкої доставки. Крім того, він пропонує абсолютно безкоштовну IDE.

BlueJ призначений для адміністрування університетських курсів і працює на Mac OS X, Windows, Linux та інших системах на основі Java. Це дозволяє новачкам почати відразу, не будучи перевантаженими варіантами. BlueJ попередньо встановлено на образ Raspberry Pi як частину дистрибутива Raspbian. Він найбільше підходить для індивідуальних програмістів або невеликих компаній з розробки програмного забезпечення. Розробники можуть оцінювати свої програми та взаємодіяти з абсолютно новими елементами.

Komodo доступний майже всім програмістам, оскільки він підтримує більшість відомих мов програмування. Спрощений інтерфейс користувача полегшує складне редагування. Komodo є широко використовуваним IDE для мобільних і веб-розробок завдяки перевірці синтаксису та можливостям одноетапного усунення несправностей (або налагодження). Він інтегрується з системами контролю версій CVS, Bazaar, Mercurial, Git, Subversion і Perforce.

C-Free – це безкоштовне інтегроване середовище розробки (IDE) для розробки на C і C++, яке можна використовувати як редактор або як окреме середовище програмування. Можна самостійно модифікувати, розробляти та налагоджувати програми, використовуючи єдиний послідовний процес із вбудованими інструментами та можливостями, які покращують ваші навички. C-Free також компактний, з розміром інсталяції 14 МБ і 80 МБ без упаковки.

IntelliJ IDEA – це IDE на основі Java для написання програм на Java, Kotlin, Groovy та інших мовах програмування на основі JVM. Він був створений JetBrains і пропонується у двох варіантах – офіційному комерційному виданні та виданні для спільноти Apache 2. Версія Ultimate IDE пропонує підключені системи контролю версій, системи керування базами даних, а також інструменти збірки та пакування.

AWS Cloud9 – це хмарна IDE, розміщена на Amazon Web Services (AWS) , яка дозволяє користувачам створювати, виконувати та налагоджувати код, використовуючи лише веб-браузер. Він поставляється з редактором коду, налагоджувачем і терміналом. Cloud9 має попередньо встановлені інструменти для основних мов програмування, таких як JavaScript, Python, PHP та інші. За допомогою Cloud9 ви можете миттєво запропонувати своїм колегам своє середовище програмування, дозволяючи об’єднувати програми та контролювати дії один одного в реальному часі. Він також забезпечує плавну розробку безсерверних програм.

Selenium IDE відноситься до інтегрованого середовища розробки Selenium. Це розширення Firefox дозволяє користувачам записувати свою діяльність і експортувати її як сценарії на різних комп’ютерних мовах. Функція запису Selenium IDE пропонує інтуїтивно зрозумілий інтерфейс для створення автоматизованих тестів. Функція запису та відтворення робить навчання дуже простим із мінімальним досвідом програмування.

Глобальний ринок IDE на сьогоднішній день бурхливо розвиваєтсья тому доцільно провести класифікацію іннуючих IDE. Найбільш заслуговують на увагу:

Хмарні IDE починають набирати популярність. Функції цих перших в Інтернеті IDE швидко розширюються, і всім великим виробникам, ймовірно, доведеться надавати їх, щоб залишатися конкурентоспроможними. Хмарні IDE дозволяють розробникам отримувати доступ до своєї роботи з будь-якого місця. Nitrous і AWS Cloud9 – це платформи хмарного середовища розробки, які підтримують Ruby, Python, Node.js та інші мови.

Існують спеціалізовані IDE для розробників, які працюють однією мовою. CodeLite і C-Free для C/C++, Jikes і Jcreator для Java, Idle для Python і RubyMine для Ruby/Rails – інші приклади. Однак нещодавно відбувся зсув у бік багатомовних IDE через їхню універсальність. Іноді за допомогою плагінів розробники також можуть отримати підтримку додаткових мов. Flycheck, наприклад, є плагіном перевірки синтаксису для GNU Emacs 24, який підтримує 39 різних мов.

Багатомовні IDE підтримують багато мов програмування. Eclipse підтримує такі мови, як Perl, C, C++, Ruby, Python , Java та PHP, серед інших. Це безкоштовний редактор із відкритим кодом для кількох фреймворків для розробки програмного забезпечення. Він починався як середовище програмування Java, але згодом розширився за рахунок додавання плагінів. Це інтегроване середовище розробки управляється та контролюється під егідою Консорціуму Eclipse.org.

Середовища розробки додатків HTML є одними з найпоширеніших доступних IDE. Наприклад, DreamWeaver, HomeSite і FrontPage автоматизують кілька процесів, пов’язаних із розробкою веб-сайтів. Вони мають вирішальне значення для створення веб- програмного забезпечення як послуги (SaaS).

Існують спеціалізовані IDE для мобільної розробки, такі як Titanium Mobile від Appcelerator і PhoneGap. Численні IDE, особливо багатомовні, містять плагіни мобільної розробки. Eclipse, наприклад, має подібні функції. Xcode також підтримує мови програмування Swift і Objective-C, а також Cocoa Touch і Cocoa API. Це інтегроване середовище розробки (IDE) призначене для створення додатків для Mac і iOS і містить програму моделювання iPad/iPhone і дизайнер GUI.

Розробники використовують різні інструменти на етапах розробки, створення та тестування програмного забезпечення. Текстові редактори, бібліотеки коду, програмне забезпечення для відстеження помилок, компілятори та платформи тестування є одними з найпоширеніших інструментів розробки. Без IDE розробник повинен самостійно вибирати, інсталювати, підключати та керувати цими інструментами.

Інтегроване середовище розробки об’єднує кілька пов’язаних із розробкою технологій в єдину структуру. Коли всі утиліти відображаються на одному робочому місці, розробникам не потрібно витрачати незліченну кількість годин на вивчення того, як використовувати кожну з них окремо. Це особливо корисно для розробників-початківців, які можуть використовувати IDE для виявлення основних процесів та інструментів команди.

## 1.4 Постановка задач дослідження

Під час проведених досліджень в даному розділі було наведено проаналізовано та проведено класифікацію програмних додатків та сфер їх використання. Класифікація проводились на основі функціональних особливостей окремих програмних додатків. Досліджено та проведено класифікацію сучасних мов програмування, що дозволило вибрати мову програмування для подальшого дослідження. Досліджено та проведено класифікацію інтегрованих середовищ розробки програмних додатків, виділено їх основні архітектурні, функціональні рішення та структури даних для зберігання, обробки та трансляції програмних кодів з мов високого рівня на мову асемблер.

Для досягнення поставленої мети необхідно розв’язати наступні задачі.

* провести класифікацію програмних додатків на основі сфер використання;
* проаналізувати існуючі типи мов програмування та провести їх класифікцію;
* провести аналітичний огляд інтегрованих середовищ розробки програмних додатків;
* проаналізувати існуючі алгоритми перекодування програмнго коду на мову асемблер;
* розробити алгоритм перекодування програмного коду написаного на мові високого рівня на мову ассемблер;
* реалізувати програмний додаток перекодування програмного коду написаного на мові високого рівня на мову ассемблер, провести його тестування та порівняти з програмами-аналогами.

## 1.5 Висновки до розділу

Проведено аналіз та класифікацію програмних додатків на основі їх функціональних можливостей, що дозволило виділити основні внутрішні архітектурні особливості та формати даних при створенні виконуваних файлів.

Проаналізовано та проведено класифікацію мов програмування високго та низького рівнів, що дозволило визначити мосу С++ як основну для подальших досліджень.

Проведено аналіз сучасних інтегрованих середовищ розробки програмних додатків, на основі дослідження принципів проведення трансляції з мов високого рівня на мову асемблер та механізми формування виконуваного файла.

# 2 Алгоритми трансляції кодів з мов високого рівня на мову асемблер

## 2.1 Структура та особливості мови високого рівня С++

C++ – це мова середнього рівня, і, на відміну від інших мов високого рівня, вона має свої плюси та мінуси. Усі розробники C++ мають справу з UNIX і Linux або платформами та апаратними пристроями. Мова C++ допомагає зробити всю вбудовану та об’єктно-орієнтовану функцію мови універсальною, роблячи її унікальною та відмінною від структурованої мови. Тому кожен інженер повинен вивчити цю мову та забруднити руки цим стандартом кодування мови та парадигмами.

Існує потреба в кожній технології, і мови програмування є великим порятунком для будь-якої розробки програмного забезпечення. C++ – це одна з мов програмування, яка допомагає програмістам і розробникам працювати з багатьма концепціями UNIX або Windows низького чи базового рівня або рівня ядра. Вивчення цього має ще багато переваг, а саме:

Розробники на C++ мають справу з апаратними компонентами та концепціями програмування системного рівня, які мають вирішальне значення для розробки будь-якої програми чи продукту.

Вони можуть отримати розуміння та знання об’єктно-орієнтованої мови, яка допомагає зробити логіку та концепцію будь-якої мови програмування кристально зрозумілою.

Усі поняття керування пам’яттю, створення віртуальних таблиць, таблиць віртуальних покажчиків тощо будуть роз’яснені.

Усі компілятори, завантажувачі, динамічні компонувальники, класи зберігання, типи змінних, області видимості тощо можна легко диференціювати.

Серед особливостей мови С++ слід відмітити наступні:

Розробка прикладного програмного забезпечення передбачає розробку майже всіх типів можливих ОС, як згадувалося раніше, усіх ОС, таких як Windows, MAC OS, OSX і Linux.

Більшість веб-браузерів також написані цією мовою програмування середнього рівня.

Використовується при розробці відомої бази даних, наприклад, MYSQL.

Розробка підпрограм: Крім того, багато інших мов було розроблено з використанням існуючої мови C++, включаючи C#, Java, Python, JavaScript тощо.

Швидка обчислювальна мова: ефективність якої допомагає вченим виконувати роботу швидше.

Розробка ігор і вбудованих систем: ця потужна мова допомагає розробляти ігри та вбудовані системи, такі як високоякісні медичні пристрої та системи САПР.

Серед преваг використання мови С++ є ряд особливостей які значно спрощують процес написання програмних кодів (рисунок 2.1).

Переваги використання мови С++

Простота

Портативність

Об'єктно-орієнтоване програмування

Мова програмування середнього рівня

Динамічний розподіл пам'яті

Багато бібліотек

Чутливий до регістру

Рисунок 2.1 – Переваги використання мови С++

Простота. Мова C++ це фактично одна з найпростіших для вичення мов програмування. ЇЇ також легко зрозуміти та вивчити, оскільки вона походить від мови програмування C. Принципи C++ використовують шлях «знизу вгору» для для роботи в будь якому проекты, який допомагає розбивати великі завдання на частини завдань.

Об'єктно-орієнтоване програмування. Даний підхід є основнийм козирем мови C++, та одним з факторів чому вона стала відомим. У C++ усе розглядається як об’єкти, тому це називається об’єктно-орієнтованим підходом під час розробки програми. Для виконання необхідних завдань використовуються об’єкти класів, а не функції. Вона має різні функції, як-от поліморфізм і успадкування, які спрямовані на об’єднання функцій і даних в єдиний блок, щоб зробити програми безпечними та ефективними.

Портативність. C++ не залежить від платформи. Але вона досить портативна, щоб мати можливість роботи з різними машинами, вносячи деякі або не вносячи жодних змін. Для кожної з операційних системах користувач має можливість запускати той самий код. Програмний код, що був реалізований одного разу, може бути використаний повторно, коли винекне необхідність запуску відповідної функції. Очевидно не можна стверджувати, що С++ повністю незалежний від платформи. Наприклад, прораміст написали код під ОС Linux, проте є необхідність конвертувати його під ОС Windows, в даному випадку код працюватиме коректно, необхідно тільки перекомпілювати його.

Динамічний розподіл пам'яті. Завдяки підтримці покажчиків у мові C++. Розподіл пам'яті можна легко виконати динамічно, а не статично. Можна будь-коли звільнити пам’ять за допомогою функції free().

Рекурсія. Завдяки можливостям багаторазового використання коду ми можемо викликати будь-яку функцію всередині функції, заощаджуючи місце в пам’яті, не записуючи той самий код знову і знову. Багаторазове використання коду є для кожної функції.

C++ є об'єктно-орієнтованою мовою, а не процедурною мовою. Існують особливості C++, які роблять його швидшим і потужнішим за будь-яку іншу мову програмування. Машинне навчання за допомогою бібліотеки C++ також є однією з найкращих переваг використання C++ як мови програмування для процесу розробки.

C++ відома своєю швидкістю, ніж будь-яка інша мова програмування. Паралельний запуск кількох кодів є однією з найкращих засобів підтримки, яка безпосередньо допомагає пришвидшити виконання. Забезпечте найкращу продуктивність, навіть якщо сервер переживає високе навантаження.

На основі апаратного забезпечення. Коли програмне забезпечення тісно пов’язане з апаратним забезпеченням і потрібна деяка низькорівнева підтримка на рівні програмного забезпечення. C++ забезпечує таку підтримку, оскільки він ближче до апаратного забезпечення порівняно з будь-якою іншою мовою програмування.

Функціональність. Такі функції C++, як успадкування, інкапсуляція, абстракція, роблять його корисним для програмістів у процесі розробки програмного забезпечення. Поєднання цих переваг дає видатний і ефективний продукт. Ефективність завжди більша в C++. C++ підтримує перевантаження операторів і дозволяє використовувати визначені користувачем оператори разом із перевантаженням функцій.

Обробка винятків є в C++. Розробникам не потрібно визначати винятки для деяких випадків, оскільки вони мають вбудовані винятки для перехоплення та викидання на екран виведення користувача. Винятком є назва класу для реалізації обробки винятків за допомогою програмування C++. Крім того, ви також можете визначити свій виняток залежно від вимог вашого проекту. Ключові слова try, catch і throw використовуються для обробки винятків у C++. Усі види винятків обробляються обробниками винятків у програмуванні на C++. Змінні можуть бути оголошені будь-де в програмі на C++, але мають бути оголошені перед їх використанням.

Стандартна програма Windows складається з кількох добре відомих елементів які проілюстровані на рисунку 2.2. Дана послідовність кротівк є стандартною та виконується кожного разу коли створюється віконний додаток з графічнийм інтерфейсом. При цьому в програмі передбачені можливості повноцінно реагувати на на всі дії користувачів при цьому зберігаються основні принципи функціонування та обміну повідомленями між операційною системою та програмним додатком.

Точка входу, \_tWinMain , яка надає HINSTANCE програми, аргументи командного рядка та прапорець, що вказує, як показувати головне вікно

Виклик RegisterClass для реєстрації класу головного вікна

Виклик CreateWindow(Ex) для створення головного вікна

Виклик ShowWindow і UpdateWindow для показу головного вікна

Виклик ShowWindow і UpdateWindow для показу головного вікна

Цикл повідомлень для надсилання повідомлень

Процедура для обробки повідомлень головного вікна

Виклик PostQuitMessage після знищення головного вікна

Рисунок 2.2 – Структура програмного додатку на мові С++ під операційну систему Windows

Усі програми Windows мають однакові вимоги. Ці вимоги можуть бути виражені в процедурних викликах Win32, як щойно показано в прикладі. Однак, коли процедурні виклики моделюють базову об’єктну модель, програмісти C++ відчувають потребу загорнути ці виклики в функції-члени. Віконна частина Win32 API (часто називається User32) чітко реалізує базову об’єктну модель, що складається з класів Window (представлених структурою WNDCLASSEX ), об’єктів Window (представлених HWND ) і виклику функцій-членів (представлених викликами до WndProc ). Для програміста на C++, який протидіє розриву між бажаною об’єктною моделлю та моделлю User32, ATL надає невеликий набір віконних класів, як показано на рисунку 2.3.



Рисунок 2.3 – Набір віконних класів для операційної системи Windowsc

Класи, виділені жирним шрифтом, CWindow, CWindowImpl, CWinTraits, CWinTraitsOR, CDialogImpl, CSimpleD ialog і CContainedWindowT є найважливішими. Інші, CWindowImplRoot, CWindowImplBaseT і CDialogImplBaseT, є допоміжними класами для відділення параметризованого коду від інваріантного. Це розділення допомагає зменшити розвантаження коду, пов’язаного з шаблонами, але ці класи не є фундаментальною частиною віконних класів ATL.

Нижче наведено способи використання мови асемблера з кодом C/C++:

Використовуйте окремі модулі змонтованого коду та пов’яжіть їх зі скомпільованими модулями C/C++.

Використовуйте змінні та константи мови асемблера в коді C/C++.

Використовуйте вбудовану мову асемблера, вбудовану безпосередньо в джерело C/C++.

Змініть код мови асемблера, створений компілятором.

Взаємодія C/C++ із функціями мови асемблера є простою, якщо дотримуватись правил виклику і умов викликів , а також умов реєстру. Код C/C++ може отримати доступ до змінних і викликати функції, визначені мовою асемблера, а код асемблера може отримати доступ до змінних C/C++ і викликати функції C/C++.

Підпрограми переривання повинні зберігати всі регістри, які вони використовують. Коли викликаються функції C/C++ з мови асемблера, завантажте призначені регістри з аргументами та надішліть решту аргументів у стек. Функції мають повертати значення правильно відповідно до їхніх декларацій C/C++. Жоден модуль збірки не повинен використовувати розділ .cinit для будь-яких цілей, крім автоініціалізації глобальних змінних. Процедура запуску C/C++ передбачає, що розділ .cinit повністю складається з таблиць ініціалізації. Порушення таблиць шляхом розміщення іншої інформації в .cinit може призвести до непередбачуваних результатів. Компілятор призначає імена посилань усім зовнішнім об’єктам. Таким чином, коли ви пишете код мови асемблера, ви повинні використовувати ті самі імена посилань, які призначає компілятор.

Будь-який об’єкт або функція, оголошена мовою асемблера, доступ до якої здійснюється або викликається з C/C++, має бути оголошена за допомогою директиви .def або .global у модифікаторі мови асемблера. Це оголошує символ зовнішнім і дозволяє компонувальнику вирішувати посилання на нього.

Так само, щоб отримати доступ до функції або об’єкта C/C++ з мови асемблера, оголосите об’єкт C/C++ за допомогою директиви .ref або .global у модулі мови асемблера. Це створює неоголошене зовнішнє посилання, яке компонувальник вирішує. Функції, визначені в C++, які будуть викликані з асемблера, мають бути визначені як extern “C” у файлі C++. Функції, визначені в асемблері, які будуть викликані з C++, мають бути прототиповані як зовнішній «C» у C++.

Приклад нижче ілюструє функцію C++ під назвою main(), яка викликає функцію мови асемблера під назвою asmfunc. Функція asmfunc приймає свій єдиний аргумент, додає його до глобальної змінної C++ під назвою gvar і повертає результат.

extern "C" {

extern int asmF (int num);

int gNam = 0;

}

void main()

{

 int IntegerNum = 5;

 IntegerNum = asmF(IntegerNum); }

Програмний код написаний на мові асемблер

 .global asmF

 .global gNum

asmF:

 LDR r\_1, GvarA

 LDR r\_2, [r\_1, #0]

 ADD r\_0, r\_0, r\_2

 STR r\_0, [r\_1, #0]

 MOV p\_c, lr

GvarA.field Gvar, 32

Іноді програмі C/C++ корисно отримати доступ до змінних або констант, визначених мовою асемблера. Існує кілька методів, якими ви можете скористатися для цього, залежно від того, де і як визначено елемент: змінна, визначена в розділі .bss, змінна, не визначена в розділі .bss, або символ зв’язування.

Доступ до змінних із розділу .bss або розділу з назвою .usect є простим:

* Для визначення змінної використовуйте директиву .bss або .usect.
* Використовуйте директиву .def або .global, щоб зробити визначення зовнішнім.
* Використовуйте відповідну назву посилання мовою асемблера.

У C/C++ оголосите змінну як extern і отримайте до неї звичайний доступ.

.bss var,4,4

.global var

Програма C для доступу до мови асемблера

extern int var;

var = 1;

Для змінних, визначених у C/C++ або мові асемблера, таблиця символів містить адресу значення, яке містить змінна. Коли ви отримуєте доступ до змінної складання за назвою з C/C++, компілятор отримує значення, використовуючи адресу в таблиці символів.

Однак для констант складання таблиця символів містить фактичне значення константи. Компілятор не може визначити, які елементи в таблиці символів є адресами, а які значеннями. Якщо ви отримуєте доступ до константи збірки (або компонувальника) за іменем, компілятор намагається використати значення в таблиці символів як адресу для отримання значення. Щоб запобігти такій поведінці, потрібно використовувати оператор & (адреса), щоб отримати значення (\_symval). Іншими словами, якщо x є константою мови асемблера, його значення в C/C++ дорівнює &x. Додаткові приклади використання \_symval.

Можна використовувати приведення та #defines, щоб полегшити використання цих символів у вашій програмі.

extern int table\_size;

for (I=0; i<TABLE\_SIZE; ++I)

Переклад даного коду на мову асемблер

\_table\_size .set10000

 .global \_table\_size

Оскільки ви посилаєтеся лише на значення символу, яке зберігається в таблиці символів, оголошений тип символу неважливий. У прикладі 5 використовується int. Подібним чином можна посилатися на символи, визначені компонувальником.

У програмі C/C++ можна використовувати оператор asm, щоб вставити один рядок мови асемблера у файл мови асемблера, створений компілятором. Ряд інструкцій asm розміщує послідовні рядки мови асемблера у вихідні дані компілятора без проміжного коду..

Інструкція asm корисна для вставки коментарів у вихідні дані компілятора. Просто починайте рядок коду складання з крапки з комою (;), як показано нижче:

asm(";\*\*\* this is an assembly language comment");

C++ є найефективнішою та найпотужнішою мовою завдяки своїм високорівневим функціональним можливостям. Основні 4 стовпи C++: абстракція, інкапсуляція, успадкування та поліморфізм. Ці стовпи окремо можуть виконувати динамічні завдання та виконувати функції в будь-якому процесі розробки програмного забезпечення, а разом вони є найпотужнішими у світі програмування.

## 2.2 Алгоритми трансляції програмних кодів на мову асемблер

Машинний код є найнижчим рівнем мови програмування, оскільки інструкції виконуються безпосередньо центральним процесором (CPU) комп’ютера. Важливо розуміти, що кожен ЦП або сімейство ЦП має власний набір інструкцій машинного коду.

Інструкції машинного коду – це просто числа, які зберігаються у вигляді двійкового шаблону бітів, наприклад 10101001 01000000.

Інструкція машинного коду зазвичай складається з двох частин:

* оператор (код OP), який є частиною інструкції, яку ЦП виконує (виконує). тобто 10101001 може бути інструкцією для ЦП щодо завантаження з пам’яті.
* операнд (зазвичай адреса пам’яті, звідки зчитуються або записуються дані, залежно від оператора). тобто 01000000 може бути адресою пам’яті 64.

Деякі OP-коди, такі як той, що завершує програму, не вимагають операнда. Написання програм безпосередньо в машинному коді було б виснажливим і схильним до помилок, оскільки всі числові адреси для інструкцій розгалуження та розташування даних потрібно було б обчислювати вручну. Мова асемблера була першим рішенням цієї проблеми, а потім з'явилися все більш складні мови програмування вищого рівня.

Оскільки центральний процесор може виконувати лише інструкції машинного коду, вихідний код усіх інших мов програмування має бути перетворений у машинний код, перш ніж його можна буде виконати. Цей переклад виконується спеціальними програмами, які називаються компіляторами, трансляторами або асемблерами.

Оскільки кожна інструкція машинного коду складається лише з чисел, що зберігаються у вигляді двійкового бітового шаблону, людям дуже важко читати або розробляти програмне забезпечення безпосередньо за допомогою машинного коду. Мова асемблера була оригінальною спробою вирішити цю проблему (за якою пішли дедалі вдосконалені мови програмування високого рівня ).

Мова асемблера – це дуже проста мова програмування, яка використовує мнемоніку (допоміжні засоби пам’яті) для безпосереднього представлення інструкцій машинного коду. Він використовує мітки для представлення адрес пам’яті призначень гілок і даних.

Асемблер перекладає інструкції мови асемблера в інструкції машинного коду. Між інструкціями мови асемблера та інструкціями машинного коду існує майже однозначна відповідність. Це означає, що програма машинного коду, перекладена з мови асемблера, є дуже ефективною і, отже, зазвичай потребує менше пам’яті та працює швидше, ніж програма машинного коду, перекладена з мови високого рівня за допомогою компілятора.

У таблиці 2.1 нижче показано кілька прикладів мнемоніки мови асемблера та інструкцій 16-розрядного машинного коду, створених під час їх складання. Кожна інструкція машинного коду складається з оператора (операційного коду) і операнда.

Таблиця 2.1 – Приклади трансляції з асемблера в машинну мову

|  |  |  |
| --- | --- | --- |
| Мнемоніка мови асемблера | Машинний код | Що воно робить |
| LDA №64 | 10101001 01000000 | Завантажує в акумулятор число 64 |
| ADC №64 | 01101001 01000000 | До того, що є в накопичувачі, додайє число 64 |
| LDA 64 | 10100101 01000000 | Завантажує в накопичувач дані, що зберігаються в пам’яті за адресою 64 |
| ADC 64 | 01101001 01000000 | Додайтє в накопичувач дані, що зберігаються в пам'яті за адресою 64 |

Переваги написання програм на мові асемблера:

Зазвичай він створює швидко запущені програми, оскільки відповідність один до одного означає, що створена програма машинного коду буде дуже ефективною.

Трансляція в машинний код буде дуже швидкою завдяки однозначній відповідності між інструкціями мови асемблера та інструкціями машинного коду.

Легше для розуміння порівняно з машинним кодом завдяки використанню мнемоніки.

Мітки можна використовувати для позначення адрес пам'яті. Без цього додавання або видалення інструкції означає, що всі адреси пам’яті, на які посилається програма, повинні бути перераховані.

Недоліки написання програм на мові асемблера:

Для різних процесорів часто потрібні різні версії мови асемблера, що ускладнює передачу програм між процесорами.

Програми на мові асемблера часто пишуться для певного обладнання, що означає, що вони часто несумісні з іншим обладнанням.

Щоб виконати відносно прості завдання, потрібно багато коду складання, тому складні програми вимагають багато інструкцій з складання, і написання програми займе багато часу.

Код на мові асемблер дуже відрізняється від коду високого рівня. Немає програмних конструкцій, до яких звикли в мовах високого рівня, і чим вище рівень мови, тим дальший він від асемблерного представлення. Найважливіша техніка, яку потрібно розглянути, полягає в тому, як перекладати керуючі конструкції (оператори if, цикли, оператори switch) у форму, яку легко перекласти на асемблер. Проблема така:

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| *if (якась-умова)* *щось;* | *if (якась-умова)* *goto somewhere;* |

Асемблерна форма цього рівняння реалізована інструкцією розгалуження, такою як інструкція JEQ. Ось простий спосіб зробити перетворення по іншом:

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| *if (якась-умова)* *щось;* | *if (!деяка-умова)* *goto x;**щось;**x:* |

Використання міток goto є незвичним для мов високого рівня. Але це суть справи: асемблерний код не структурований. Це просто послідовність коду з операторами розгалуження та мітками – дуже негарно писати та слідувати, але не так складно писати, якщо зробити переклад. Розглянемо трохи складнішу ситуацію:

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| *if (some-condition)* *something**else* *somethingelse;* | *if (якась-умова)* *goto dosomething;**somethingelse**goto somethingdone;**dosomething:**щось**somethingdone:* |

інший спосіб зробити це

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| *if (some-condition)* *something**else* *somethingelse;* | *if (!some-condition)* *goto dosomethingelse;**щось;**goto somethingdone;**dosomethingelse:**щось ще**somethingdone:* |

Розглянемо реальний приклад

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| if (N<0) RESULT=1Else RESULT=-1; | якщо (N<0) goto Nltz;РЕЗУЛЬТАТ=-1;goto Ngez;Nltz: РЕЗУЛЬТАТ=1;Ngez: |

Звичайно, залежно від ситуації, можна почати оптимізувати цей код:

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| РЕЗУЛЬТАТ=-1;if (N<0) РЕЗУЛЬТАТ=1; | РЕЗУЛЬТАТ=1;if (N<0) goto Nltz;РЕЗУЛЬТАТ=-1;Nltz: |

Якщо оператори розгалуження мають відносно просту структури для конвертування, то для трансляції циклів необхідно набагато більше зусиль. Зрозуміло, що цикли завжди йдуть парамельно з масивами, адже саме для зручної обробки масивів найкраще застосовувати циклічні алгоритми. Проаналізуємо алгоритм, яке перетворення програмного коду простого циклу, що буде знаходити суму цілих чисел від 0 до (N-1):

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| for (SUM=0,i=0;i<N;i++) SUM += i; | SUM=0;i=0;цикл: if (i>=N) goto loopdone;SUM = SUM + i;i++;цикл переходу;зациклено: |

Якщо проаналізувати запропонований підхід, то все просто та очевидно, проте данай цикл фактично нічого користного не робить. Для більш складного аналізу модифікуємо аналізований цикл. Тепер необхідно знайти суму не послідовних чисел, а суму усіх елементів масиву.

int myarray[N];

for (SUM=0,i=0;i<=N;i++) SUM += myarray[i];

У асемблерному коді масив – це просто мітка з деяким пробілом (ініціалізованим або неініціалізованим). Мітка є основою масиву, і це адреса, у нашому випадку &*myarray*[0] або в коді C просто *myarray*.

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| елемент = myarray[i] | temp=myarray;температура += i;елемент = \*temp; |

Тепер простим сумуванням деякких значеньобійтись не вийде, тому в середині асемблерного коду необхідно добавити частину коду, яка буде відповідати саме з знаходження наступноге елемента масиву та проводити операцію по знаходженню суми елемента масиву зі значенням змінної в якій зберігаються попредні значення даної змінної:

|  |  |
| --- | --- |
| Код на мові високого рівня | в асемблері |
| int myarray[N];for (SUM=0,i=0;i<N;i++) SUM += myarray[i]; | SUM=0;i=0;loop: if (i>=N) goto loopdone;temp=myarray;temp+=i;SUM = SUM + \*temp;i++;goto loop;loopdone: |

Ситуація ще більш ускладниться якщо в програмному коді ззбільшити кількість масивів які приймають участь в програмних обчисленнях:

int myarray[N], otherarray[N+1];

for (i=0;i<N;i++)

 otherarray[i+1] = myarray[i];

Тепер оптимізація стає більш складнішою. До оптимізації компіляторів програмістам не заохочували писати код із посиланнями на масиви. Замість того їх заохочували писати код за допомогою вказівників. Припустимо, *N* дорівнює 1000. Спираючись на інформацію про те скільки операторів виконується в останньому циклі у формі «асемблера» (6 на ітерацію \* 1000 + 3 = 6003). Але в циклі багато коду, який непотрібний. Для цього перепишемо цикл наступним чином:

int myarray[N]; int \*temp, \*arrayend;

for (SUM=0, temp=myarray, arrayend=&myarray[N] ; temp < arrayend ;temp++)

 SUM += \*temp;

Тоді в асемблерній трансляції даний код буде мати наступний вигляд:

SUM=0;

temp=myarray;

arrayend=&myarray[N];

loop: if (temp >= arrayend)

 goto loopdone;

SUM += \*temp;

temp++;

goto loop;

loopdone:

В даній інтерпретації програмний код вже скорочений до 4\*1000 + 4 інструкції . І це стає більш значущим, якщо додати другий масив.

Оптимізаційні компілятори зробили цей вид кодування спірним. Насправді ручна оптимізація часто заважає компілятору, а не допомагає йому сьогодні, оскільки компілятори очікують простого кодування. Лише в особливих ситуаціях ручна оптимізація може створити кращий код, ніж сьогодні оптимізуючий компілятор. Однак, оскільки робимо переклад на асемблер вручну, подібні перетворення часто полегшують остаточний переклад переставленого C-подібного коду на асемблер.

## 2.3 Алгоритм трансляції програмного коду на мову асемблер

Як було проілюстровано в попередньому розділі на сьогоні процес транслювання програмного коду з мови високого рівня на мову асемблер може мати різні шляхи і при цьому кількість операцій в кінцевому програмному додатку може суттєво відрізнятись. А це, в свою чергу призведе до того, що час виконання програм які виконують одні і тіж завдання може суттєво відрізнятись одна від одної. Окрім того не слід забувати, що кожна команда під час своєї роботи використовує певну кількість електроенергії. Отже написання оптимізованого програмного коду може забезпечити не тільки швидку роботу програмних додатків, але й відчутні фінанстові виграші за рахунок економії електроенергії. Для коректної роботи алгоритму трансляції програмного коду було розроблено ряд моделей, які будуть відповідати тим чи іншим алгоритмічним ситуація. Зобрема були описані ситуації длятрансляції блоків коду з лінійними, циклічними алгоритмами, алгоритмами розгалуження, опрацювання масивів тощо. Окрім того, а масив було занесено всі заразервовані слова мови С++, для коректного аналізу вхідного коду. На основі запропонованих моделей трансляції програмного коду в код на мові асемблер було розроблено алгоритм роботи програмного додатку трансляції програмних кодів. Блок-схема алгоритму наведена на рисунку 2.4.



Рисунок 2.4 – Блок-схема алгоритму трансляції програмного коду з мови високого рівня на мову асемблер

Запропонований алгоритм базується на аналізі введеного програмного коду та масиву розролених моделей трансляції. В його основі закледено принцип покрокової перевірки введеного програмного коду з метою виявлення основних структурних блоків та їх перекодування з одного представлення в інше без втати основної суті алгоритму. Алгоритм містить в собі наступні кроки:

1. Завантаження масиву моделей.
2. Отримання вхідного програмного коду.
3. Послідовна перевірка усіх слів вхідного програмного коду
4. Якщо аналізоване слово є керуючим словом мови програмування С++, то проводиться пошук моделю яка йому відповідає.
5. На основі знайденої відповідної моделі проводиться пошук повного блоку команд які будуть описувати параметри роботи даного програмного блоку.
6. Якщо було знайдено усі частини програмної команди то проводимо процес трансляції на мову асемблер. Якщо параметри не були встановлені, то вважаємо що користувач допустив синтаксичку помилку, видаляємо ключове слово тапереходимо на пунк 3.
7. Якщо всі слова в програмному коді на мові асемблер були перетворені в код на мові асемблер, то виводимо результат та завершуємо роботу інакше прехід на пункт 3.

На початковому етапі проводиться перевірка вхідного слова на співпадіння з зарезервованим словом мови С++, оскільки їх є невелика кількість, то це дозволяє швидко визначити приналежність відповідного слова. Після ідентифікації обраного слова запускається процедура семантичного аналізу наступних після нього слів з метою встановлення усіх параметрів програмної команди. Ці операції відбуваються максимально швидко, оскільки текст програмного коду є структурованим та відповідає чітким правилам написання.

До основних переваг розробленого алгоритму відносяться:

* висока швидкість трансляції програмного коду з однієї мови програмування в іншу;
* технічні парамери робочої станції можуть бути мінімальними, оскільки основні часові затримки залежться від кількості програмного коду якийнеобхідно транслювати;
* може використовуватись для довільої мви високого рівня, при цьому необхідно лизе модифікувати набір зарезервованих слів;

Недоліки:

* алгоритм може коректно транслювати програмний код з однієї мови на іншу, проте кінцевий результат все ж таки залежить від вміння програміста писати хороший програмний код;
* чутливий до ситуацій, коли в програмному коді зустрічаються назви змінних подібні до зарезервованих слів.

Аналіз результатів моделювання запропонованого алгоритму показав, що часові затримки відсютні, трансляція коду відувається з невеликим часовими затримками. Точніст підбору моделей при умові коректно написаного вхідного окду відбувається без похибок.

## 2.4 Висновки до розділу

Проведно аналіз алгогитмі та механізмів трансляції програмного коду мовою високого рівня на мову асемблер на основі використання шаблонів перекодування, що дозволило розробити алгоритм перекодування програмного коду на мову асемблер.

Розроблено алгоритм генерації оптимального машинного коду для мов високого рівня, що дало можливість розробити структуру та програмно реалізувати додаток написання оптимального програмного коду на мові високого рівня.

# 3 Програмний додаток написання оптимального програмного коду

## 3.1 Структура програмного додатку написання коду мовою С++

Найефективнішим методом оптимізації є використання профайлера для виявлення вузьких місць продуктивності. Часто важко вгадати, яка частина вашої програми споживає найбільше ресурсів, і якщо ви базуєте свої зусилля з оптимізації на спекуляціях, а не на реальних даних, ви витратите багато часу на прискорення тих частин вашої програми, які вже були швидкими.

Після того, як визначили вузьке місце, наприклад цикл, який виконується тисячі разів, пам’ятайте, що найкраще – це перепроектувати програму так, щоб їй не потрібно було виконувати цикл тисячі разів. Це більш ефективно, ніж пришвидшення циклу на 10%, але однаково часто, що оптимізуючий компілятор може робити сам по собі. Оптимізація - це просто марна трата часу програміста, якщо будь-яке з цих тверджень вірне:

* частини програми ще не написані;
* програма не повністю перевірена та налагоджена;
* здається, вже досить швидко працює.

Також враховуйте, як програма буде використовуватися. Якщо це програма для створення звітів, яку потрібно запускати лише раз на день, користувач може запустити її перед тим, як піти обідати, і якщо так, то справді немає сенсу завершувати її до того, як вони повернуться. Якщо він викликається з іншої програми, яка навіть повільніша за вашу, користувач знову не помітить різниці. Але якщо він обробляє події відстеження миші для GUI, користувач скаржиться на будь-яку помітну затримку.

Враховуючи, що оптимізація є доцільною, скомпілюйте в режимі повної оптимізації та запустіть програму на вхідних даних «реального світу». Якщо у вас немає доступу до справжніх вхідних даних, обережно вибирайте вхідні дані для тестування: програмісти, як правило, тестують із меншим набором вхідних даних і різними випадками, ніж ті, з якими програма, ймовірно, зіткнеться, коли потрапить у руки користувачів.

Типовий сучасний цифровий комп’ютер має лише 4 типи апаратних інструкцій, які використовуються для програмування додатків:

1) Він може переміщувати число з одного місця в інше: розташування пам’яті та введення/виведення.

2) Він може виконувати прості обчислення, такі як додавання та множення чисел.

3) Він може виконувати умовні стрибки та переходи

4) Він може виконувати виклики підпрограм і повернення.

Мови високого рівня, такі як C, взяли ці 4 інструкції та створили 4 абстракції, які програмісти використовували для кодування всього програмного забезпечення:

1) Змінні та складніші структури даних є корисною абстракцією для чисел, що зберігаються в місцях пам’яті. Це дані в обробці даних.

2) Обчислювальні оператори використовують змінні, які додають, множать інші обчислення, а потім зберігають результат в іншій змінній. Це так звані оператори присвоєння, які виконують усі обчислення в комп’ютерних програмах.

3) Логіка керування потоком використовує інструкцію умовного переходу для реалізації циклів, операторів if-then, do-while і switch-case з використанням змінних. Це дозволяє програмістам створювати логіку керування потоком комп’ютерних програм, і, нарешті

4) Функції (підпрограми) використовують виклик – повертають інструкції та стек, приймають вхідні змінні та створюють вихідні змінні та виконують іншу корисну роботу. Функції – це робочі двигуни комп’ютерних програм.

Для реалізації програмного додатку герерації та оцінки запропонованих алгоритмів трансляції програмного коду з однієї мови програмування на іншу спроектовано внутрішню архітектуру для програмного додатку. За основу було обрано модульний підхід, оскільки використання даного механізму дозволяє більш простіше вносити правки або розширювати функціонільні можливості програмної розробки. Спрощену структуру програми проілюстровано на рисунку 3.1.

Основна керуюча частина

Модуль вводу програмного коду

Модуль роботи з масивом моделей

Модуль асемблерного

кодування

Модуль корекції атрибутів

Модуль виводу даних

Рисунок 3.1 – Узагальнена структура програмного додатку трансляції програмного коду на мову ассемблер

Розроблена структура програмного додатку повинна забезпечити можливість програмно реалізувати запропоновані алгоритми обробки текстової інформації для перекодування прогармних кодів з мов високого рівня на мову асемблер. При проектуванні було використано принципи модульної побудови які в свою чергу володіють рядоом переваг при нарощування або модифікації функціональних можливостей програмних додатків. Окрім того даний тип архітектурного рішення дозволяє проводити реалізацію програми в декілька етапів, що значно спрощує процес реалізації прогармного коду та спрощує механізми підключення зовнішніх програмних бібліотек. При програмній реалізації обрані алгоритми були згруповані у відповідності до задач які вони вирішують. Серед блоків які можна видіти в програмній системі є: керуюча частина, блок корекції параметрів роботи, набір функції для організації вводу програмного коду, засоби організації запитів до бази даних моделей, а також функції які реалізують запропонований алгоритм трансляції програмного коду написаного мовою високого рівня на мову асемблер.

Керування роботою програми відбуває в блоці управління. В ньому зібрані функції для проведення тестування складових програмного додатку. У випадку відсутності або некоректної роботи одного з модулів функції, що були закладені в процесі реалізації зможуть коректно опрацювати помилкову ситуацію та недопустити примусового завершення програмного додатку. Дана особливість особливо є корисною на етапі реалізації, тестування або нарощування програмного додатку новими функціями. На етапі тестування для прискорення процесу аналізу роботоздатності програмного коду при умові відсутності деяких функціональних блоків, замість результатів їхньої роботи будуть підставлені тестові набори, що дозволить проводити тестування не тільки цілої програмної системи, але і її окремих модулів. Окрім того, для підвищення зручності роботи користувачів з програмою в даному блоці було реалізовано систему вспливаючих повідомлень, які можуть показати поточний стан або проінформувати про винекнення критичних/непередбачуваних помилкових ситуацій. Для старту роботи функцій даного блоку, їй на вхід необхідено передати вектор технічних параметрів. В результаті роботи буде отримано звіт проготовність програми до роботи та встановлення праметрів роботи по замовчувані.

Для ораганізації взаємодії користувачів з програмою було закладено блок вводу програмного коду у вигляді або текстового файлу або безпосереднього вводу текстової інформації у відповідне вікно програми. У першому випадку користувач може скористатись можливістю повернутись до програмного коду який було створено під час попереднього запуску програмного додатку або створеного в інших інтегрованих середовищах розробки програмних засобів на мовах високого рівня. Даний підхід дозволяє провести оцінку програмного коду створеного як іншими користувачами так і провести оцінку вже існуючого коду. В другому випадку користувачі можуть в режимі реального часу провести оцінку трансляції програмного коду та провести оптимізацію написаного коду з метою отримати більш швидший код для виконання якого треба менша кількість апаратних ресурсів.

Одним з основних блокі програмної системи є набір функцій за допомого якого організовано взіємодію користувача з базоб даних моделей, що описують структури мови програмування С++ (рисунок 3.2). Серед функціональних можливостей які є в даному блоці слід відмітити стандартні засоби для взаємодії з базою даних. Початковий набір моделей задається по замовчуванню, проте користувача можуть додавати нові моделі або вносити корективи в вже існуючі. Окремо, слід відзначити функції пошуку відповідних моделей відносно виділених ключових слів. Пошук здійснюється на основі бінарного алгориму, що забезпечить швидкий пошук та не призведе до додатковиз часових затримок роботи програми в цілому.

Модуль роботи з БД моделей

Пошук моделей

Корекція моделей

Додавання моделей

Оцінка відповідності моделі

Рисунок 3.2 – Узагальнена структура модуля роботи з моделями

В результаті роботи даного набуру функцій проводиться взаємодія з базою даних та визначається модель яка буде відповідати аналізованій частині програмного коду.

Блок трансляції програмного коду мовою програмування високого рівня на мову програмування асемблер є допоміжним та працює на основі отриманої моделі та проведення семантичного аналізу стрічки програмного коду або блоку команд з метою отримання параметрів відповідної команди. Пошук відбуваєть на основі знайденої моделі оформлення команди на мові С++, та може бути проаналізованим в залежності від стилю написання програмного коду конкретним користувачем. Наприклад програмний код:

number = number +1;

number+=1;

number++;

По своїй суті виконує одну і туж операції, а саме збільшення значення змінної *number* на одиницю, а отже дані стріки коду будуть транслітеровані як один код на асемблері. В той самий час на мові С++ даний програмний код має різний синтаксичний формат. Результатом роботи функцій даного блоку буде програмний код перетворений з формату мови програмування високого рівня в формат на мові асемблер.

Блок виводу отриманих результатів роботи програмного додатку – це допоміжна структурна одиниця запропонованої архітектури. Головний завданням даного блоку є організація коректного відображення трансльованого коду у формат зручний для перегляду користувачем. Окрім того, як додатковий функціональний засіб, в процесі реалізації було додано можливість зберігання отриманих результатів в вигляді текстового файла.

Дана архітектура є цілісним технічним рішенням. Що забезпечує можливість швидкої програмної реалізації окремих складових елементів, проведення тестування як розробки в цілому так і кожного модуля, модифікація та нарощування як окремих функціональних мождивостей окремо взятого структурного блоку так і додавання принципово нових структурних елементів. При цьому загальна цілісність та основні функціональні можливості програмного додатку траняляції програмного коду мовою високого рівня на мову асемблер не зазнають відчутних змін.

Після завершення етапу проектування та дослідження структури програмного додатку необхідно провести адаптацію запропонованого алгоритму та моделювання отриманої архітектури програмного додатку з метою визначення можливих проблемних ситуацій та проблеми при обміні інформацією між різними структурними областями програми. Дане моделювання було проведено засобами універсальної мови моделювання з використання попередньо отриманого досвіду розробки програмних додатків. Для проведення процесу моделювання було обрано наступні діаграми, що входять в пакет UML. А саме це:

* діаграми класів (аналіз структурування даних);
* діаграми предедентів (можливості доступу окремих груп користувачів до функціональних можливостей програми);
* діаграми посділовностей (аналіз можливих сценаріїв поведінки користувачів та програми для досягнення поставлених результатів).

Процес моделювання очевидно має значні переваги, оскільки дозволяє ще на початкових етапах визначити проблемні місця в запропонованій архітектурі та алгоритмах, а також промоделювати можливі помилкові дії користувачів. Що дозволить внести відповідні корективи та продовжити процес розробки програмного додатку без додаткових фінансових та часових затрат. Проте, саме по собі моделювання вимагає короткострокових витрат на його якісне та швидке проектування. Серед моживих затрат, які необхідно врахувати в процесі моделювання:

* процес моделювання сам по собі теж вимагає деяких часових затрат, адже необхідно провести підбір кваліфікованих фахівців, надати їм час на ознайомлення запропонованими розробками та час на те щоб вони створили та проаналізували відповідні моделі. Окрім того незважаючи на те, що моделі всетаки є схематичними і не залежить від робочого коду, проте час на сам процес моделювання всетаки необхідно закласти в терміни виконання замовлення.
* моделювання є суб’єктивним процесом і його результат залежить від кваліфікації людей які будуть його проводити, окрім того під час моделювання слід врахувати той фактор, що проводитьс аналіз цілої системи, а не її окремих елементів. Тому при недостатньому рівні уваги до деталей можлива втрата інформації, що в подальшому може призвести до більш критичних проблем на етапі реалізації програмного коду.
* Для проведення моделвання необхідні робочі станції та додаткові апаратні ресурси які виділені на проект в цілому. Тому ще одим фажливим фактором є ресурсні втрати під час проведення етапу проектування. Очевидним є і той факт, що коли проводиться моделювання то процес створення програмного продукту зупиняється в очікуванні отриманих результатів. А це в свою чергу призводить до додаткових часових та фінантових витрат.

Проте незважаючи на перелічені недолікиетап моделювання є важливим та обов’язковим елементом проектування та реалізацій програмних систем зі складною структурою. Для перевірки доступу до функціональинх можливостей програмного додатку за основу було взято моделювання за допомогою UML діаграми прецедентів. На даній діаграмі будуть відображатись можливості доступу окремих груп користувачів до функціоналу програмного додатку, відображено їх взаємодію в середині запропонованих програмних блоків. Приклад діаграми прецеденів наведено на рисунку 3.3.



Рисунок 3.3 – Діаграма прецедентів програмної системи

В результаті аналізу діаграми прецедентів надав можливість оцінити загальні зв’язки які закладені між окремими блоками системи, провести дослідження структур та форматів даних за допомогою яких буде організовуватись інтерфейс взаємодії, а також доступ кожної з груп користувачів до механізмів програмного модуля. Зокрема як показано на риунку 3.3 група акторів «Користувач» має доступ до засобів введення програмного коду мовою високого рівня, засобів роботи з базо даних моделей програмних структур, а також до аналізу отриманих результатів. Даний абір дозволяє користувачеві повністю контролювати процес роботи програми та впливати на отримані результати. Проте частина функцій прихована від нього з метою мінімізувати людський фактор під час роботи програми та зменшити ймовірність критичного завершення програми в зв’язку з неправильними діями користувача.

Окрім аналізу організації внутрішніх механік робти програмного додатку іншим важливим пунктом перевірки запропонованої архітектури та алгориму є аналіз сценаріїв поведінти користувача та системи які закладаються в роботи програми. Для дослідження запропонованих сценаріїв використано діаграму послідовностей, яка в ходить в склад універсаьної мови моделювання. Дана ліаграма позволяє розглянути принципи взаємодії акторів та програмної системи при винекнені ситуацій при вирішенні поставлених завдань. Результат виконаного моделювання наведено на рисуноку 3.4.



Рисунок 3.4 – Діаграма послідовності програмної системи

Під час провведення моделювання проводвся аналіз кроків користувачів для отримання програмного коду з клавіатури або з файлу, наявністьпомилок в вхідному програмному коді тощо. В результаті подулювання було зроблено висновки, що запропонована структура програмного додатку. А також алгоритм трансляціх програмного коду з однієї мови на іншу повністю виконують поставлені завдання. Помилкові ситуації не виникали, отримані результати моделювання були очікуваними та в межах допустимої похибки.

На останньому атапі моделювання необхідно було проаналізувати повноту запропоновних структур даних та наявність у них засобів по взаємодії між собою в межах запропонованої архітектури. З цією метою було проведено моделювання на основі діаграми класів. Отримані результати проілюстровані на рисунку 3.5



Рисунок 3.5 – Діаграма класів програмної системи

Даний тип аналізу був використаний для оцінки внутрішньої структури класів які будуть реалізовані в межах розробки даного програмного додатку. Візуальний аналіз отриманої діаграми класі демострує той факт, що всі класи у достатній мірі описують відповідні сутності, набір використаних атрибутів у повній мір однозначно дозволяє характеризувати обєкти відповідних кламів, а множина методі виконати усі поставлені перед ними завдання. Окрім в того в структуру класів було закладено механізм обміну даними за наперед узгодженими правилами та з дотримками відповідних інтерфейсів.

В підсумку етап моделювання проводився на основі використання трьох різних факторів оцінки архітектури та розроблених структур даних, що в кінцевому результаті проілюстрували коректність роботи програми пдічас виконаня завдань, а також мінімізації винектення непередбачуваних ситуацій в результаті впливу користувачів на процеси в середині програмного модуля.

Наступним завдання яке необхідо вирішити в межах проектування та розробки системи генерування опимального програмного коду – це розробка інтерфейсу для надання моживості користувачеві у зручній та доступній формі завантажувати вхідні дані та аналізувати результати робти програмного додатку. Оскільки користувач буде безпосередньо взаємодіяти з програмною розробкою, то для підвищення зручності було розроблено графічний інтерфейс з подвійним меню. Загальний вигляд запропонованого графічного інтерфейсу для проектуованої системи наведено на рисунку 3.6.



Рисунок 3.6 – Графічний інтерфейс користувача

Про проектування інтерфейсу головного вікна програмного додатку були враховані наступні фактори:

* основний пит вхідної та вихідної фнвотмації буде представлений у текстовому форматі;
* кількість активних керуючих елементів буде мінімальною, оскільки для коректної роботи програми необхідний мінімальний вплив людського фактору;
* процес вводу програмного коду повинен бути зручним та не викликати додаткові запитання у користувачів;
* результат роботи програми повинен виводитись у зручному для користувача форматі та не викликати дискомфорт під час візуального оцінювання.

Базуючись на цих вимогах було розроблено графічний інтерфйс користувача, що повністю задовільняє поставленим умовам, дозволяє користувачаві у зручному форматі отримувати доступ до всіх елементів управління програмою, а отримані результати ілюструються в мксимально зручному для користувача форматі.

Додатково було проведено аналіз на визначення обов’язкових та допоміжних кроків, що повинен здійснити користувач для досягнення поставленого завдання при мінімальних часових затратах. Набір відповідної послідовності дій відображено на рисунку 3.7:

Активація системи

Встановлення параметрів роботи

Ввід програмного коду на мові С++

Корекція масиву моделей

Аналіз тексту та пошук моделі

Трансляція програмного коду

Рисунок 3.7 – Послідовність дій користувача при роботі з системою трансляції програмного коду мовою високого рівня на мову асемблер

Як проілюстровано на рисуноку 3.7 кількіст дій для досягнення очікуваного коректного результату включає всебе три основні операції:

* ввід програмного коду на мові високого рівння;
* вибір моделі опису частини програмного кодуна основі пошуку ключових слів та параметрів;
* проведення візуальної оцінки ориманих результатів.

Дані оперції можуть бути підсилені ще рядом допоміжних функці, таких як: корекія параметрів робти програми, налаштування бази даних моделей тощо. Дані дії призначені для підвищення якості отриманих результатів та підвищення рівння зручності використання запропонованої системи.

## 3.2 Програмні модулі для оцінки та генерації програмного коду

Процес роботи програмного додатку можна умовно розділити на декілька незалежних етапів починаючи від вводу програмного коду, пошук ключових слів, визначення моделів, заповнення параметрів моделі, трансляція програмного коду. Якщо перші два етапи не викликають запитання, їх можна достатньо просто реалізувати за допомогою стандартних засобів мови програмування за допомогою організації обробки вхідного каналу інформації та відображення в класі візуалізації текстових даних. А другий атап це фактично послідовний перебір введених слів з метою отримання співпадіння у базі даних моделей.

if(!searchModel(str))

 return -1;

else

{

 int valueModel = hash(str);

 if(arrayModel[value].length != 0)

 {

 return valueModel;

 }

}

То для знаходження параметрів моделінеобхідно провести додатковій аналіз отриманої стрічки програмного коду. На рисунку 3.8 наведено типовий приклад програмного коду циклічного алгоритму.



Рисунок 3.8 – Приклад програмного коду циклічного алгоритму на мові С++

При знаходжені в срічці коду ключового слова *for* очевидним стає той фак, що в даному місці буде знаходити частина коду яка відповідає за реалізацію набору команд які будуть повторятись. Утворюючи собою цикл, при цьому згідно стандарту мови С++ біля ключового слова повинні бути додатково розміщені ще ряд елементів, с саме круглі дужки «()» в яких повинні перебувати параметри робти циклу, а також символи «;» які повинні розділяти внутрішній простів в круглих дужках на три окремих сектори. Окрім того після набору даних керуючих символів додатково може розпочинатись блок команд «тіла циклу» які повинні знаходитись в фігурних дужках. Після знаходження всіх структурних елементів та визначення параметрів для обраної моделю, проводиться додаткових аналіз «тіла циклу»:

array[i] = rand() % 5 + 12;

cout << array[i] << ' ';

Оскільки «тіло циклу» в свою чергу є програмним кодо який необліхно перетворити в асемблерне представлення, то для нього теж застосовується посторний аналіз. Дана процедура повторюєится до того моменту коли буде повністю проаналізований програмний код тарозбитий на окремі прості складові.

Для оцінки проведення перетворення програмного коду в асемблерне представлення виконаємо ряд перетворень програмного коду, що буде ілюструвати прості типи лагоритмів.

Лінійні алгоритми характеризують послідовним виконання усіх кроків, та не вимагають переходів в середині програми. Всі преходи здійснюються в порядку розташування програмних команд. На рисунку 3.9 наведений приклад програмної реалізації простого лінійного алгоритму додавання двох змінних та результат його трансляціх на сову асемблер.

|  |  |
| --- | --- |
|  |  |
| а) мова С++ | b) мова асемблер |

Рисунок 3.9 – Приклад трансляції лінійного алгоритму

Як видно з наведеного прикладу для виконання простого лінійного алгоритму на мові асемблер довелось додатково використати регістри, щоб реалізувати процедуру отримання суми двох змінних які розташовані в комірках пам’яті. Проте конвертація програмного коду дакого виду не викликає складнощів, оскільки всі команди йдуть послідовно, вони є однозначними та їх чітко можна ідентифікавати. Вони не мають між собою якихось внутрішніх зв’язків. Кожний рядок програмного коду можна з легкістю конвертувати окремо, приймаючи до уваги тільки парметри які будуть характерні для простих операторів. В прикладі це оперетори присвоєння «=» та отримання суми «+».

Для аналізу можливості трансляції програмних кодів що відповідають алгоритмам розгалуження розглянемо приклад наведений на нирунку 3.10.

|  |  |
| --- | --- |
|  |  |
| а) мова С++ | b) мова асемблер |

Рисунок 3.10 – Приклад трансляції алгоритму розгалуження

Даний варіант є складнішим для трансляції оскільки в ньому вже присутній елемент переходу міжрізними частинами програмного коду з використанням команд умових та безумовних переходів. Це є очевидним, оскільки програма на мові асемблер це є послідовність команд які йдуть одна заодною, то пропуск частини команд або перехід на конкретний сегмент коду може відбутись тільки за рахунок примусового переміщення вказівника активної команди.

Для завершення аналізу можливостей трансляції програмних кодів проаналізуємо приклад перекодування циклічного алгоритму. Для даного типу алгоритмів характерним є те, що в ньому повинна бути присутня чстина команд які будуть повторюватись. Ц дозволить програмісто зменшити розмір файлу за рахунок відміни потреби написання подібного коду декілька разів. Прика конвертації наведено на рисуноку 3.11.

|  |  |
| --- | --- |
|  |  |
| а) мова С++ | b) мова асемблер |

Рисунок 3.11 – Приклад трансляції циклічного алгоритму

## 3.3 Тестування та аналіз реалізованого програмного додатку

На завершальному етапі процесу створення програмного додатку для трансляції програмного коду мовю високого рівня на мову асемблер було провденно тестування готової програми. Для успішного провдення тестування та для отримання об’єктивних результатів під час підбору апаратних характеристи робочої станції необхідно враховувати осодивості експлуатації програмної ситсеми. Серед обмежень які накладаються на програмну систему слід відмітити:

* вхідна та вихідна інформація буде подаватись у текстовому форматі, що змешує вимоги до характеристик пристроїв вводу даних;
* перекодування повинну здійснюватис без помітних часових затримок при умові, що розмір вхідного програмного коду є в межах допустимого;
* вивід результатів роботи повинен здійснюватись у зручному для користувачі форматі;
* вхідні та вихідні дані повинні мати моливісь бути збереженими на твері носії даних для повторного використання.

Аналіз наявних на сьогодні апаратних рішень та їх цінові арактеристики показав, що для провдення тестування робоча станція може належати до помірного цінового діапазону та відноситись до «офісного» типу комп’ютерів. Дана група характеризується помірними технічними характеристиками, проте їх вартість теж є низькою. Ни рисунку рисунок 3.13 наведено технічні параметри:



Рисунок 3.12 – Технічні характеристики тестової робочої станції

Для даного типу комп’ютерів буде характерне використання для прости задач які не вимагають складних обчислень, наприклад такі машини можуть використовувати секретарі, бухгалтери, прості користувачі для робти з офісними програмами та перегляду інформації в глобальній мережі.

Окремо звернумо увагу на те, що розробники програмного забезпечення багато часу проводят дивлячить на монітор робочого комп’юптера, тому для перегляду результатів роботи програми та підвищення зручності роботи з розробленою програмною системою було обрано монітор середньої цінової категорії та технічними параметрами Технічні характеристики пристрою відображення зображень (DELL P2018H 19.45″ ) представлені на рисунку 3.13:



Рисунок 3.13 – Технічні характеристики пристрою відображення зображень

При аналізі обраних технічних параметрів було відмічено, що дана конфігурація у повній мірі підходить для виконання поставлених завдань та може успішно використовувати для проведення процесу тестування розроленого програмного забезпечення та отримання об’єктивну оцінку. Дана конфігурація може бути легко повторена з метою перевірки отриманих результатів з метою підтвердження отриманих даних. Тому даного набуру апаратних засобів обєднаних в систему тестування є достітньо для проведення процесу тестування програмної системи транслювання програмного коду мовою високого рівня на мову асемблер.

Для перевірки функціональни можливостей розробленого програмного додатку було сформовано набір критеріїв на основі яких були підібрані типи програмних тестів, які дозволять оцінити роботу програмної розробки. Серед критеріїя які враховувались були:

* тип алгоритма який використовується;
* кількість необхідних переходів в середині алгоритмів;
* обробка масивів;
* наявність поєднання декількох типів алгоритмів.

На основі даних параметрів було обрано наступні типи алгоримтів які будуть предворені з однієї мови на іншу:

* прості лінійний алгоритми – дана група алгоритмів характеризується просторою внутрішньою структурою без додаткових переходів в середині програмного коду. В алгоритмах не передбачена робробка масивів;
* прості алгоритми розгалуження – в даній групі алгоритмів представлені приклади в який використовуються елементи розгалуження при прийнятті рішень. Розгалуження може бути як одинарним так і множинним. В алгоритмах не передбачено використання коскадів розгалуження;
* прості циклічні алгоритми – алгоритми даної групи характеризують присутністю в них елементів повторення програмного коду. Масиви в даних алгоритмах не застосовуються. Тіло циклу містить тільки лінійні елементи алгоритмів. В алгоритмах не передбачено механізм вкладених циклів;
* прості гібридні алгоритми – дана група алгоритиів репрезентує приклади алгоритмів які поєдрують у собі різні типи простих алгоритмів. Використання масивів в даному випадку не предбачено;
* прості алгоритми з використанням масивів – для оцінки можливості роботи програмного засобу з алгоритмами в яких передбачено обробку даних збережених у форматі масиву. Алгоритми можуть бути тільки одного типу;
* гібридні алгоритми з обробкою масивів – це найбільша група алгоритмів які представляють звичайні алгоритми які використовуються для вирішення звичних задач під час програмування.

Результати провденого тестування були опрацьовані на представлені наведеній таблиці 3.1

Таблиця 3.1 – Узагальнена таблиця результатів тестування

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тестові групи/Тип оцінювання | Швидкість опрацювання | Поява помилок | Складність алгоритму | Обробка масивів даних |
| прості лінійний алгоритми | Мінімальний час | Відсутні | Простий алгоритм | Відсутня |
| прості алгоритми розгалуження | Мінімальний час | Відсутні | Простий алгоритм | Відсутня |
| прості циклічні алгоритми | Мінімальний час | Відсутні | Простий алгоритм | Відсутня |
| прості гібридні алгоритми | Мінімальний час | Відсутні | Простий алгоритм | Відсутня |
| прості алгоритми з використанням масивів | Мінімальний час | Відсутні | Складний алгоритм | Присутня |
| гібридні алгоритми з обробкою масивів | Мінімальний час | Відсутні | Складний алгоритм | Присутня |

Тримані результати тестування проілюстрували, що виокристаня структура та розроблений набір алгоритмів є доцільним під час проектування та розробки прогармної ситеми опрацювання програмних кодів.

## 3.4 Висновки до розділу

Реалізовано та здійснено поточне моделювання програмного модуля трансляції програмного коду мовою високого рівня на мову ассемблер, що надало можливість програмно реалізувати програмних додаток генерації оптимального програмного коду.

Здійснено тестування розробленого додатоку генерації оптимального програмного коду, що підтвердило правильність у виборі методів, алгоритмів та цифрових бібліотек при реалізації програми та розробленого алгоритму.

# Висновки

На основі аналізу сучасних інтегрованих середовищ розробки та алгоритмів трансляції програмних кодів мовами високого рівня можна зробити наступні висновки:

1. Проведено аналіз та класифікацію програмних додатків на основі їх функціональних можливостей, що дозволило виділити основні внутрішні архітектурні особливості та формати даних при створенні виконуваних файлів.
2. Проаналізовано та проведено класифікацію мов програмування високго та низького рівнів, що дозволило визначити мосу С++ як основну для подальших досліджень.
3. Проведено аналіз сучасних інтегрованих середовищ розробки програмних додатків, на основі дослідження принципів проведення трансляції з мов високого рівня на мову асемблер та механізми формування виконуваного файла.
4. Проведно аналіз алгогитмів та механізмів трансляції програмного коду мовою високого рівня на мову асемблер на основі використання шаблонів перекодування, що дозволило розробити алгоритм перекодування програмного коду на мову асемблер.
5. Розроблено алгоритм генерації оптимального машинного коду для мов високого рівня, що дало можливість розробити структуру та програмно реалізувати додаток написання оптимального програмного коду на мові високого рівня.
6. Здійснено тестування розробленого додатоку генерації оптимального програмного коду, що підтвердило правильність у виборі методів, алгоритмів та цифрових бібліотек при реалізації програми та розробленого алгоритму.

# Список використаної літератури

1. Сергійчук В.В., Демчук Ю.І. Алгоритм конвертації програмних кодів з оптимізацією. Збірник тез VІ Науково-практична конференція молодих вчених і студентів «Інтелектуальні комп’ютерні системи та мережі»., Тернопіль, 10 листопада 2022 р. с. 28.
2. Сергійчук В.В., Демчук Ю.І. Алгоритм виділення окремих елементів стрічки програмного коду на основі семантичного аналізу. Збірник тез VІ Науково-практична конференція молодих вчених і студентів «Інтелектуальні комп’ютерні системи та мережі»., Тернопіль, 10 листопада 2022 р. с. 28.
3. Adenso-Diaz, B. and Laguna, M. Fine-tuning of algorithms using fractional experimental design and local search. Operations Research 54, 1 (Jan.–Feb. 2006), 99–114.
4. Babic´, D. and Hu, A.J. Structural abstraction of software verification conditions. In Proceedings of the 19th International Computer Aided Verification Conference, Vol. 4590 LNCS. Springer-Verlag, Berlin/ Heidelberg, 2007, 366–378.
5. Battiti, R., Brunato, M., and Mascia, F. Reactive Search and Intelligent Optimization. Operations Research/ Computer Science Interfaces Series, Vol. 45. Springer, 2008.
6. Birattari, M., Yuan, Z., Balaprakash, P., and Stützle, T. F-Race and Iterated F-Race: An overview. In Experimental Methods for the Analysis of Optimization Algorithms. Springer-Verlag, Berlin/Heidelberg, 2010, 311–336.
7. Carchrae, T. and Beck, J. Applying machine learning to low knowledge control of optimization algorithms. Computational Intelligence 21, 4 (Nov. 2005), 373–387.
8. Cheeseman, P., Kanefsky, B., and Taylor, W.M. Where the really hard problems are. In Proceedings of the 12th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, San Mateo, CA, 1991, 331–337.
9. Da Costa, L., Fialho, Á., Schoenauer, M., and Sebag, M. Adaptive operator selection with dynamic multi-armed bandits. In Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation. ACM Press, New York, 2008, 913–920.
10. Diao, Y., Eskesen, F., Froehlich, S., Hellerstein, J.L., Spainhower, L., and Surendra, M. Generic online optimization of multiple configuration parameters with application to a database server. In Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Vol. 2867 LNCS. Springer-Verlag, Berlin/Heidelberg, 2003, 3–15.
11. Gomes, C.P. and Selman, B. Algorithm portfolios. Artificial Intelligence 126, 1–2 (Feb. 2001), 43–62.
12. Guerri, A. and Milano, M. Learning techniques for automatic algorithm portfolio selection. In Proceedings of the 16th European Conference on Artificial Intelligence. IOS Press, Amsterdam, 2004, 475–479.
13. Hamerly, G. and Elkan, C. Learning the k in k-means. In Proceedings of the Conference on Advances in Neural Information Processing Systems. MIT Press, Cambridge, MA, 2004, 281–288.
14. Hoos, H.H. Automated algorithm configuration and parameter tuning. In Autonomous Search, Y. Hamadi and F. Saubion, Eds. Springer-Verlag, 2011.
15. Hoos, H. Programming by Optimisation. Technical Report TR-2010-14. Department of Computer Science, University of British Columbia, Vancouver, 2010.
16. Hoos, H. Computer-Aided Design of High-Performance Algorithms. Technical Report TR 2008-16. Department of Computer Science, University of British Columbia, Vancouver, 2008.
17. Hoos, H.H. and Stützle, T. Stochastic Local Search: Foundations and Applications. Morgan Kaufmann Publishers, San Francisco, 2004.
18. Huberman, B., Lukose, R., and Hogg, T. An economics approach to hard computational problems. Science 275, 5296 (Jan. 1997), 51–54.
19. Hutter, F., Babic´, D., Hoos, H.H., and Hu, A.J. Boosting verification by automatic tuning of decision procedures. In Proceedings of Formal Methods in Computer-Aided Design. IEEE Computer Society Press, Los Alamitos, CA, 2007, 27-34.
20. Hutter, F., Hoos, H., Leyton-Brown, K., and Stützle, T. ParamILS: An automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36 (Sept.-Dec. 2009), 267–306.
21. Hutter, F., Hoos, H., and Stützle, T. Automatic algorithm configuration based on local search. In Proceedings of the 22nd National Conference on Artificial Intelligence. AAAI Press, Palo Alto, CA, 2007, 1152–1157.
22. Hutter, F., Hoos, H.H., and Leyton-Brown, K. Automated configuration of mixed integer programming solvers. In Proceedings of the Seventh International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Vol. 6140 LNCS. Springer-Verlag, Berlin/Heidelberg, 2010, 186–202.
23. Hutter, F., Hoos, H.H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In Proceedings of the Fifth International Conference on Learning and Intelligent Optimization, Vol. 6683 LNCS. Springer-Verlag, Berlin/ Heidelberg, 2011, 507–523.
24. Kadioglu, S., Malitsky, Y., Sellmann, M., and Tierney, K. ISAC: An instance-specific algorithm configuration. In Proceedings of the 19th European Conference on Artificial Intelligence. IOS Press, Amsterdam, 2010, 751–756.
25. KhudaBukhsh, A., Xu, L., Hoos, H., and Leyton-Brown, K. SATenstein: Automatically building local search SAT solvers from components. In Proceedings of the 21st International Joint Conference on Artificial Intelligence. AAAI Press, Palo Alto, CA, 2009, 517–524.
26. Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. A portfolio approach to algorithm selection. In Proceedings of the 18th International Joint Conference on Artificial Intelligence. Morgan Kaufmann Publishers, San Francisco, 1542–1543.
27. Li, X., Garzarn, M.J., and Padua, D. Optimizing sorting with genetic algorithms. In Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society Press, Washington, D.C., 2005, 99–110.
28. Maron, O. and Moore, A.W. Hoeffding races: Accelerating model selection search for classification and function approximation. In Proceedings of the Seventh Conference on Advances in Neural Information Processing Systems. Morgan Kaufmann Publishers, San Francisco, 1994, 59–66.
29. Monette, J.N., Deville, Y., and Hentenryck, P.V. Aeon: Synthesizing scheduling algorithms from high-level models. Operations Research and Cyber-Infrastructure Series, Vol. 47. Springer Science+Business, New York, 2009, 43–59.
30. Nell, C.W., Fawcett, C., Hoos, H.H., and Leyton-Brown, K. HAL: A framework for the automated design and analysis of high-performance algorithms. In Proceedings of the Fifth International Conference on Learning and Intelligent Optimization, Vol. 6683 LNCS. Springer-Verlag, Berlin/Heidelberg, 2011, 600–615.
31. Nudelman, E., Leyton-Brown, K., Devkar, A., Shoham, Y., and Hoos, H.H. Understanding random SAT: Beyond the clauses-to-variables ratio. In Principles and Practice of Constraint Programming, Vol. 3258 LNCS. Springer-Verlag, Berlin/Heidelberg, 2004, 438–452.
32. Pan, Z. and Eigenmann, R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society Press, Washington, D.C., 2006, 319–332.
33. Mohan, M.; Greer, D. A survey of search-based refactoring for software maintenance. J. Softw. Eng. Res. Dev. 2018, 6, 3–55.
34. Shepperd, M.; Kadoda, G. Comparing software prediction techniques using simulation. IEEE Trans. Softw. Eng. 2022, 27, 1014–1022.
35. Azeem, M.I.; Palomba Sanders, P. and Schultes, D. Engineering fast route planning algorithms. In Proceedings of the Sixth International Workshop on Experimental Algorithms, Vol. 4525 LNCS. Springer-Verlag, Berlin/Heidelberg, 2007, 23–36.
36. Spall, J. Introduction to Stochastic Search and Optimization. John Wiley & Sons, Inc., New York, 2003.
37. Vallati, M., Fawcett, C., Gerevini, A., Hoos, H.H., and Saetti, A. Automatic generation of efficient domainoptimized planners from generic parametrized planners. In Proceedings of the Eighth RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, 2011;
38. Simon, F.; Steinbruckner, F.; Lewerentz, C. Metrics based refactoring. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, 14–16 March 2001; pp. 30–38.
39. Baqais, A.; Alshayeb, M. Automatic software refactoring: A systematic literature review. Softw. Qual. J. 2020, 28, 459–502.
40. Mariani, T.; Vergilio, S.R. A systematic review on search-based refactoring. Inf. Softw. Technol. 2017, 83, 14–34.
41. Westfold, S.J. and Smith, D.R. Synthesis of efficient constraint-satisfaction programs. Knowledge Engineering Review 16, 1 (Mar. 2001), 69–84.
42. Whaley, R.C., Petitet, A., and Dongarra, J.J. Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27, 1–2 (Jan. 2001), 3–35.
43. Xu, L., Hoos, H., and Leyton-Brown, K. Hydra: Automatically configuring algorithms for portfoliobased selection. In Proceedings of the 24th AAAI Conference on Artificial Intelligence. AAAI Press, Palo Alto, CA, 2010, 210–216.
44. Xu, L., Hutter, F., Hoos, H.H., and Leyton-Brown, K. SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32 (MayAug. 2008), 565–606.
45. Tsantalis, N.; Chatzigeorgiou, A. Identification of refactoring opportunities introducing polymorphism. J. Syst. Softw. 2022, 83, 391–404.
46. Heged ˝us, P.; Kádár, I.R. Ferenc, Gyimóthy T. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. Inf. Softw. Technol. 2018, 95, 313–327.
47. Gharehchopogh, F.; Gholizadeh, H. A comprehensive survey: Whale Optimization Algorithm and its applications. Swarm Evol. Comput. 2019, 48, 1–24.
48. Rosli, M.; Teo, N.H.I.; Yusop, N.S.M.; Mohamad, N.S. Fault prediction model for web application using genetic algorithm. Int. Conf. Comput. Softw. Modeling (IPCSIT) 2011, 14, 71–77. 39. Ebrahimi, A.; Khamehchi, E. Sperm whale algorithm: An effective metaheuristic algorithm for production optimization problems. J. Nat. Gas Sci. Eng. 2016, 29, 211–222.
49. Березький О.М., Дубчак Л.О., Мельник Г.М. Методичні рекомендації до виконання кваліфікаційної роботи з освітнього ступеня “Магістр”. Спеціальність: 123 - Комп’ютерна інженерія. Магістерська програма - Комп’ютерна інженерія". Тернопіль: ЗУНУ, 2022. 32 с.
50. Гураль І.В., Дубчак Л.О. Методичні вказівки до оформлення курсових проектів, звітів про проходження практики, випускних кваліфікаційних робіт для студентів спеціальності «Комп’ютерна інженерія» Тернопіль: ТНЕУ, 2019. 33 с.