Прикарпатський національний університет імені Василя Стефаника
Фізико-технічний факультет
Кафедра комп'ютерної інженерії та електроніки

Нечай Юрій Ігорович
Yurii Nechai

УДК 004:42

Спеціальність 123 «Комп'ютерна інженерія»

Кваліфікаційна робота
на здобуття освітнього ступеня бакалавра

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

Науковий керівник:
д.т.н, професор Ігор КОГУТ
Рецензент:
д.ф.-м.н., проф. каф. матер.
і новітніх технологій,
Іван ЯРЕМІЙ

Івано-Франківськ
2024

Формат	Поз.	Позначення	Найменування	К-ть	Прим.
А4			Діаграма прецендентів прогармної системи	1	
А4			Блок-схема алгоритму трансляції	1	
А4			Пояснювальна записка	64	

Змн.	Арк.	№ докум.	Підпис	Дата	123.КІ-41.11		
Розробив		Нечай Ю.І.			Специфікація	Літ.	Арк. 2 / Аркушів 1
Перевірив		Когут І.Т.					
Н. Контр.							
Затвердив							

АНОТАЦІЯ

Мета даної кваліфікаційної роботи полягає у дослідженні та розробці алгоритмів генерації оптимального машинного коду для мов високого рівня. Оптимальний машинний код визначається як код, що забезпечує максимальну ефективність виконання програми на цільовій апаратурі за умов обмежень щодо ресурсів, часу виконання та інших параметрів.

Актуальність теми обумовлена стрімким розвитком програмної індустрії та зростаючим попитом на високопродуктивне програмне забезпечення. Мови високого рівня пропонують зручність у програмуванні, але ефективність згенерованого машинного коду може значно вплинути на загальну продуктивність програми. Отже, оптимізація процесу генерації машинного коду є вирішальним фактором у досягненні високої ефективності програмних продуктів.

Кваліфікаційна робота включає в себе аналіз поточних методів генерації машинного коду для мов високого рівня, розробку нових алгоритмів оптимізації та проведення експериментів для порівняння продуктивності нових алгоритмів з існуючими методами, використовуючи специфічні показники ефективності.Rephrase Результати дослідження підвищать якість і продуктивність програмного забезпечення, а також розширять теоретичне розуміння в сферах компіляції та оптимізації програм

ABSTRACT

Goal of this qualification thesis is the investigation and development of algorithms for the generation of optimal machine code for high-level programming languages. Optimal machine code is defined as code that ensures the maximum execution efficiency of a program on the target hardware, taking into account resource constraints, execution time and other parameters.

The relevance of the topic arises from the rapid growth of the software industry and the increasing demand for high-performance software. While high-level languages offer programming convenience, the efficiency of the generated machine code can significantly affect program performance. Therefore, optimizing the process of generating machine code is a key aspect to achieve high software efficiency.

The tasks of the qualification work include: the analysis of existing methods for generating machine code for high-level languages, the development of new optimization algorithms and the experimental comparison of new algorithms with existing methods using various efficiency metrics.

The research results will contribute to improving the quality and productivity of software and to expanding theoretical knowledge in the field of program compilation and optimization

ПЕРЕЛІК ОСНОВНИХ СКОРОЧЕНЬ

АГОМК - Алгоритм генерації оптимального машинного коду.

МВР - Мови високого рівня.

ОПМК - Оптимальний машинний код.

КР - Кваліфікаційна робота.

ОП - Оптимізація.

КД - Кодогенерація.

ІО - Інструкційний оптимізатор.

ДР - Додаткові ресурси.

КС - Компілятор мови високого рівня.

АС - Аналіз семантики.

КО - Кодоптимізатор.

МА - Машинна архітектура.

РТ - Реєстрові трансформації.

ПК - Підтримка компілятора.

СГ - Система генерації.

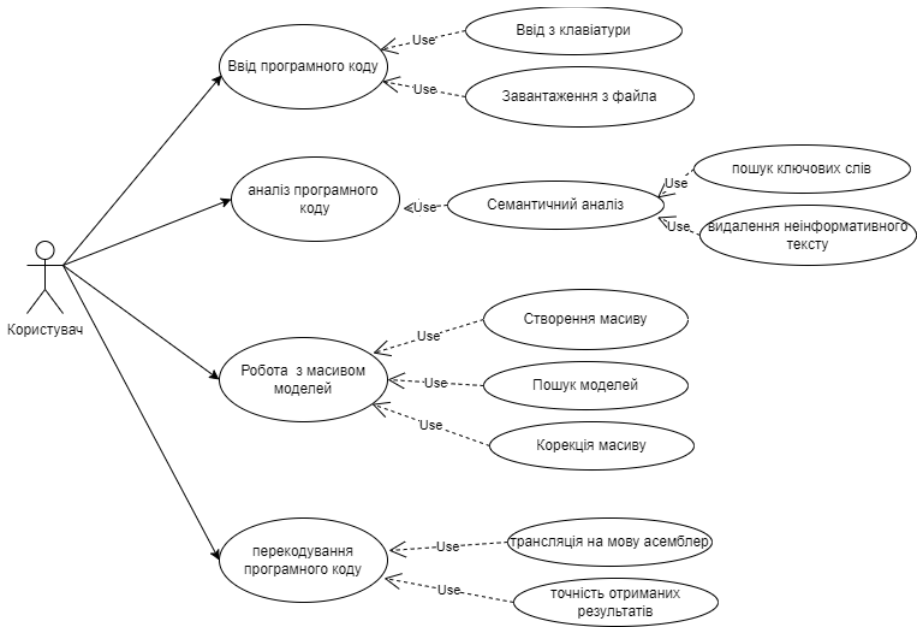
СА - Структурний аналіз.

ПО - Процедурні оптимізації.

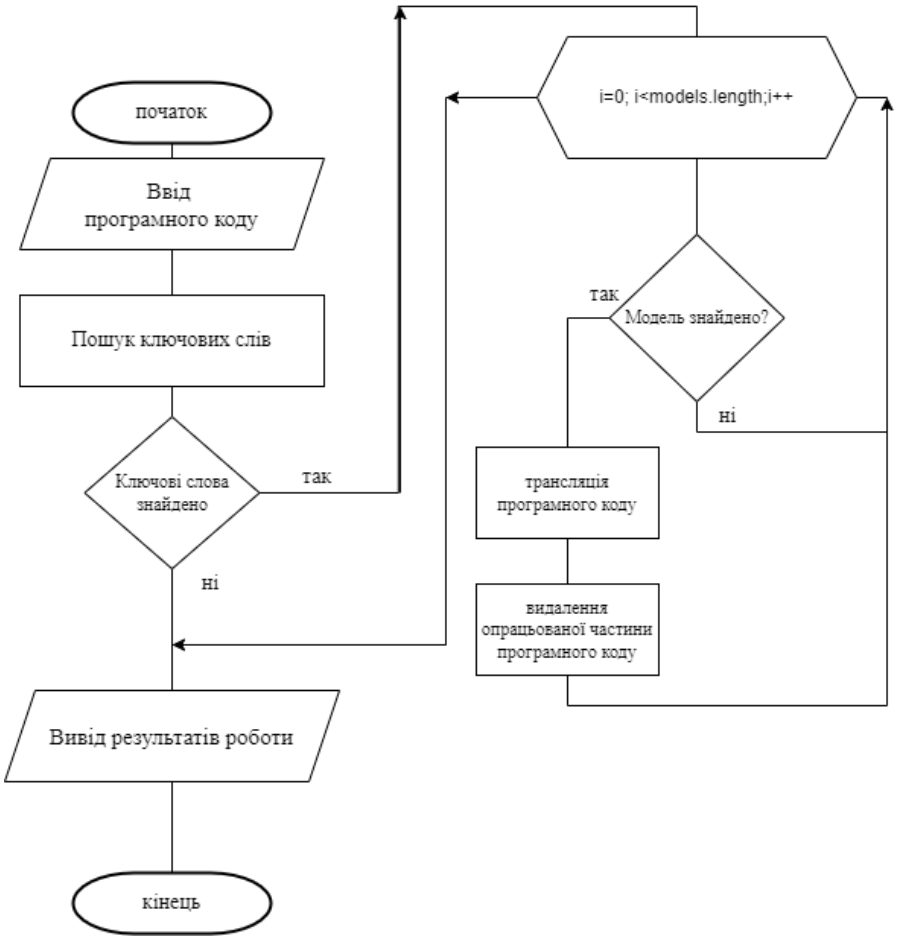
КП - Контроль пам'яті.

ДК - Дерево конструкцій.

СВР - Середовище виконання.



					123.КІ-41.11		
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*			
Розробив		Нечай Ю.І.			Діаграма прецендентів прогармної системи	*Літ.*	*Арк.* 5 / *Аркушів* 1
Перевірив		Когут І.Т.					
Н. Контр.							
Затвердив							



					123.КІ-41.11	
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*		
Розробив		Нечай Ю.І.			Блок-схема алгоритму трансляції коду	*Літ.* / *Арк.* 6 / *Аркушів* 1
Перевірив		Когут І.Т.				
Н. Контр.						
Затвердив						

Пояснювальна записка

до кваліфікаційної роботи

на тему:

«Алгоритм генерації оптимального машинного коду для мов високого рівня»

					123.КІ-41.11			
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*				
Розробив		Нечай Ю.І.			. Пояснювальна записка	*Літ.*	*Арк.*	*Аркушів*
Перевірив		Когут І.Т.					7	74
Н. Контр.								
Затвердив								

ЗМІСТ

ВСТУП

Актуальність роботи. Оптимізація є важливим аспектом у багатьох сферах життя, зокрема й у програмуванні. Оптимізований код забезпечує кращу продуктивність порівняно з попередніми версіями. Процес оптимізації включає написання або переписування коду для зменшення споживання пам'яті, дискового простору, пропускної здатності мережі або часу виконання програми. Оптимізація також допомагає зробити код коротшим і зрозумілішим для інших програмістів, особливо новачків. Простота і зрозумілість алгоритмів є ключем до створення оптимальних програм. Варто уникати оптимізації на рівні рядків, оскільки зміни на пізніх етапах можуть зробити такі зусилля марними. Вимоги в IT-індустрії постійно змінюються, тому код повинен бути готовий до майбутніх змін. Краще залишити можливість для зміни коду, ніж намагатися передбачити всі майбутні вимоги. Використання циклів у коді покращує його читабельність і сприяє оптимізації. Оптимізація є бажаною якістю в будь-якій мові програмування і має на меті забезпечити виконання вимог із мінімальним використанням ресурсів. Метою цієї роботи є розробка алгоритму для генерації оптимального програмного коду мовами високого рівня.

Для досягнення поставлених цілей були визначені наступні завдання: провести класифікацію програмних додатків за сферами їх використання, здійснити аналіз наявних типів мов програмування, розглянути інтегровані платформи для розробки програмного забезпечення, вивчити методи трансляції програмного коду на асемблерну мову, розробити алгоритм для конвертації коду з мов високого рівня на асемблер, створити програмний інструмент для трансляції коду, виконати його тестування та порівняти з іншими аналогічними програмами.

Основні результати дослідження включають глибокий аналіз та класифікацію методів трансляції коду з мов високого рівня на асемблерну мову, що дозволило визначити їхні сильні та слабкі сторони. Розроблено алгоритм для перетворення коду з мов високого рівня на асемблер, що сприяло оптимізації коду та підвищенню ефективності роботи програмного забезпечення.

Практичне значення дослідження полягає в теоретичному дослідженні та розробці програмного інструменту для конвертації коду з мов високого рівня на асемблер. Реалізація програмного рішення для генерації асемблерного коду та застосування моделей трансляції для різних типів алгоритмів підтвердила його ефективність та корисність.

РОЗДІЛ 1. ІНТЕГРОВАНІ СИСТЕМИ РОЗРОБКИ ТА ТЕСТУВАННЯ ПРОГРАМНИХ ДОДАТКІВ

1.1 Програмні додатки класифікація, структура та сфери застосування

Працюючи за комп’ютером, користувачі використовують різні програми, які допомагають спростити завдання та підвищити продуктивність. Від створення простих документів до перегляду веб-сторінок — усе це можливе завдяки програмному забезпеченню.

У нашій сучасній цифровій епосі ми оточені безліччю програм, кількість яких постійно зростає. Незалежно від операційної системи чи платформи, програмне забезпечення полегшує нам життя. Без програмного забезпечення комп'ютер був би лише сукупністю різних апаратних компонентів.

Програмне забезпечення, за визначенням, — «це комплекс даних, інструкцій, програм та правил, що спрямовують роботу комп’ютерної системи або іншого електронного пристрою, забезпечуючи виконання конкретних завдань».

Іншими словами, програмне забезпечення — це узагальнена назва для будь-якої програми чи скрипту, що працює на комп’ютерних пристроях і допомагає їм виконувати певні функції або обробляти дані. Воно є змінною частиною комп’ютерної системи, тоді як апаратне забезпечення є її постійною частиною.

Комп'ютерне програмне забезпечення, яке також відоме як комп’ютерна програма, являє собою набір інструкцій. Ці інструкції написані на різних мовах, зрозумілих комп'ютеру. Вони (їх зазвичай називають кодами) дозволяють давати команди комп'ютеру для виконання завдань і задоволення потреб користувачів.

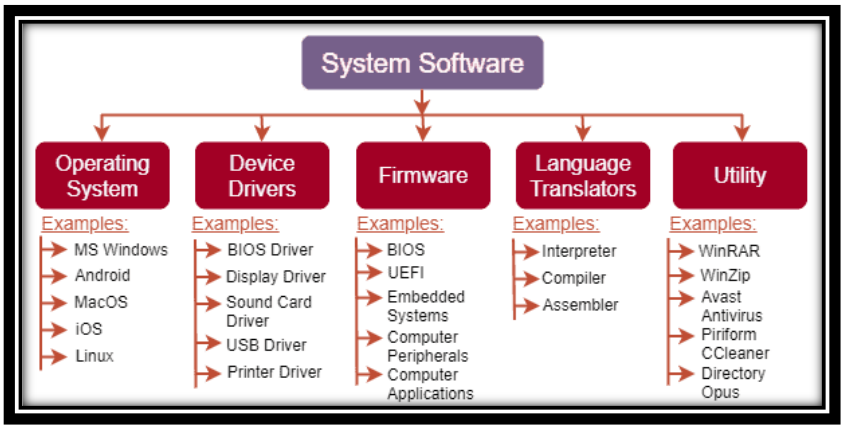


Рис. 1.1 – Категорії системного програмного забезпечення та їх приклади

Операційна система, часто скорочено ОС, є основним видом програмного забезпечення, яке управляє комп’ютерними ресурсами та послугами, забезпечуючи платформу для інших програм.

Хоча всі операційні системи працюють на основі програмного коду та інструкцій, більшість з них мають графічний інтерфейс користувача (GUI), який дозволяє користувачам легко взаємодіяти з системою без необхідності використовувати код. Кожен електронний пристрій, хай то буде настільний комп'ютер, ноутбук чи мобільний телефон, потребує встановленої операційної системи для функціонування і надання основних можливостей.

Операційна система є першою програмою, яка запускається при завантаженні комп’ютера, вона керує всіма аспектами роботи комп’ютера та ефективно управляє такими ресурсами, як процесор, пам'ять, пристрої зберігання даних (HDD або SSD), принтери тощо.

До найпопулярніших операційних систем належать MS-Windows, Android, macOS, iOS, Linux, Ubuntu, Unix та CentOS.

Драйвери пристроїв — це спеціалізоване програмне забезпечення, розроблене для керування певними апаратними компонентами комп'ютерної системи. Комп'ютери використовують різні апаратні пристрої, для кожного з яких потрібен відповідний драйвер для коректної роботи.

Найбільш поширені апаратні пристрої, для яких потрібні драйвери, включають монітори, відеокарти, звукові карти, жорсткі диски, принтери, миші та клавіатури. Деякі з цих пристроїв вимагають ручної інсталяції драйверів користувачем, тоді як операційна система автоматично встановлює підтримувані драйвери для інших. Існують два основні типи драйверів пристроїв: драйвери ядра та драйвери користувацького рівня. Прикладами популярних драйверів є драйвер BIOS, драйвер материнської плати та драйвер дисплея.

Мікропрограмне забезпечення, або постійне програмне забезпечення, зберігається в пам’яті материнської плати або в ПЗУ (постійній пам’яті) комп’ютерної системи. Проте насправді воно не є постійним, оскільки більшість

сучасних комп’ютерів дозволяють користувачам оновлювати мікропрограму за допомогою спеціального програмного забезпечення.

Мікропрограмне забезпечення, як і інші види програмного забезпечення, містить набір інструкцій. Основною функцією мікропрограми є перевірка системи на наявність помилок та забезпечення належної роботи всіх апаратних пристроїв при увімкненні комп’ютера. Якщо все працює правильно, мікропрограма виводить систему зі стану сну та передає керування операційній системі.

Перекладачі мов програмування відіграють важливу роль у перетворенні коду мов високого рівня (людинозрозумілого коду) на машинний код і навпаки. Це перетворення здійснюється за допомогою трансляторів мов програмування або компіляторів.

Транслятори мов програмування конвертують комп'ютерні програми, написані мовами високого рівня, такими як Java, C, C++, Python, у команди, які можуть бути зрозумілими машинам як об’єктний або машинний код. Окрім спрощення коду, перекладачі мов програмування також виконують функції з керування сховищем даних, генерування діагностичних звітів, виявлення та виправлення системних помилок під час виконання програми.

Утиліти

Утиліти — це допоміжні програми, створені для аналізу, оптимізації, налаштування та обслуговування комп’ютерної системи або її компонентів. Вони забезпечують безперебійну роботу комп’ютера, контролюючи стан операційної системи та пропонуючи або автоматично вносячи зміни для покращення продуктивності. Приклади утиліт включають антивірусні програми, засоби очищення диска, оптимізатори продуктивності, дефрагментатори, програми для стиснення файлів та інші.

Прикладне програмне забезпечення

Прикладне програмне забезпечення складається з програм та інструкцій, що виконують конкретні завдання на комп’ютері. Воно розроблене для задоволення специфічних потреб користувачів чи середовищ і завантажується та інсталюється вручну, не пов'язане з основними функціями системи.

Прикладне програмне забезпечення зазвичай працює у зручному інтерфейсі, що полегшує його використання. Однак для його функціонування потрібна система, яка забезпечується операційним середовищем. Наприклад, для роботи веб-браузера, такого як Google Chrome, потрібна встановлена операційна система. На відміну від системного програмного забезпечення, прикладне ПЗ не є обов'язковим для роботи системи, але допомагає виконувати різноманітні завдання.

Основні характеристики прикладного програмного забезпечення:

1. Призначене для виконання конкретних завдань, таких як редагування зображень, обробка текстів, ігри тощо.
2. Зазвичай має великий розмір та займає більше місця на жорсткому диску.
3. Розробляється на мовах високого рівня.
4. Висока взаємодія з користувачами, що робить його зручним у використанні.
5. Легше розробляється порівняно із системним програмним забезпеченням.

Типи прикладного програмного забезпечення представлені на рисунку 1.2.

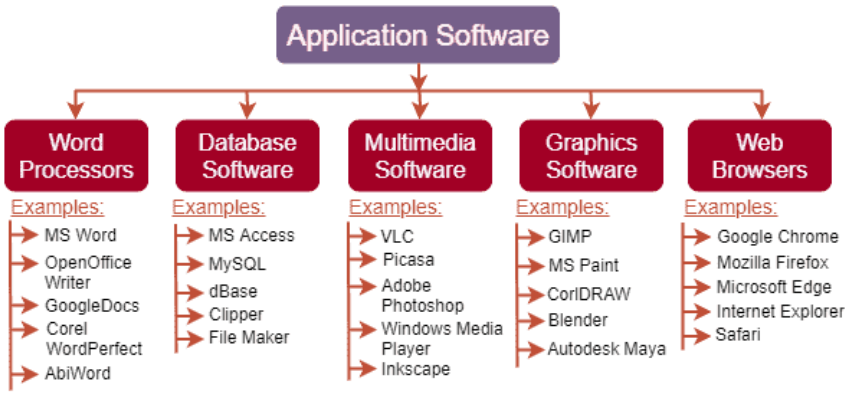


Рис. 1.2 – Приклади прикладного програмного забезпечення

Текстові процесори – це програми для обробки текстів, призначені для створення документів на комп'ютерах або інших електронних пристроях. Крім того, вони дозволяють редагувати, форматувати та друкувати документи. Популярними прикладами текстових процесорів є MS Word, Google Docs, OpenOffice Writer та інші.

Програмне забезпечення для баз даних: ці програми створені для створення та управління базами даних, також відомі як системи управління базами даних (СУБД). Програмне забезпечення для баз даних відіграє важливу роль в організації даних на комп'ютерах та серверах. Прикладами таких програм є MS Access, MySQL, dBase.

Мультимедійне програмне забезпечення дані програми використовуються для роботи з мультимедійними файлами, такими як аудіо- та відеофайли. Вони дозволяють відтворювати, створювати та редагувати мультимедійний контент, а також виконувати завдання, пов'язані з графікою та анімацією.

Серед популярних програм: Windows Media Player, Windows Movie Maker, Adobe Photoshop.

Графічне програмне забезпечення призначене для роботи з графічними зображеннями. Ці програми допомагають створювати логотипи, редагувати зображення та вносити зміни у візуальні дані, пропонуючи широкий набір інструментів і функцій для створення ілюстрацій.

Веб-браузери: програми для перегляду веб-сторінок та пошуку інформації в Інтернеті. Вони дозволяють користувачам отримувати доступ до веб-сайтів і ресурсів в Інтернеті. Більшість комп'ютерів та пристроїв постачаються з попередньо встановленим браузером, але користувачі завжди можуть встановити додаткові браузери, такі як Google Chrome, Firefox або Safari.

Навчальне та довідкове програмне забезпечення – ці програми створені для допомоги користувачам у вивченні конкретних тем або предметів, доступні як в Інтернеті, так і в офлайні. Вони часто називаються освітнім або академічним програмним забезпеченням і можуть бути як безкоштовними, так і платними.

Спеціалізоване програмне забезпечення, також відоме як програми спеціального призначення, розроблене для виконання конкретних завдань або задоволення потреб певних організацій. Це включає системи бронювання залізничних квитків, системи обліку рахунків та системи бронювання авіаквитків.

Програмне забезпечення класифікується за доступністю та можливістю спільного використання:

- безкоштовне програмне забезпечення;
- умовно-безкоштовне програмне забезпечення;
- програмне забезпечення з відкритим кодом;
- програмне забезпечення з закритим кодом;
- проміжне програмне забезпечення.

З розвитком комп'ютерів, автоматизації та робототехніки програмування стало необхідним для управління цими системами. Мови програмування поділяються на дві основні категорії: мови низького рівня та мови високого рівня.

1.2. Мови програмування та їх класифікація

Мови низького рівня близькі до машинної мови та використовуються для написання програм, що безпосередньо взаємодіють з апаратним забезпеченням. Вони ефективні з точки зору пам'яті та швидкості, але складні для розуміння і використання програмістами.

Основні категорії:

- **Машинна мова**: набір двійкових команд, виконуваних комп’ютером.
- **Мова асемблера**: використовує мнемотехнічні коди замість двійкових послідовностей.

Мови високого рівня більш схожі на людську мову і забезпечують вищий рівень абстракції від машинної мови. Вони легкі для кодування, налагодження та обслуговування. Програми на цих мовах потребують компіляції або інтерпретації для виконання на комп'ютері. Приклади: Python, Java, C++.

Мови програмування можуть бути інтерпретованими або скомпільованими. Інтерпретовані мови виконуються безпосередньо інтерпретатором, який перекладає код на ходу. Скомпільовані мови спочатку перетворюються компілятором у машинний код, який потім виконується комп’ютером.

Переваги інтерпретованих мов:

- Простота у вивченні та використанні;
- Швидке редагування та виконання коду.

Недоліки інтерпретованих мов:

- Повільне виконання;
- Обмежені можливості оптимізації.

Переваги компільованих мов:

- Швидке виконання;
- Оптимізація для цільового обладнання.

Недоліки компільованих мов:

- Потреба в компіляторі;
- Повільне редагування та розгортання коду.

Таблиця 1.1 представляє зіставлення мов програмування різних рівнів.

№	Мови високого рівня	Мови низького рівня
1	Вони швидші за мову високого рівня.	Вони порівняно повільніші.
2	Мови низького рівня ефективно використовують пам'ять.	Мови високого рівня не є ефективними для пам'яті.
3	Мови низького рівня важко вивчити.	Мови високого рівня легко вивчати.
4	Програмування на низькому рівні вимагає додаткові знання комп'ютера архітектура.	Програмування на високому рівні не потребує додаткові знання комп'ютера архітектура.
5	Вони залежать від машини і не залежать портативний.	Вони незалежні від машини та портативні.
6	Вони забезпечують менше або зовсім не абстрагуються від обладнання.	Вони забезпечують високу абстракцію від обладнання.
7	Вони більш схильні до помилок.	Вони менш схильні до помилок.
8	Налагодження та обслуговування складні.	Налагодження та обслуговування є порівняльними легше.
9	Вони зазвичай використовуються для розробки системне програмне забезпечення (операційні системи) і вбудовані програми.	Їх використовують для розробки різноманітних програми, такі як – настільні програми, веб-сайти, мобільне програмне забезпечення тощо.

1.3 Інтегровані середовища розробки

Інтегроване середовище розробки, або IDE, являє собою програмну платформу, яка полегшує створення інших програм, забезпечуючи середовище для написання, налагодження та тестування коду, а також надаючи інструменти, що зменшують зусилля розробників.

До появи інтегрованих середовищ розробки у 90-х роках, програмістам доводилося писати код у текстових редакторах, як-от Notepad, а потім окремо компілювати його. Після цього потрібно було вручну виправляти помилки,

повертаючись до текстового редактора, що робило процес розробки програмного забезпечення громіздким та неефективним, оскільки кодування, компіляція та налагодження виконувалися в окремих етапах.

Впровадження IDE наприкінці 1980-х змінило цю ситуацію. Компанія Softlab Munich створила перше у світі інтегроване середовище розробки під назвою Maestro I, яке стало популярним серед тисяч програмістів по всьому світу. Пізніше Microsoft розробила свою власну IDE Visual Basic (VB), яка набула великої популярності. З Visual Basic IDE стали невід'ємною частиною сучасного циклу розробки та DevOps.

Щоб зрозуміти, що таке інтегроване середовище розробки, розглянемо критичні компоненти, які забезпечують його функціонування (рис. 1.4).

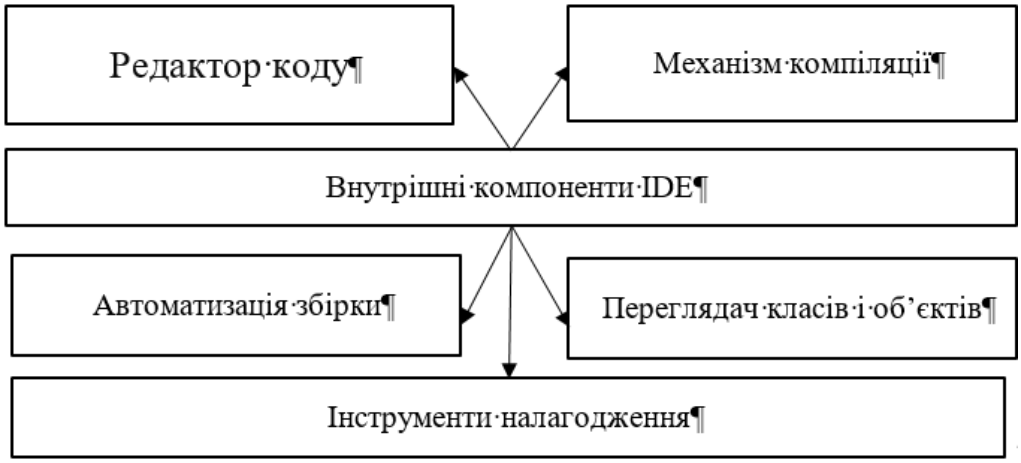


Рис. 1.4 – Внутрішні основні компоненти IDE

Редактор коду – це ключовий елемент інтегрованого середовища розробки (IDE), в якому програмісти пишуть програмний код. На перший погляд він може виглядати як звичайний текстовий редактор, але насправді містить численні функції, що полегшують процес написання коду. Наприклад, редактор IDE може автоматично передбачати ваші дії та виконувати команди.

Механізм компіляції: IDE включає в себе вбудований компілятор, що дозволяє запускати код безпосередньо в середовищі для його перевірки. Результати компіляції зазвичай відображаються в окремому вікні в межах тієї ж платформи IDE, що дозволяє легко переключатися між редактором коду та компілятором.

Інструменти налагодження: IDE надає основні засоби тестування для виявлення помилок у програмному забезпеченні на рівні вихідного коду. Хоча ці інструменти не можуть виявити логічні помилки, вони допомагають виявити та виправити помилки кодування, такі як некоректні команди, відсутні змінні, синтаксичні помилки тощо. IDE підсвічує конкретне місце помилки, що дозволяє розробникам швидко виправляти код.

Засіб перегляду класів і об'єктів: при роботі з об'єктно-орієнтованим програмуванням (ООП) IDE може включати інструменти для перегляду об'єктів і класів у програмі. Цей інструмент навіть здатний візуалізувати ієрархії класів, що полегшує повторне використання об'єктів та покращує продуктивність.

Автоматизація збірки: автоматизація збірки полягає в підготовці програмного коду до виконання. Деякі IDE мають вбудовані інструменти для автоматизації збірки, які допомагають створювати, упаковувати та розгортати код.

Використання IDE – це досить простий процес, який не потребує від розробника специфічних навичок, окрім знань програмування та розуміння базових функцій платформи. Кожна IDE має свої особливості, окрім п'яти основних компонентів, тому важливо знати найкращі практики у галузі.

Інтегровані середовища розробки існують вже багато років. Вони еволюціонували від простих платформ для налагодження та тестування до комплексних програмних пакетів, які значно розширюють можливості розробників. Саме інтегровані компоненти вирізняють найкраще програмне забезпечення IDE від звичайних редакторів коду.

Ось кілька найкращих рішень для кодування, які є одночасно простими у використанні та багатофункціональними:

1. Visual Studio Code - популярний редактор коду, що підтримує безліч мов програмування і має широкий спектр розширень.
2. IntelliJ IDEA - потужне середовище для розробки на Java, яке пропонує розширені можливості для налагодження та тестування.
3. Eclipse - широко використовуване IDE для розробки на Java, що має безліч плагінів для розширення функціональності.

4. PyCharm - спеціалізоване середовище для розробки на Python, яке включає в себе інструменти для тестування, налагодження та аналізу коду.
5. NetBeans - багатоплатформне IDE, що підтримує розробку на різних мовах, таких як Java, PHP, і C++.

   Microsoft Visual Studio — це потужне інтегроване середовище розробки (IDE), призначене для створення різноманітних програм, включаючи ті, що мають графічний інтерфейс користувача та консольні додатки. Ця платформа також підтримує розробку веб-сайтів, веб-додатків, онлайн-сервісів, а також програм на базі Windows Forms і WPF. Однією з ключових особливостей Visual Studio є його редактор коду, який оснащений функцією IntelliSense для автозавершення коду та інструментами для рефакторингу коду. Крім того, Visual Studio має вбудовані інструменти, такі як інтегрований налагоджувач, аналізатор коду, дизайнер GUI, засоби для веб-розробки, дизайнер класів і дизайнер схем баз даних. Ці інструменти значно спрощують процес розробки програмного забезпечення, забезпечуючи розробникам все необхідне для ефективної роботи.

   Eclipse є одним із найпоширеніших інтегрованих середовищ розробки (IDE) для мови програмування Java. Це настільний додаток, який може працювати на багатьох платформах, що робить його надзвичайно гнучким і зручним для розробників.Однією з найпривабливіших особливостей Eclipse є його користувацький інтерфейс, який включає підтримку функції перетягування елементів.

   Крім того, Eclipse дозволяє виконувати статичний аналіз коду, що допомагає виявляти потенційні проблеми на ранніх стадіях розробки. Програма також підтримує налагодження та аналіз, що робить її потужним інструментом для будь-якого розробника.

   NetBeans — це безкоштовне інтегроване середовище розробки з відкритим кодом. NetBeans відзначається інтуїтивно зрозумілим інтерфейсом з підтримкою перетягування елементів, а також наявністю багатьох корисних шаблонів проектів. Це робить його ідеальним вибором як для налаштування

існуючих проектів, так і для розробки нових з нуля. Хоча NetBeans зазвичай використовується для розробки програм на мові Java, існують пакети, які дозволяють підтримувати й інші мови програмування, що розширює можливості цього середовища. Завдяки своїй відкритій архітектурі та активному співтовариству розробників, NetBeans залишається популярним вибором для багатьох програмістів у всьому світі.

Хмарні IDE стають дедалі популярнішими. Їхні функції швидко розширюються, і провідним виробникам програмного забезпечення доведеться пропонувати їх, щоб залишатися конкурентоспроможними. Хмарні IDE дозволяють розробникам працювати з будь-якого місця, наприклад, за допомогою таких сервісів, як Nitrous і AWS.

Amethyst 2 є інтегрованим середовищем розробки, створеним у 2006 році Х'ю Коллінгборном і Дермотом Хоганом. Amethyst 2 доступний у двох варіантах: платна версія Amethyst Ultimate та безкоштовна Amethyst Personal. Amethyst Ultimate пропонує широкий спектр інструментів, таких як Amethyst Designer, налагоджувач, рефакторинг коду тощо. Android Studio — це офіційна IDE для розробки додатків на платформі Android, яка надає прості та зручні інструменти для створення додатків для різних пристроїв Android. Поєднання настроюваної системи збірки та швидкого розгортання дозволяє розробникам створювати якісні додатки та швидко їх постачати. Крім того, вона є повністю безкоштовною.

Cloud9 — це хмарне середовище розробки, яке підтримує такі мови, як Ruby, Python, Node.js та інші. Існують спеціалізовані IDE для розробників, які працюють з однією мовою. Наприклад, CodeLite і C-Free для C/C++, Jikes і Jcreator для Java, Idle для Python і RubyMine для Ruby/Rails. Однак зараз спостерігається перехід до багатомовних IDE через їх універсальність.

Розробники можуть також використовувати плагіни для підтримки додаткових мов. Flycheck, наприклад, є плагіном для перевірки синтаксису у GNU Emacs 24, який підтримує 39 різних мов.

Багатомовні IDE підтримують безліч мов програмування, таких як Perl, C, C++, Ruby, Python, Java і PHP. Це безкоштовний редактор з відкритим кодом для

багатьох програмних фреймворків. Спочатку він був розроблений як середовище для Java, але згодом був розширений завдяки плагінам. Це інтегроване середовище розробки знаходиться під контролем консорціуму Eclipse.org. Середовища розробки HTML-додатків є одними з найпоширеніших IDE для створення веб-програмного забезпечення як послуги (SaaS).

C-Free дозволяє змінювати, розробляти та налагоджувати програми в одному зручному середовищі, використовуючи вбудовані інструменти та функції, які покращують ваші можливості. C-Free також компактний, його розмір установки становить лише 14 МБ, а без упаковки — 80 МБ.

IntelliJ IDEA — це IDE, орієнтована на Java, яка також підтримує мови Kotlin, Groovy та інші мови на основі JVM. Вона була створена компанією JetBrains і пропонується в двох варіантах: комерційна версія і версія для спільноти з відкритим кодом.

Версія Ultimate IDE забезпечує підключення до систем контролю версій, баз даних та інструментів для збирання і пакування. Існують також спеціальні IDE для мобільних розробок, такі як Titanium Mobile від Appcelerator і PhoneGap. Багато IDE, особливо багатомовні, підтримують плагіни для мобільних розробок, наприклад, Eclipse має подібні функції.

Xcode також підтримує мови програмування Swift і Objective-C, а також фреймворки Cocoa Touch і Cocoa API. Це інтегроване середовище розробки призначене для створення додатків для Mac та iOS і включає в себе симулятор для iPad/iPhone і дизайнер графічного інтерфейсу.

Розробники використовують різні інструменти на етапах розробки, створення та тестування програмного забезпечення. Текстові редактори, бібліотеки коду, інструменти для відстеження помилок, компілятори та платформи для тестування є одними з найпоширеніших інструментів. Без IDE розробникам доводиться вручну обирати, встановлювати, інтегрувати та керувати цими інструментами.

Інтегроване середовище розробки об'єднує кілька технологій, що стосуються розробки, в одному інтерфейсі. Коли всі необхідні інструменти доступні в одному

місці, розробникам не потрібно витрачати багато часу на навчання їх окремому використанню. Це особливо корисно для новачків, які можуть використовувати IDE для вивчення основних процесів та інструментів.

1.4 Постановка задач дослідження

У рамках цього дослідження було проведено аналіз і класифікацію програмних додатків та їх областей застосування. Класифікація ґрунтувалася на функціональних характеристиках кожного програмного продукту. Було досліджено і класифіковано сучасні мови програмування, що дозволило визначити мову для подальших досліджень.

Проведено аналіз і класифікацію інтегрованих середовищ розробки програмного забезпечення, визначено їх основні архітектурні і функціональні рішення, а також структури даних, що використовуються для зберігання, обробки та трансляції програмного коду з мов високого рівня на асемблер. Для досягнення цих цілей потрібно виконати наступні завдання: - Класифікувати програмні додатки за їхніми областями застосування, а також проаналізувати і класифікувати існуючі типи мов програмування. - Провести аналітичний огляд інтегрованих середовищ розробки програмного забезпечення. - Аналізувати існуючі алгоритми трансляції програмного коду на асемблер. - Розробити алгоритм трансляції програмного коду, написаного на мовах високого рівня, на асемблер. - Реалізувати програмне забезпечення для трансляції програмного коду з мов високого рівня на асемблер, провести його тестування і порівняти з аналогічними програмами.

1.5 Висновки до розділу

Було проведено ґрунтовний аналіз і класифікацію програмних додатків на основі їх функціональних можливостей. Це дало змогу виявити ключові архітектурні особливості та формати даних, які використовуються під час створення виконуваних файлів.

Також була здійснена класифікація мов програмування високого і низького рівня, що дозволило визначити мову C++ як основну для подальших досліджень. Ця класифікація допомогла зрозуміти, чому C++ є важливою для вивчення та оптимізації процесу трансляції коду.

Крім цього, проведено аналіз сучасних інтегрованих середовищ розробки програмних додатків. Аналіз зосереджувався на дослідженні принципів трансляції мов високого рівня на асемблер і механізмах створення виконуваних файлів. Це допомогло зрозуміти, як сучасні інструменти розробки обробляють код і перетворюють його у виконувані програми.

РОЗДІЛ 2. АЛГОРИТМИ ТРАНСЛЯЦІЇ КОДІВ З МОВ ВИСОКОГО РІВНЯ НА МОВУ АСЕМБЛЕР

2.1 Структура та особливості мови високого рівня C++

C++ є середньорівневою мовою програмування, яка має свої специфічні переваги та недоліки у порівнянні з іншими мовами високого рівня. Розробники, які працюють з C++, зазвичай мають досвід роботи з UNIX і Linux або з платформами та апаратними пристроями. Ця мова дозволяє зробити всю вбудовану та об'єктно-орієнтовану функціональність універсальною, що робить її унікальною порівняно зі структурованими мовами. Тому кожен інженер повинен ознайомитися з C++ та стандартами кодування та парадигмами, які вона пропонує.

У сучасному світі володіння різними технологіями є важливим, а мови програмування є важливим інструментом для розробки програмного забезпечення. Однією з таких мов є C++, що дозволяє працювати на різних рівнях з різними концепціями UNIX або Windows. Вивчення цієї мови має багато переваг, які можуть бути корисні для будь-якого професіонала.

Розробники, які працюють з C++, повинні розуміти апаратні компоненти системного рівня та концепції програмування, необхідні для розробки програми або продукту. Вони повинні мати знання об'єктно-орієнтованої мови, яка допомагає зрозуміти логіку і концепцію кожної мови програмування. Вони також повинні розуміти поняття управління пам'яттю, створювати віртуальні таблиці, віртуальні таблиці відображення тощо. Крім того, вони повинні вміти розрізняти компілятори, навантажувачі, динамічні лінкери, класи зберігання, змінні типи, області застосування тощо. Серед ключових особливостей C++ можна виділити наступні, які значно полегшують процес написання коду (див. рис. 2.1).

Розробка прикладного програмного забезпечення включає створення практично всіх можливих типів операційних систем. Як зазначено вище, це стосується всіх платформ, таких як Windows, macOS і Linux. Більшість веб-браузерів створюються за допомогою цієї середньорівневої мови програмування. Вона також використовується при розробці популярних баз даних, таких як MySQL.

Постійне програмування: Окрім цього, багато інших мов кодування було розроблено на основі C++, включаючи C#, Java, Python і JavaScript.

Швидкодійність: Її продуктивність дозволяє вченим оперативніше виконувати свої дослідження.

Розробка ігрових та вбудованих систем: Ця потужна мова підтримує створення ігор та вбудованих систем, таких як передові медичні пристрої та системи автоматизованого проектування (САПР).

Серед переваг C++ можна відзначити кілька ключових аспектів.

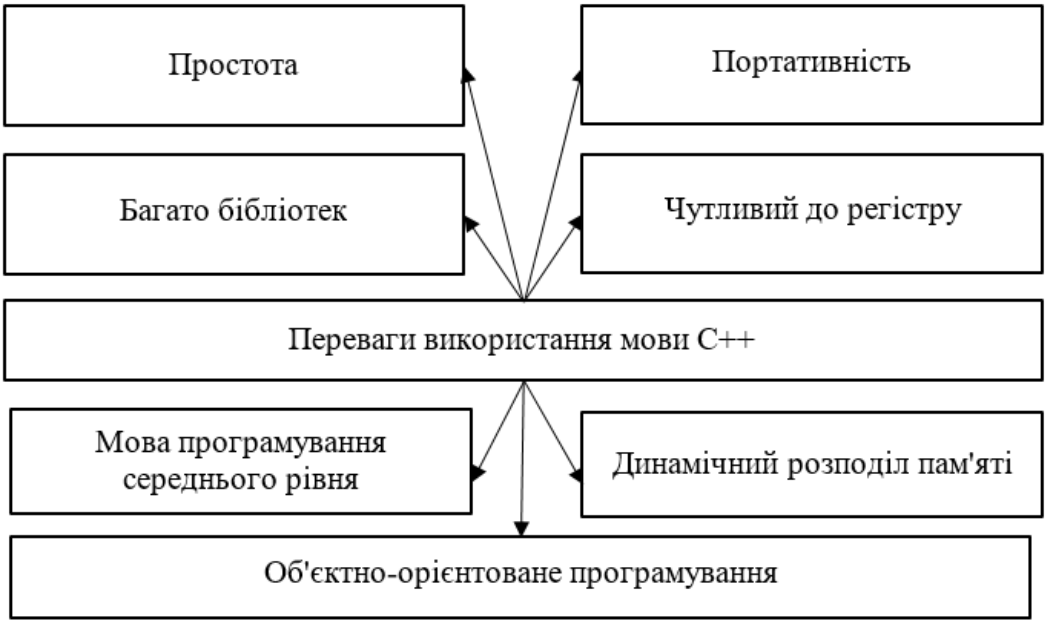


Рис. 2.1 – Основні переваги використання мови C++

Легкість освоєння: C++ є однією з найпростіших для вивчення мов програмування. Вона зрозуміла та доступна для вивчення, оскільки базується на мові C. Основні принципи C++ можуть бути використані в будь-якому проекті, що дозволяє розбивати складні завдання на більш керовані частини.

Об'єктно-орієнтоване програмування. Цей підхід становить основу мови C++ і є одним з ключових факторів, що зумовили її популярність. У C++ всі об'єкти розглядаються як об'єкти, що відображає об'єктно-орієнтований підхід. Для виконання завдань використовуються об'єкти класів, а не лише функції. Функціональні можливості мови включають поліморфізм і успадкування, що

дозволяють об'єднувати функції та дані в єдиний блок, роблячи програми більш захищеними та ефективними.

Портативність. C++ не залежить від конкретної платформи і може працювати на різних пристроях з мінімальними або взагалі без змін. Код, написаний один раз, може бути використаний повторно для виконання відповідних функцій. Однак, не можна стверджувати, що C++ є абсолютно незалежним від платформи. Наприклад, код, розроблений під Linux, може потребувати адаптації для роботи під Windows, проте після перекомпіляції він працюватиме належним чином.

Динамічне керування пам'яттю. Завдяки підтримці покажчиків у мові C++ пам'ять можна розподіляти динамічно, а не статично, та звільняти в будь-який момент за допомогою функції free().

Рекурсія. Завдяки можливості багаторазового використання коду, можна викликати будь-яку функцію всередині іншої, що економить пам'ять і дозволяє уникнути повторення коду. Це є основним принципом для кожної функції.

Об'єктно-орієнтована структура C++ робить її потужнішою та швидшою за багато інших мов програмування. Особливо важливо, що використання бібліотеки C++ є однією з найбільших переваг у розробці програмного забезпечення.

Швидкодія. C++ відомий своєю високою швидкодією, яка перевершує більшість інших мов програмування. Паралельний запуск декількох кодів є одним з ключових методів оптимізації, що допомагає прискорити виконання. Це гарантує найкращу продуктивність, навіть при високому навантаженні на сервер.

Однак, не можна стверджувати, що C++ є повністю незалежним від платформи. Наприклад, код, написаний під Linux, може потребувати конвертації для роботи під Windows, але після перекомпіляції він буде працювати коректно.

Динамічний розподіл пам'яті. Завдяки підтримці покажчиків у мові C++ пам'ять можна легко розподіляти динамічно, а не статично, і звільняти в будь-який момент за допомогою функції free().

Рекурсія. Завдяки можливості багаторазового використання коду можна викликати будь-яку функцію всередині іншої, що економить пам'ять і дозволяє уникнути повторення коду. Це основний принцип для кожної функції.
C++ є об'єктно-орієнтованою мовою, а не процедурною. Її особливості роблять її швидшою і потужнішою за багато інших мов програмування.

Машинне навчання з використанням бібліотеки C++ також є однією з головних переваг цієї мови у процесі розробки. C++ відома своєю швидкістю, яка перевершує інші мови програмування.

Паралельне виконання коду є однією з ключових функцій, яка значно прискорює виконання завдань. Це забезпечує високу продуктивність навіть при значному навантаженні на сервер. На рівні апаратного забезпечення. Коли програмне забезпечення тісно пов'язане з апаратним забезпеченням і потребує низькорівневої підтримки, C++ забезпечує цю підтримку, оскільки ближче до апаратного забезпечення, ніж інші мови програмування.

Функціональність. Такі особливості C++, як наслідування, інкапсуляція та абстракція, роблять її дуже корисною для розробників програмного забезпечення. Поєднання цих переваг дає змогу створювати ефективні та якісні продукти. Ефективність C++ завжди на високому рівні. C++ підтримує перевантаження операторів, що дозволяє використовувати визначені користувачем оператори разом із перевантаженням функцій. В C++ реалізована обробка винятків.

Розробникам не потрібно явно визначати винятки для різних ситуацій, оскільки вони можуть користуватися вбудованими механізмами для перехоплення і відображення повідомлень користувачеві. Однак у разі потреби вони можуть створити власний клас для ефективної обробки винятків у C++. Розробники можуть визначити свій тип винятку залежно від вимог і специфіки проєкту. У C++ для обробки винятків використовуються ключові слова try, catch і throw. Усі види винятків можуть бути ефективно оброблені за допомогою механізмів обробки винятків у C++. Змінні можна оголошувати в будь-якій частині програми на C++, але перед використанням їх потрібно оголосити.

Стандартний додаток для операційної системи Windows включає кілька основних компонентів, які проілюстровані на рисунку 2.2. Ця послідовність дій є типовою і повторюється при кожному створенні віконного застосунку з графічним інтерфейсом. Програма забезпечує повну реакцію на всі дії користувачів, одночасно дотримуючись основних принципів взаємодії та обміну повідомленнями між операційною системою та програмним забезпеченням. Слід зазначити, що середовище програмування Windows передбачає можливість реагування на широкий спектр подій користувача, що дозволяє реалізувати різноманітні функції та забезпечити взаємодію з системою.

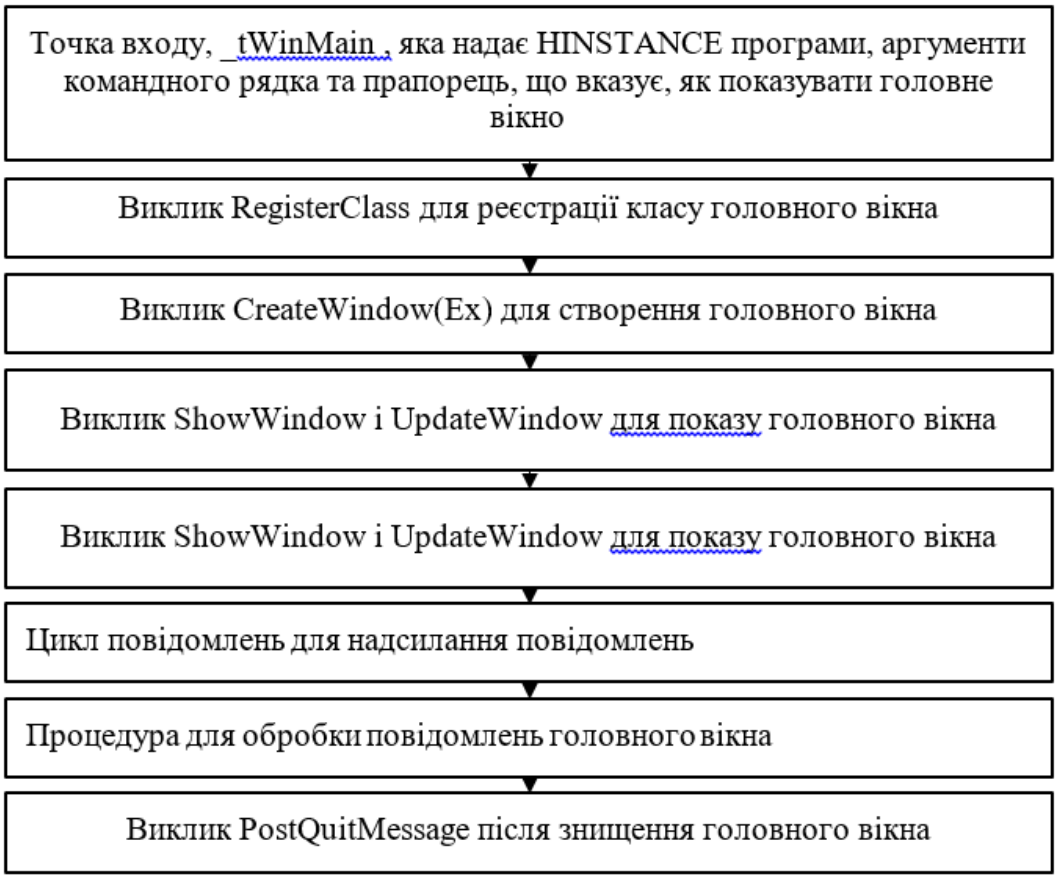


Рис. 2.2 – Архітектура програмного застосунку на мові C++ під операційну систему Windows

Усі додатки для Windows мають спільні вимоги, які можна виразити через виклики процедур Win32, як показано в прикладі. Однак, розробники на C++ часто бажають інкапсулювати ці виклики в методи класів при моделюванні базової об'єктної моделі. User32, яка є віконною частиною Win32 API, чітко реалізує основну об'єктну модель, що складається з класів Window (відображених

структурою WNDCLASSEX), об'єктів Window (відображених HWND) і методів (викликів до WndProc). Для розробника на C++, який прагне уникнути розриву між бажаною об'єктною моделлю та моделлю User32, ATL надає невеликий набір віконних класів, як показано на рисунку 2.3.

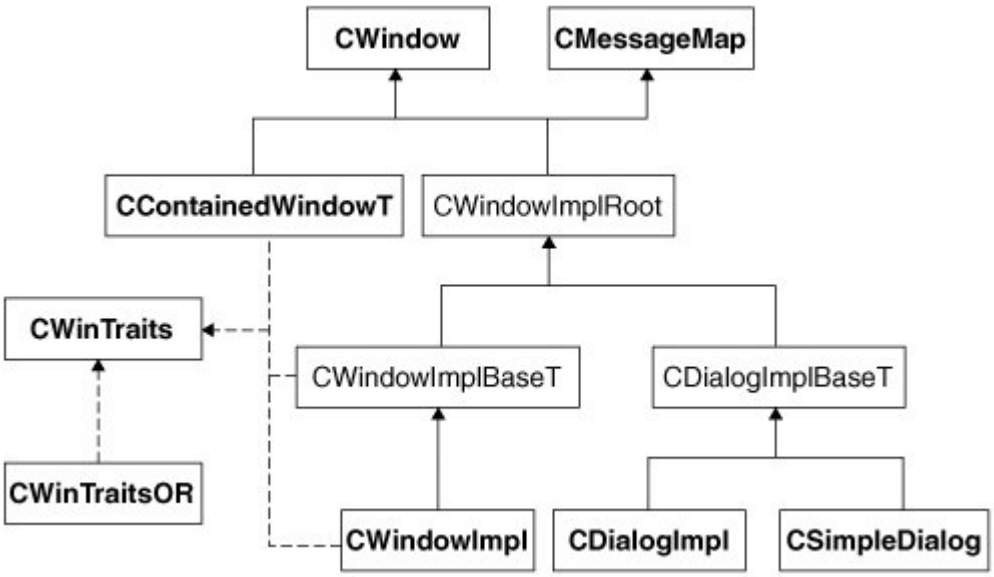


Рис. 2.3 – Набір віконних класів для операційної системи Windows

Найважливішими є класи CWindow, CWindowImpl, CWinTrains, CWinTrainsOR, CDialogImpl, CSimpleDialog та CContainedWindow. Інші, такі як CWindowImplRoot, CWindowImplBaseT і CDialogImplBaseT, є допоміжними класами для відокремлення параметризованого коду від інваріантного. Це розділення допомагає зменшити об'єм коду, пов'язаного з шаблонами, але ці класи не є ключовими компонентами віконних класів ATL.

Способи використання мови асемблера з C/C++: Використовуйте окремі зібрані модулі коду, пов'язані з компільованими модулями C/C++. Використовуйте асемблерні та постійні змінні в коді C/C++. Мова асемблера інтегрується безпосередньо у вихідний код C/C++. Змінює асемблерний код, згенерований компілятором. Взаємодія C/C++ з функціями асемблера проста, якщо дотримуватися правил виклику та умов реєстрації. Код C/C++ може отримати доступ до змінних і функцій, визначених мовою асемблера, тоді як код асемблера може отримати доступ до змінних C/C++ і викликати функції C/C++.

Процедури переривань повинні зберігати всі регістри, які вони використовують. Коли функції C/C++ викликаються з мови асемблера, необхідно завантажити призначені регістри аргументами і відправити інші аргументи в стек. Функції повинні коректно повертати значення відповідно до їх декларацій у C/C++.

Уникайте використання секції .cinit у модулях збірки, окрім випадків, коли це потрібно для автоматичної ініціалізації глобальних змінних. Секція .cinit повинна містити виключно таблиці ініціалізації, оскільки порушення цього правила може призвести до непередбачуваних наслідків.

Для доступу до функцій або об'єктів C/C++ з мови асемблера їх потрібно оголосити за допомогою директив .ref або .global у модулі асемблера. Функції, визначені в C++, які будуть викликані з асемблера, слід оголошувати як extern «C» у файлі C++.

Аналогічно, функції, визначені в асемблері для виклику з C++, повинні бути прототиповані як extern «C» у C++. У наведеному прикладі функція C++ main() викликає асемблерну функцію asmfunc, яка додає аргумент до глобальної змінної gvar у C++ і повертає результат.

```
extern "C" {
extern int asmF (int num);
int gNam = 0;
}
void main ()
{
    int IntegerNum = 5;
    IntegerNum = asmF (IntegerNum);}
```

Програмний код створено за допомогою асемблерної мови:

```
        .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
```

Код програми, написаний мовою асемблера, іноді може використовувати змінні або константи, визначені на C/C++. Є кілька способів це зробити, залежно від того, де і як визначено елемент: змінна в секції .bss, змінна за межами секції .bss, або символ зв'язування.

Доступ до змінних із секції .bss або .usect можна здійснити наступним чином:

1) Для оголошення змінної використовуйте директиву .bss або .usect.

2) Використовуйте директиву .def або .global, щоб зробити оголошення зовнішнім.

3) Використовуйте відповідну назву посилання на асемблерній мові.

У C/C++ оголосіть змінну з ключовим словом extern і звертайтеся до неї як зазвичай.

```
.bss     var,4,4
.global  var
```

Програма на C для доступа до мови асемблера:

```
extern int var;
var = 1;
```

У мовах C/C++ та асемблера таблиця символів містить адреси значень змінних. Коли ви отримуєте доступ до змінної за її назвою в C/C++, компілятор використовує таблицю символів для отримання значення.

Однак, для констант таблиця символів містить безпосереднє значення константи. Компілятор не може визначити, які елементи таблиці символів є адресами, а які – значеннями. Якщо ви отримуєте доступ до асемблерної константи за її іменем, компілятор може спробувати використати значення з таблиці символів як адресу для отримання значення. Щоб уникнути цього, використовуйте оператор & (адреса), щоб отримати значення (_symval). Наприклад, якщо x є константою мови асемблера, її значення в C/C++ буде &x. Тут наведено додаткові приклади використання _symval.

Щоб полегшити використання цих символів у вашій програмі, можна використовувати приведення типів і директиви #defines.

```
Extern int table_size;
For (I=0; i<TABLE_SIZE; ++I)
```

Переклад даного коду на мову асемблер

```
_table_size .set10000
            .global _table_size
```

Оскільки це стосується лише значення символу, збереженого в таблиці символів, оголошений тип символу не має значення. Приклад використовує int, але аналогічно можна звертатися до символів, визначених лінкером.

У C/C++ можна за допомогою оператора asm вставити асемблерний код у файл умовної мови асемблера, створений компілятором. Рядок asm тверджень додає послідовні інструкції асемблера у вихідний файл компілятора без проміжного коду. Оператор asm корисний для вставки коментарів у вихідний код компілятора.

Просто вставте рядок коду збірки, завершений крапкою з комою (;), як показано нижче:

*Asm ("; *** це коментар на мові асемблера");*

C++ є найбільш ефективним і потужним мовою завдяки своїй функціональності високого рівня. Чотири основні принципи C++: абстракція, інкапсуляція, успадкування та поліморфізм. Ці принципи можуть окремо виконувати динамічні завдання та реалізовувати функції в будь-якому процесі розробки програмного забезпечення, і разом вони є одними з найпотужніших в світі програмування.

2.2 Алгоритми трансляції програмних кодів на мову асемблер

Машинний код є найбільш базовим рівнем програмування, де команди безпосередньо виконуються центральним процесором комп'ютера. Важливо розуміти, що кожен процесор або лінія процесорів має свій унікальний набір інструкцій машинного коду. Ці інструкції зазвичай представлені числовими значеннями, закодованими у двійковому форматі, наприклад, як 10101001 01000000 Зазвичай інструкція машинного коду складається з двох частин:

Оператор (OP код), вбудований в інструкцію для виконання процесором. Наприклад, 10101001 може вказувати процесору ініціювати завантаження даних з пам'яті. Операнд часто представляє адресу пам'яті, з якої дані зчитуються або записуються, залежно від оператора. Наприклад, 01000000 може означати адресу 64 в пам'яті.

Деякі ОР-коди, як ті, що відповідають за завершення програми, не вимагають операнда. Пряме написання програм на машинному коді було б складним і схильним до помилок, оскільки вимагало б ручного розрахунку всіх числових адрес для розгалужень і інструкцій позиціонування даних. Мова асемблера виникла як рішення цієї проблеми, що відкрило шлях до створення більш складних мов програмування високого рівня.

З огляду на те, що процесор виключно розуміє інструкції машинного коду, будь-який вихідний код, написаний на альтернативних мовах програмування, повинен пройти перетворення в машинний код перед виконанням. Це перетворення виконується спеціалізованими утилітами, відомими як компілятори, перекладачі або асемблери.

Крім того, завдяки тому, що кожна інструкція машинного коду представлена виключно двійковими бітовими шаблонами, розшифрування або побудова програмного забезпечення безпосередньо через машинний код є надзвичайно складним завданням для людей. Мова асемблера виникла як початкова спроба вирішити цю проблему (згодом послідувала поява все більш складних мов високого програмувального рівня).

Асемблер — це низькорівнева мова програмування, яка використовує мнемонічні позначення для безпосереднього представлення інструкцій машинного коду. Для позначення адрес пам'яті, до яких спрямовуються гілки і дані, застосовуються спеціальні мітки.

Асемблерні інструкції перетворюються на машинний код за допомогою асемблера. Між асемблерними інструкціями та інструкціями машинного коду існує майже точна відповідність. Це означає, що програма, створена за допомогою асемблера, є дуже ефективною, і зазвичай вимагає менше пам’яті та працює швидше, ніж програма, згенерована компілятором з високорівневої мови.

У таблиці 2.1 наведено приклади перетворення асемблерних мнемонік на 16-розрядний машинний код. Кожна інструкція машинного коду містить оператор (операційний код) та операнд.

Таблиця 2.1 - Приклади трансляції з асемблерної мови на машинний код

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	Додайте в накопичувач дані, що зберігаються в пам'яті за адресою 64

Переваги програмування на мові асемблера:

Програмування на мові асемблера дозволяє створювати програми, які швидко виконуються, оскільки існує однозначна відповідність між асемблерними командами та машинним кодом. Це забезпечує високу ефективність програмного забезпечення. Завдяки чіткому перетворенню асемблерних інструкцій на машинний код, процес трансляції відбувається дуже швидко.

Асемблерний код легше зрозуміти порівняно з машинним, оскільки він використовує мнемонічні позначення.

Також можливо використовувати мітки для позначення адрес пам'яті, що спрощує керування адресами у разі додавання або видалення інструкцій, інакше всі посилання на пам'ять у програмі потрібно було б перераховувати вручну.

Недоліки програмування на мові асемблера:

Програмування на мові асемблера ускладнюється тим, що для різних процесорів можуть знадобитися різні версії асемблерної мови, що ускладнює перенесення програм між різними процесорними архітектурами. Асемблерні програми часто створюються під конкретне обладнання, що робить їх несумісними з іншим апаратним забезпеченням.

Для виконання відносно простих завдань потрібно написати велику кількість коду, тому складні програми вимагають великої кількості інструкцій і значного часу для їх розробки. Код на мові асемблера суттєво відрізняється від коду на мовах високого рівня. В асемблері немає звичних програмних конструкцій, які є в мовах високого рівня, і чим вищий рівень мови, тим далі вона від представлення на асемблері.

Найважливішим підходом, який необхідно розглянути, є переклад керуючих конструкцій (наприклад, операторів if, циклів, операторів switch) у форму, яку легко можна перетворити на асемблерний код.

Основна складність полягає в наступному:

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

Асемблерне представлення цього рівняння реалізується за допомогою інструкцій розгалуження, таких як команда JEQ. Ось простий спосіб виконати перетворення інакше:

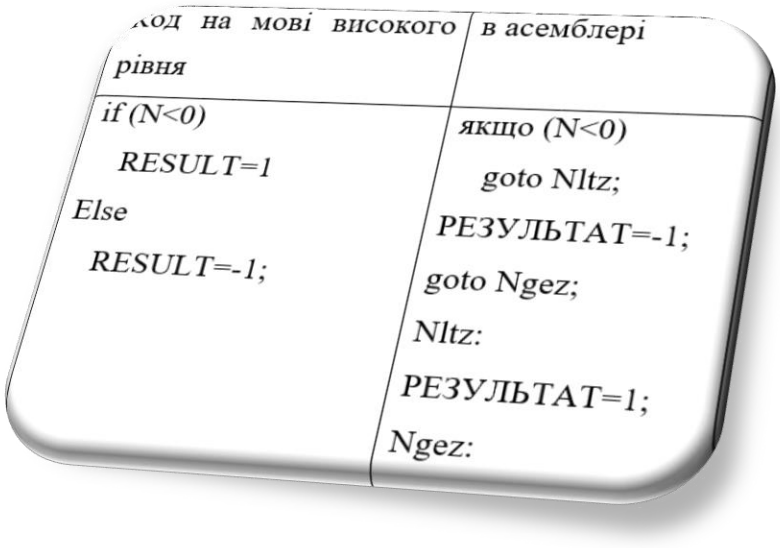
Код на мові високого рівня	в асемблері
if (якась-умова) щось;	if (!деяка-умова) goto x; щось; x:

Використання міток «goto» є нетиповим для мов високого рівня, але це необхідно, оскільки асемблерний код не має чіткої структури. Він складається з послідовності інструкцій, що включають оператори розгалуження та мітки, що робить його написання і читання незручним. Проте цей підхід може бути менш складним, якщо правильно організувати переклад. Розглянемо більш складний випадок: Інший спосіб зробити це.

Код на мові високого рівня	в асемблері

if (some-condition)
something
else
somethingelse;
	if (якась-умова)
	goto
	dosomething;
	somethingelse
	goto
	somethingdone;
	dosomething:
	щось
	somethingdone:

Розглянемо реальний приклад



Залежно від конкретної ситуації, можна почати оптимізувати цей код наступним чином:

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 структуру для перетворення, то для трансляції циклів потрібно набагато більше зусиль. Очевидно, що цикли часто використовуються разом з масивами, оскільки саме циклічні алгоритми є найбільш ефективними для обробки масивів. Розглянемо алгоритми, які перетворюють програмний код простого циклу, що обчислює 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 здається зрозумілим і очевидним, проте цей цикл насправді не виконує жодних корисних дій. Для більш детального аналізу модифікуємо розглянутий цикл. Тепер потрібно знайти суму не послідовних чисел, а суму всіх елементів масиву.

```
Int myarray [N];
For (SUM=0, i=0; i<=N; i++) SUM += myarray[i];
```

У асемблерному коді масив представлений як мітка, яка вказує на певний обсяг пам’яті (може бути ініціалізований або ні). Ця мітка є початковою адресою масиву, що відповідає &myarray[0] в асемблерному коді або просто myarray в С.

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

Просте сумування кількох значень вже не підійде, тому у середині асемблерного коду необхідно додати частину коду, яка буде відповідати за знаходження наступного елемента масиву та проведення операції обчислення суми цього елемента з накопиченим значенням змінної, яка зберігає попередні суми:

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 ускладнюється, якщо в програмі збільшити кількість масивів, які беруть участь у обчисленнях:

```
int myarray[N], otherarray[N+1];
for (i=0;i<N;i++)
    otherarray[i+1] = myarray[i];
```

Тепер оптимізація коду стає ще більш складною. До того як з'явилися оптимізуючі компілятори, програмістів не стимулювали писати код із використанням посилань на масиви. Натомість, рекомендувалося використовувати вказівники. Наприклад, якщо N дорівнює 1000, то кількість виконуваних інструкцій у циклі «асемблера» буде 6 на ітерацію * 1000 + 3 = 6003. Проте, у циклі є багато коду, який можна оптимізувати. Перепишемо цикл наступним чином:

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

У асемблерному коді це виглядатиме так:

```
SUM=0;
temp=myarray;
arrayend=&myarray[N];
loop: if (temp >= arrayend)
    goto loopdone;
SUM += *temp;
temp++;
goto loop;
loopdone:
```

У цій версії код вже скорочений до 4*1000 + 4 інструкції. Це стане ще більш значущим, якщо додати ще один масив. Оптимізуючі компілятори зробили цей вид ручної оптимізації менш важливим. Сьогодні ручна оптимізація часто заважає компілятору, замість того, щоб допомагати, оскільки компілятори очікують простого коду. Лише в окремих випадках ручна оптимізація може дати кращий результат, ніж сучасний оптимізуючий компілятор. Однак, при ручному перекладі

на асемблер подібні перетворення часто полегшують остаточний переклад С-подібного коду на асемблер.

2.3 Алгоритм трансляції програмного коду на мову асемблер

Як було продемонстровано у попередньому розділі, процес трансляції програмного коду з мов високого рівня на асемблерну мову може проходити через різні етапи, що впливає на кількість операцій у кінцевому додатку програми. Отже, ця дисперсія дає відмінності в часі виконання програм, що виконують однакові завдання. Крім того, кожна обчислювальна операція споживає певну кількість електричної енергії. Отже, розробка оптимізованого коду має потенціал для прискорення продуктивності програмного забезпечення та збереження енергетичних ресурсів.

Для полегшення точного функціонування алгоритму перекладу програмного коду були розроблені різні моделі для вирішення різних алгоритмічних сценаріїв. Ці моделі окреслюють сценарії, що охоплюють переклад кодових блоків, що містять лінійні, циклічні та розгалужені алгоритми, а також алгоритми обробки масивів, серед інших. Крім того, був зібраний масив, що охоплює всі зарезервовані слова мови C++, щоб полегшити точний аналіз вхідного коду. Використовуючи ці моделі, було сформульовано алгоритм перетворення програмного коду в код асемблера.

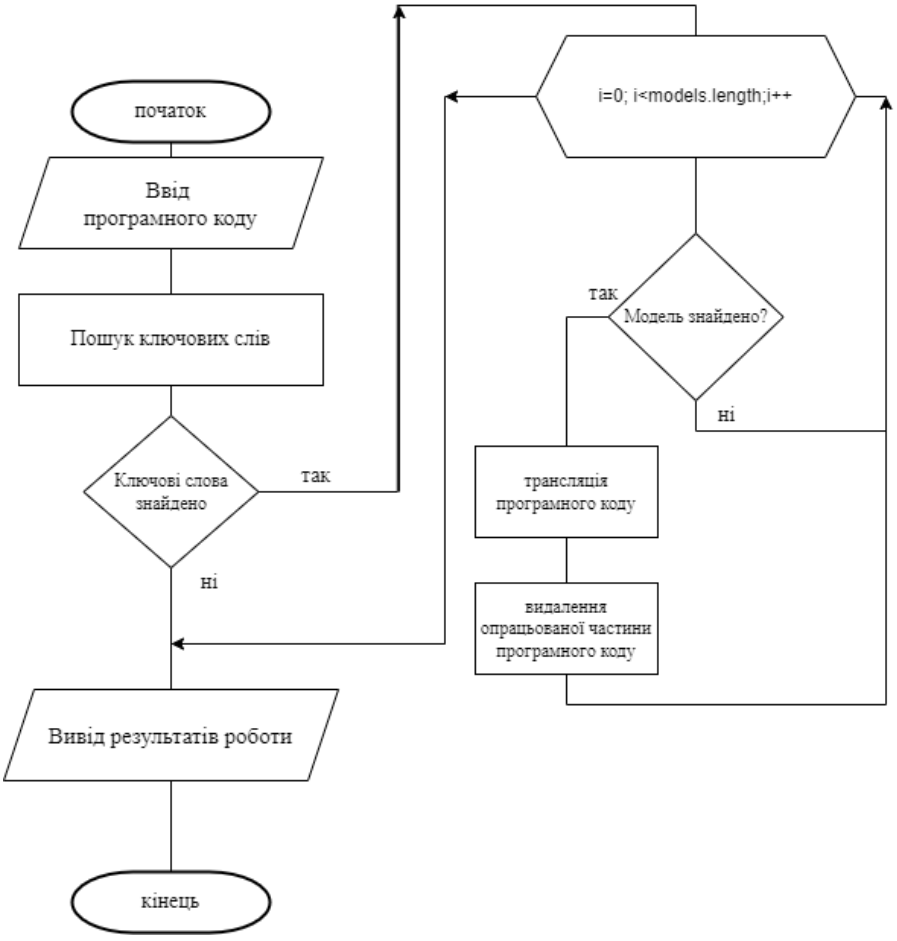


Рис.2.4 – Блок-схема алгоритму перекладу програмного коду з мови високого рівня на асемблер

Запропонований алгоритм базується на аналізі вхідного програмного коду та використанні масиву моделей трансляції. Основний принцип роботи полягає у покроковій перевірці вхідного коду для виявлення основних структурних блоків і їх перетворення з одного представлення в інше, зберігаючи при цьому основний сенс алгоритму. Алгоритм включає такі кроки:

1. Завантаження масиву моделей.
2. Отримання вхідного програмного коду.
3. Послідовний аналіз всіх слів вхідного коду.
4. Якщо слово є керуючим в C++, відбувається пошук відповідної моделі.
5. На основі знайденої моделі визначаються команди, які описують параметри роботи цього блоку.

6. Якщо всі частини команди знайдені, відбувається переклад на асемблер. У разі помилки користувача видаляється ключове слово і процес повертається до кроку 3.

7. Якщо всі слова в програмному коді перетворені на асемблер, результат виводиться і робота завершується, інакше процес повертається до кроку 3.

На початковому етапі здійснюється перевірка вхідного слова на відповідність зарезервованим словам мови C++, оскільки їх кількість невелика, це дозволяє швидко визначити приналежність слова. Після ідентифікації слова запускається процедура семантичного аналізу наступних слів для визначення параметрів команди. Операції виконуються швидко, оскільки програмний код структурований та відповідає встановленим правилам написання.

До основних переваг розробленого алгоритму належать:

– висока швидкість трансляції програмного коду з однієї мови програмування на іншу;

– мінімальні технічні вимоги до робочої станції, оскільки основні затримки залежать від обсягу коду, що підлягає трансляції;

– можливість використання для довільної мови високого рівня з незначною модифікацією набору зарезервованих слів.

А от недоліки поділяють на:

– алгоритм здатен коректно транслювати код з однієї мови на іншу, проте кінцевий результат залежить від якості написання початкового коду;

– чутливість до ситуацій, коли назви змінних у коді схожі на зарезервовані слова.

Аналіз результатів моделювання запропонованого алгоритму показав, що затримки відсутні, а процес трансляції коду відбувається з мінімальними часовими витратами. При правильному написанні вхідного коду, точність вибору моделей є високою.

2.4 Висновки до розділу

Проведено детальний аналіз алгоритмів і механізмів трансляції програмного коду з мов високого рівня на асемблерну мову з використанням моделей транскодування, що дало змогу розробити алгоритм перекладу програмного коду на асемблер. Створено алгоритм генерації оптимальних машинних кодів для мов високого рівня, що дозволило розробити структуру і реалізувати програмний додаток для написання оптимізованих програм на мовах високого рівня.

РОЗДІЛ 3. ПРОГРАМНИЙ ДОДАТОК НАПИСАННЯ ОПТИМАЛЬНОГО ПРОГРАМНОГО КОДУ

3.1 Структура програмного додатку написання коду мовою С++

Найефективніший спосіб оптимізації коду — це застосування профайлера для виявлення вузьких місць у продуктивності. Визначити, яка частина програми споживає найбільше ресурсів, досить складно без точних даних. Якщо ви спираєтесь лише на припущення, а не на конкретні дані, ви можете витратити багато часу на оптимізацію частин програми, які вже працюють швидко.

Після виявлення вузького місця, наприклад, циклу, що виконується тисячі разів, варто розглянути можливість реорганізації програми так, щоб не потрібно було повторювати цей цикл стільки разів. Це більш ефективно, ніж прискорення циклу на 10%, що, до речі, може бути автоматично здійснено оптимізуючим компілятором.

Оптимізація може бути марною тратою часу, якщо дотримуються хоча б одного з цих тверджень:

- частини програми ще не завершені;
- програма ще не була повністю протестована та відладжена;
- програма вже працює досить швидко.

Також варто врахувати, як саме використовується програма. Якщо це звітний додаток, який запускається лише раз на день, користувач може запустити його перед обідом, і немає великої потреби завершувати його до повернення. Якщо додаток викликається іншим, який працює повільніше, користувач, швидше за все, не помітить різницю в продуктивності. Однак, якщо програма обробляє події миші для графічного інтерфейсу і користувачі скаржаться на затримку, оптимізація стає важливою.

Компілюйте код у повному режимі оптимізації та протестуйте програму в реальних умовах. Якщо немає доступу до реальних вхідних даних, ретельно обирайте тестові дані. Програмісти часто перевіряють програму на меншому обсязі даних та в умовах, відмінних від тих, з якими програма може зіткнутися в реальній експлуатації.

Сучасні цифрові комп'ютери мають лише чотири основні типи апаратних інструкцій для програмування:

1) Переміщення даних з одного місця в інше або до/з пристроїв вводу-виводу.

2) Виконання простих арифметичних операцій, таких як додавання та множення.

3) Виконання умовних переходів.

4) Виконання викликів підпрограм та повернень.

Мови високого рівня, такі як C, перетворили ці чотири типи інструкцій у відповідні абстракції, які програмісти використовують для написання програмного забезпечення:

1. Змінні та складні структури даних, слугують абстракцією для чисел, які зберігаються у пам’яті. Вони є основою для обробки даних.

2. Оператори обчислень використовують змінні для виконання арифметичних операцій і збереження результату в інших змінних. Це оператори присвоєння, які здійснюють всі розрахунки в програмах.

3. Логічні оператори керування потоком використовують умовні інструкції для реалізації циклів, операторів if-then, do-while та switch-case, що дозволяє програмістам створювати логіку управління в програмі.

4. Функції (підпрограми), використовують виклики та повернення, обробляють вхідні параметри, створюють змінні та виконують необхідні обчислення.

Розроблено внутрішню архітектуру для реалізації програмного забезпечення Маршруту та оцінки запропонованих алгоритмів перекладу. Модульний підхід було обрано в якості основи, оскільки цей метод спрощує обробку та розширення функціональних можливостей. На рис. 3.1 наведена спрощена схема структури програми.

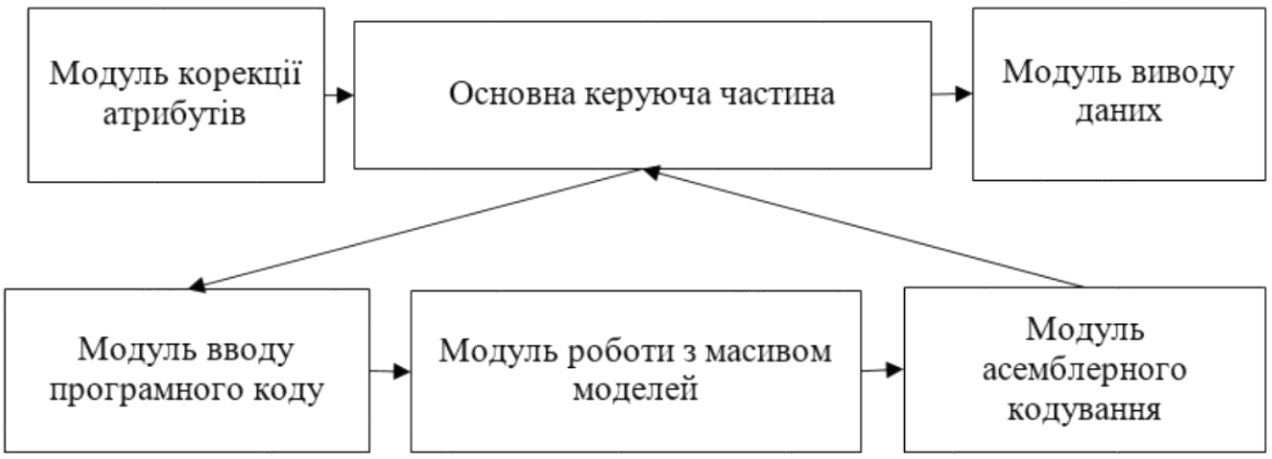


Рис. 3.1 – Загальна структура програмного додатка для трансляції коду на асемблер

Розроблена структура програмного додатка повинна забезпечити можливість програмної реалізації запропонованих алгоритмів обробки текстових даних для перекодування програмного коду з мов високого рівня на мову асемблера. У процесі проектування було використано принципи модульної архітектури, які мають низку переваг при розширенні або модифікації функціональних можливостей програмного забезпечення. Крім того, така архітектура дозволяє реалізувати програму в декілька етапів, що значно полегшує процес написання коду та спрощує інтеграцію зовнішніх бібліотек. При програмній реалізації вибрані алгоритми були згруповані відповідно до завдань, які вони вирішують. Серед основних модулів програми можна виділити: керуючий модуль, блок корекції параметрів, набір функцій для введення програмного коду, інструменти для взаємодії з базою даних моделей, а також функції, що реалізують алгоритм трансляції коду з мови високого рівня на асемблер.

Основні функції для управління роботою програми зібрані у керуючому модулі. Він містить засоби для тестування компонентів додатка. Якщо один з модулів функціонує некоректно або відсутній, функції, закладені в процесі реалізації, здатні правильно обробити цю ситуацію і запобігти аварійному завершенню програми. Ця особливість особливо корисна на етапах реалізації, тестування та розширення додатка новими функціями. Під час тестування, для прискорення аналізу функціонування програмного коду, замість результатів

роботи відсутніх функціональних блоків будуть використовуватися тестові набори, що дозволить тестувати не лише всю програму, але й окремі її модулі.

Для зручності роботи користувачів було реалізовано систему спливаючих повідомлень, які інформують про поточний стан програми або виникнення критичних або непередбачених помилок. Для запуску функцій цього модуля необхідно передати вектор технічних параметрів. Після виконання, програма надасть звіт про готовність до роботи та налаштує параметри за замовчуванням.

Для забезпечення взаємодії з користувачем був створений модуль введення програмного коду. Він дозволяє вводити код або у вигляді текстового файлу, або безпосередньо вводити текст у відповідне вікно програми. У першому випадку користувач може повернутися до коду, створеного під час попереднього запуску програми, або до коду, створеного в інших інтегрованих середовищах розробки. Цей підхід дозволяє оцінити код, створений іншими користувачами, або провести оцінку вже існуючого коду. У другому випадку користувачі можуть у режимі реального часу оцінювати трансляцію коду та оптимізувати його для отримання більш ефективного коду, який вимагає менше апаратних ресурсів.

Один з основних блоків програмної системи - це набір функцій для взаємодії з базою даних моделей, що описують структури мови програмування C++ (рис. 3.2). Функціональні можливості цього блоку включають стандартні засоби для роботи з базою даних. Початковий набір моделей задається за замовчуванням, але користувачі можуть додавати нові моделі або вносити зміни в існуючі.

Варто відзначити функції пошуку відповідних моделей за ключовими словами. Пошук здійснюється за допомогою бінарного алгоритму, що забезпечує швидкий пошук і не спричиняє додаткових затримок у роботі програми.

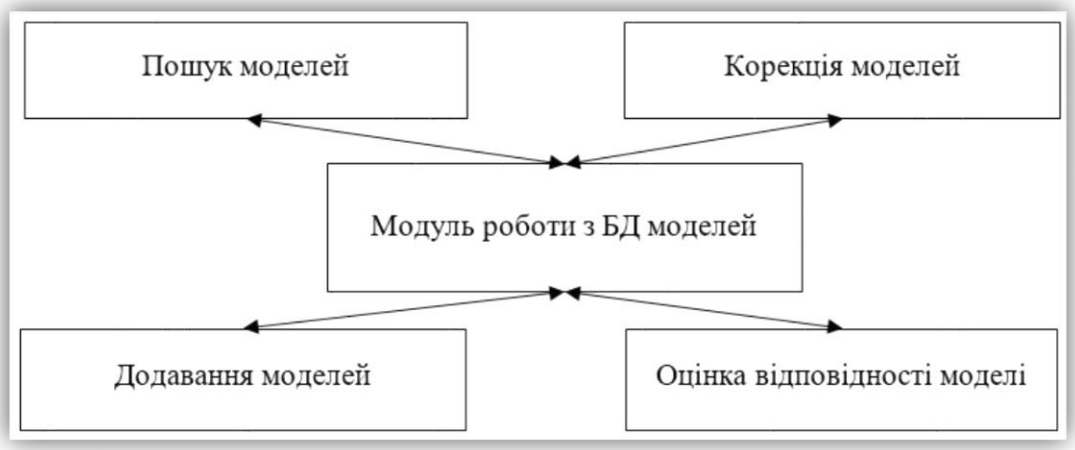


Рис. 3.2 – Загальна структура модуля для роботи з моделями

В результаті виконання цього набору функцій здійснюється взаємодія з базою даних, яка дозволяє визначити модель, що відповідає аналізованій частині програмного коду.

Блок трансляції коду з мови програмування високого рівня на асемблер функціонує як допоміжний елемент і працює на основі отриманої моделі, а також проведення семантичного аналізу коду або блоку команд для визначення параметрів відповідної інструкції. Пошук проводиться на основі знайденої моделі написання команди на мові C++ і може змінюватися залежно від стилю написання коду конкретним користувачем.

Наприклад, навидемо програмний код:

```
number = number +1;
number+=1;
number++;
```

Ці рядки коду виконують одну й ту саму операцію, тобто збільшення значення змінної number на одиницю, і, відповідно, будуть транслюватися в один і той же код на асемблері. Однак у мові C++ ці рядки коду мають різний синтаксичний формат. Результатом роботи функцій цього блоку буде перетворення програмного коду з мови високого рівня у формат на мові асемблер.

Блок виводу результатів роботи програмного додатку є допоміжною структурною одиницею запропонованої архітектури. Його головне завдання – забезпечити коректне відображення трансльованого коду у зручному для

користувача форматі. Крім того, додано функціонал для збереження результатів у текстовому файлі. Ця архітектура представляє собою комплексне технічне рішення, яке забезпечує швидке впровадження окремих компонентів, тестування як усієї розробки, так і окремих модулів, а також спрощує модифікацію та розширення функціональних можливостей.

При цьому зберігається загальна цілісність і основні функції програмного додатку для трансляції коду з високорівневої мови на асемблер не зазнають значних змін. Після завершення етапу проектування та дослідження структури додатку необхідно провести адаптацію алгоритму та моделювання архітектури, щоб виявити можливі проблеми при обміні даними між різними компонентами. Це моделювання здійснювалось за допомогою універсальної мови моделювання (UML) з використанням попереднього досвіду розробки.

Для моделювання було обрано такі UML-діаграми:

– діаграми класів (аналіз структури даних);

– діаграми прецедентів (визначення доступу різних груп користувачів до функціональних можливостей програми);

– діаграми послідовностей (аналіз можливих сценаріїв поведінки користувачів та програми для досягнення необхідних результатів).

Процес моделювання має значні переваги, оскільки дозволяє виявити проблеми в архітектурі та алгоритмах на ранніх етапах і передбачити можливі помилкові дії користувачів.Це дозволяє внести корективи без додаткових фінансових і часових витрат. Однак, моделювання потребує короткострокових витрат на його якісне виконання, зокрема:

- часові витрати на підбір кваліфікованих фахівців, ознайомлення їх з розробками, створення та аналіз моделей. Навіть якщо моделі є схематичними і не залежать від робочого коду, час на моделювання потрібно включити в терміни виконання замовлення;
- суб'єктивність процесу моделювання, оскільки його результати залежать від кваліфікації спеціалістів та їхньої уваги до деталей. Недостатня увага

може призвести до втрати інформації та критичних проблем на етапі реалізації коду;

- необхідність робочих станцій та додаткових ресурсів, що може призвести до зупинки процесу створення продукту під час очікування результатів моделювання, що спричиняє додаткові часові та фінансові витрати.

Незважаючи на ці недоліки, етап моделювання є важливою і невід'ємною частиною проектування та реалізації складних програмних систем. Для перевірки доступу до функціональних можливостей додатку було обрано моделювання за допомогою UML-діаграм прецедентів.

У цій діаграмі будуть відображені можливості доступу окремих груп користувачів до функціоналу програмного додатку, а також показано їхню взаємодію в межах запропонованих програмних блоків. Приклад діаграми прецедентів наведено на рисунку 3.3.

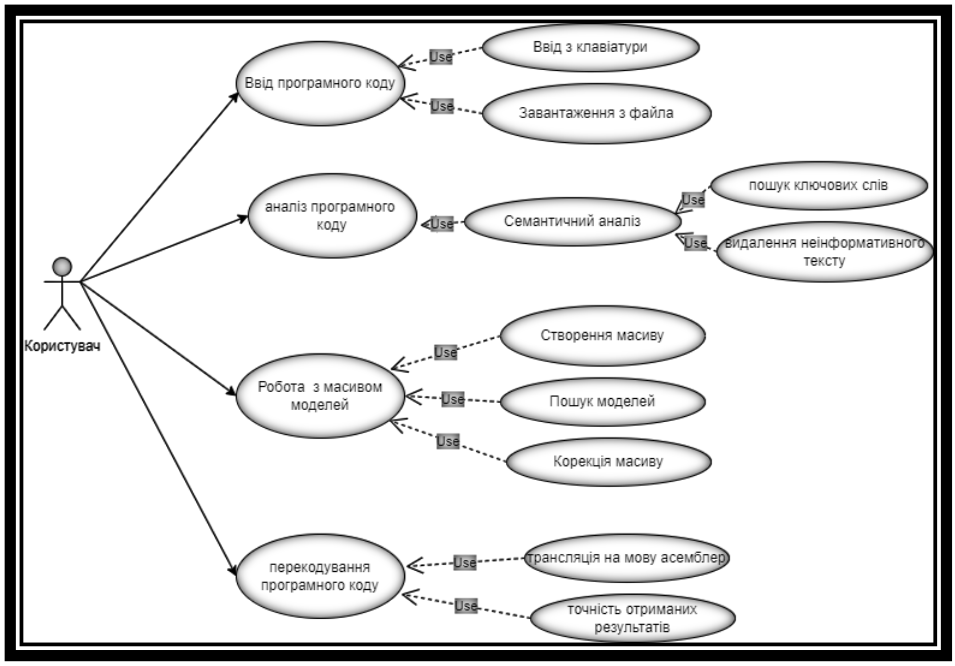


Рис. 3.3 – Діаграма сценаріїв використання програмної системи

Завдяки аналізу діаграми сценаріїв використання вдалося оцінити загальні взаємозв'язки між різними модулями системи, провести дослідження структур і форматів даних, які використовуються для організації інтерфейсу взаємодії, а також визначити рівень доступу різних груп користувачів до функціональності

програмних модулів. Як показано на рис. 3.3, група акторів «Користувач» має доступ до засобів введення програмного коду мовою високого рівня, інструментів для роботи з базою даних моделей програмних структур та функцій аналізу отриманих результатів.

Цей набір функцій надає користувачеві повний контроль над процесом роботи програми та можливість впливати на результати. Однак частина функцій залишається прихованою для мінімізації людського фактора та зменшення ймовірності виникнення критичних помилок через некоректні дії користувача.

Важливим етапом перевірки запропонованої архітектури та алгоритму є аналіз поведінкових сценаріїв користувача та системи, які закладені в програму. Для дослідження цих сценаріїв використовувалася діаграма послідовностей, яка є частиною універсальної мови моделювання (UML).

Ця діаграма дозволяє розглянути принципи взаємодії між користувачами та програмною системою у разі виникнення різних ситуацій при виконанні завдань. Результати моделювання наведено на рисунку 3.4.

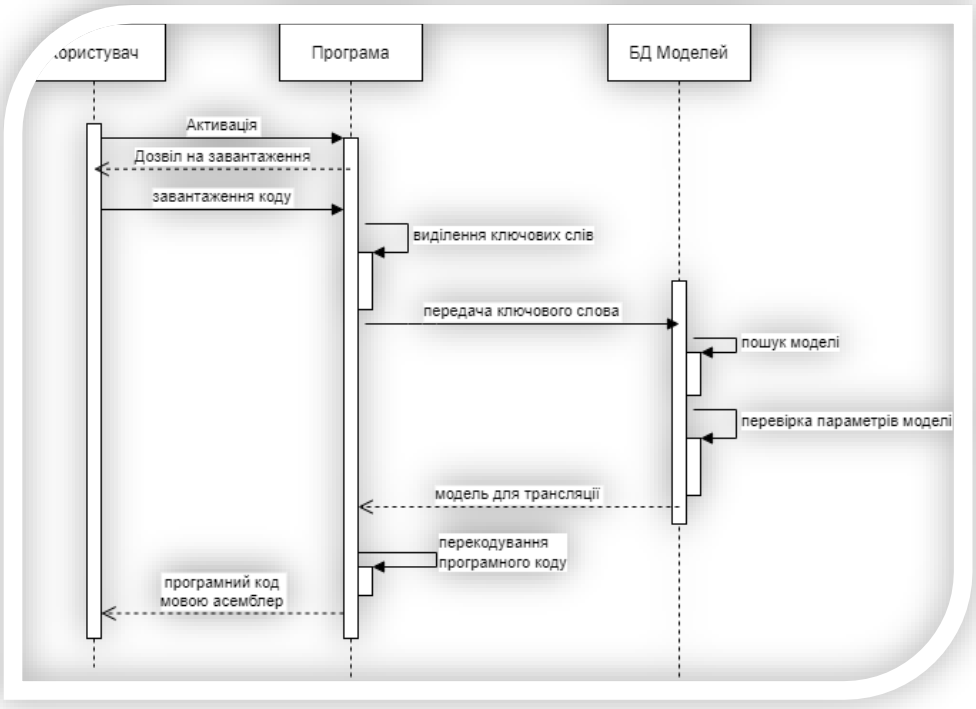


Рис. 3.4 – Схема послідовності операцій програмної системи

Під час проведення моделювання аналізувалися дії користувачів, такі як отримання програмного коду з клавіатури або з файлу, перевірка наявності помилок у вхідному коді тощо. У результаті побудови схеми були зроблені висновки, що запропонована структура програмного додатку, а також алгоритм трансляції програмного коду з однієї мови на іншу, повністю виконують поставлені завдання. Помилкові ситуації не виникали, результати моделювання відповідали очікуванням і знаходилися в межах допустимої похибки. На заключному етапі моделювання необхідно було проаналізувати повноту запропонованих структур даних і забезпечення їхньої взаємодії в межах запропонованої архітектури. Для цього було проведено моделювання на основі діаграми класів. Отримані результати проілюстровані на рисунку 3.5 нижче.

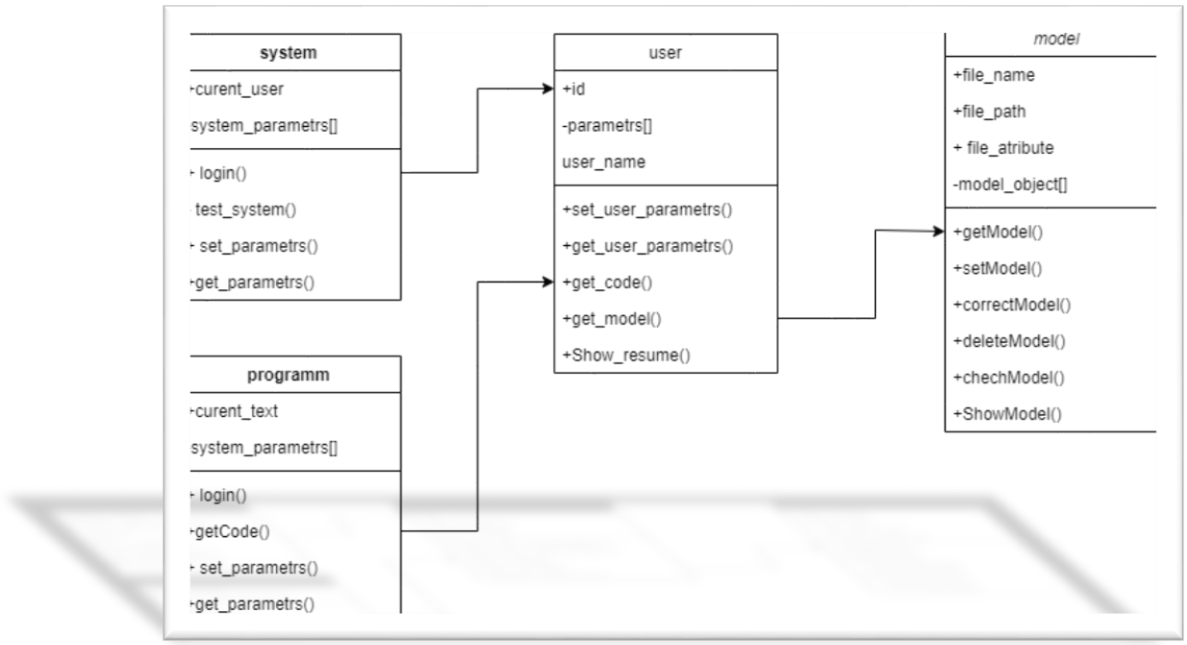


Рис. 3.5 – Структурна діаграма класів програмної системи

Цей тип аналізу був застосований для оцінки внутрішньої структури класів, які будуть реалізовані в межах розробки цього програмного додатку. Візуальний аналіз отриманої діаграми класів показав, що всі класи добре відображають відповідні сутності. Набір використовуваних атрибутів дозволяє однозначно характеризувати об'єкти відповідних класів, а множина методів дозволяє виконувати всі необхідні завдання.

Крім того, у структуру класів було інтегровано механізм обміну даними за наперед узгодженими правилами з дотриманням відповідних інтерфейсів. Зрештою, моделювання проводилося на основі трьох різних критеріїв оцінки архітектури та розроблених структур даних, що в кінцевому результаті продемонструвало коректність роботи програми під час виконання завдань і мінімізацію виникнення непередбачуваних ситуацій через вплив користувачів на процеси всередині програмного модуля.

Наступним завданням, яке необхідно вирішити в межах проектування та розробки системи генерації оптимального програмного коду, є створення інтерфейсу, що дозволить користувачам у зручній формі завантажувати вхідні дані та аналізувати результати роботи програми.

Оскільки користувач буде безпосередньо взаємодіяти з програмною системою, для підвищення зручності було розроблено графічний інтерфейс з гнучким підходом. Загальний вигляд запропонованого графічного інтерфейсу для проєктованої системи наведено на рисунку 3.6.

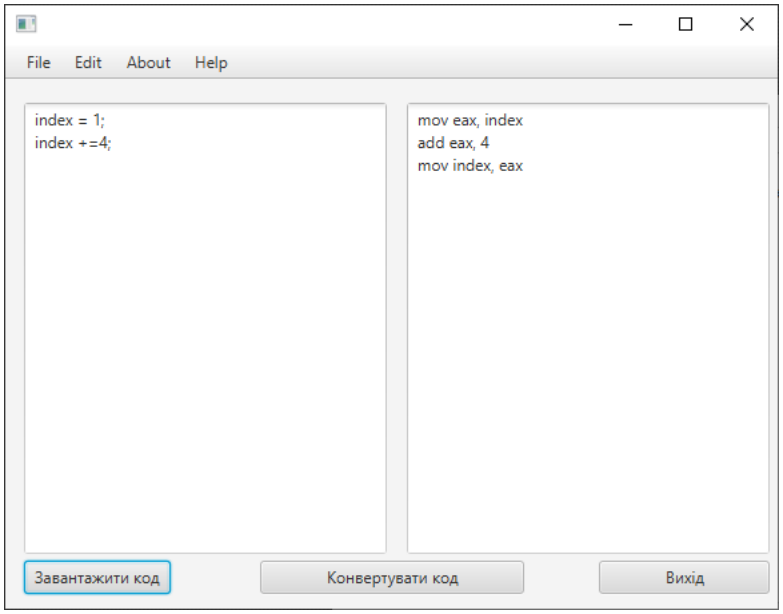


Рис. 3.6 – Візуальний інтерфейс користувача

Під час проєктування інтерфейсу головного вікна програмного додатку були враховані наступні ключові аспекти:

1. Основний потік вхідних та вихідних даних буде представлений у текстовому форматі.

2. Кількість активних елементів управління буде зведена до мінімуму, оскільки для коректної роботи програми необхідно мінімізувати вплив людського фактору.

3. Процес введення програмного коду повинен бути зручним та інтуїтивно зрозумілим, щоб не викликати у користувачів додаткових запитань.

4. Результати роботи програми повинні виводитися у зручному для користувача форматі, що не створює дискомфорту під час візуального оцінювання.

На основі цих вимог було розроблено графічний інтерфейс користувача, який повністю відповідає поставленим умовам, дозволяє користувачеві у зручній формі отримувати доступ до всіх елементів управління програмою, а результати роботи надаються у максимально зручному форматі для користувача.

Додатково був проведений аналіз для визначення необхідних та допоміжних кроків, які повинен виконати користувач для досягнення поставленого завдання з мінімальними часовими затратами. Послідовність цих дій відображена на рисунку 3.7:

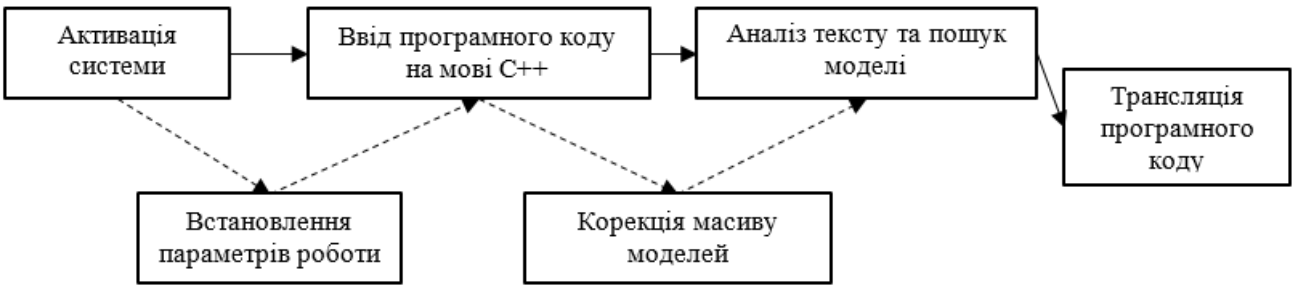


Рис. 3.7 – Порядок дій користувача у системі для трансляції програмного коду з високорівневої мови на асемблер

Як показано на рисунку 3.7, кількість дій для досягнення очікуваного коректного результату включає три основні операції:

– введення програмного коду на мові високого рівня;

– вибір моделі для опису частини програмного коду на основі пошуку ключових слів та параметрів;

– проведення візуальної оцінки отриманих результатів.

Така послідовність дій забезпечує швидке та ефективне виконання поставлених завдань, дозволяючи користувачеві досягти необхідного результату без зайвих витрат часу.

Ці операції можуть бути доповнені рядом додаткових функцій, таких як корекція параметрів роботи програми, налаштування бази даних моделей тощо. Такі дії спрямовані на підвищення якості отриманих результатів та покращення зручності використання запропонованої системи.

3.2 Програмні модулі для оцінки та генерації програмного коду

Процес роботи програмного додатку можна умовно розділити на кілька незалежних етапів: введення програмного коду, пошук ключових слів, визначення моделей, заповнення параметрів моделі та трансляція коду. Перші два етапи є досить простими і можуть бути реалізовані за допомогою стандартних засобів мови програмування через організацію обробки вхідного потоку інформації та відображення текстових даних у візуалізованому класі. Другий етап полягає в послідовному переборі введених слів для виявлення збігів у базі даних моделей.

```
if(!searchModel(str))
     return -1;
else
{
     int valueModel = hash(str);
     if(arrayModel[value].length != 0)
     {
     return valueModel;
     }
}
```

Для визначення параметрів моделі необхідно провести додатковий аналіз отриманого рядка програмного коду. На рисунку 3.8 наведено типовий приклад програмного коду циклічного алгоритму.

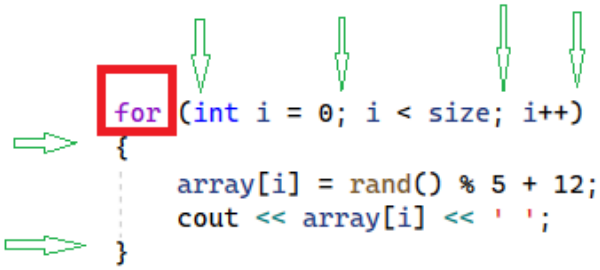


Рис. 3.8 – Зразок програмного коду циклу на мові C++

При виявленні в рядку коду ключового слова for стає очевидним, що в цьому місці знаходиться частина коду, яка відповідає за реалізацію циклічного набору команд. Цей елемент утворює цикл, і згідно зі стандартом мови C++, біля ключового слова повинні бути розміщені додаткові елементи, а саме круглі дужки () з параметрами роботи циклу, а також символи ;, які розділяють внутрішній простір у круглих дужках на три окремі сектори. Крім того, після керуючих символів має слідувати блок команд, так зване «тіло циклу», яке повинно бути розташоване у фігурних дужках {}. Після визначення всіх структурних елементів та параметрів обраної моделі, проводиться додатковий аналіз «тіла циклу»:

```
array[i] = rand() % 5 + 12;
cout << array[i] << ' ';
```

Цей аналіз включає перевірку структурних компонентів та параметрів, щоб забезпечити коректне функціонування циклу відповідно до визначених вимог. Всі кроки виконуються з урахуванням стандартів мови програмування C++ і з метою забезпечення ефективної реалізації програмного коду.

Оскільки «тіло циклу» є частиною програмного коду, яку також потрібно перетворити в асемблерний код, його аналіз повторюється. Ця процедура триває до повного аналізу програмного коду і розбиття його на прості компоненти.

Для оцінки процесу перетворення програмного коду на асемблерне представлення проведемо серію трансформацій, що ілюструють прості типи алгоритмів.

Лінійні алгоритми характеризуються послідовним виконанням всіх кроків без переходів всередині програми. Всі переходи здійснюються в порядку розташування програмних команд. На рис. 3.9 показано приклад програмної

реалізації простого лінійного алгоритму додавання двох змінних і результат його трансляції на асемблер.

```
int numberOne = 4;
int numberTwo = 5;
int numberThree = numberOne + numberTwo;
```

```
    int numberOne = 4;
00287D96  mov         dword ptr [numberOne],4
    int numberTwo = 5;
00287D9D  mov         dword ptr [numberTwo],5
    int numberThree = numberOne + numberTwo;
00287DA4  mov         eax,dword ptr [numberOne]
00287DA7  add         eax,dword ptr [numberTwo]
00287DAA  mov         dword ptr [numberThree],eax
```

a) мова C++ b) мова асемблер

Рис. 3.9 – Зразок перетворення лінійного алгоритму

Як видно з наведеного прикладу, для реалізації простого лінійного алгоритму на асемблерній мові необхідно додатково використовувати регістри, щоб здійснити операцію додавання двох змінних, розташованих у комірках пам'яті. Однак конвертація коду такого типу не викликає складнощів, оскільки всі команди виконуються послідовно, є однозначними і легко ідентифікуються. Вони не мають внутрішніх зв'язків між собою. Кожен рядок коду можна легко конвертувати окремо, враховуючи лише параметри, що характерні для простих операторів. У прикладі це оператори присвоєння «=» і додавання «+».

Для аналізу можливості трансляції програмного коду, що відповідає алгоритмам з розгалуженням, розглянемо приклад, наведений на рис. 3.10.

```
int numberOne = 4;
if (numberOne == 4)
{
    int numberTwo = 5;
}
else
{
    int numberThree = 5;
}
```

```
    int numberOne = 4;
005D7D96  mov         dword ptr [numberOne],4
    if (numberOne == 4)
005D7D9D  cmp         dword ptr [numberOne],4
005D7DA1  jne         __$EncStackInitStart+60h (05D7DACh)
    {
        int numberTwo = 5;
005D7DA3  mov         dword ptr [ebp-7Ch],5
    }
005D7DAA  jmp         __$EncStackInitStart+6Ah (05D7DB6h)
    else
    {
        int numberThree = 5;
005D7DAC  mov         dword ptr [ebp-88h],5
    }
```

a) мова C++ b) мова асемблер

Рис. 3.10 – Зразок перетворення алгоритму з розгалуженням

Цей варіант трансляції є складнішим, оскільки включає елементи переходу між різними частинами програмного коду, використовуючи команди умовних та безумовних переходів. Це зрозуміло, оскільки програма мовою асемблера є

послідовністю команд, що виконуються одна за одною. Пропуск частини команд або перехід на певний сегмент коду може бути реалізований лише через примусове переміщення вказівника активної команди.

Для завершення аналізу можливостей трансляції програмного коду, розглянемо приклад перекодування циклічного алгоритму. Для цього типу алгоритмів характерно, що частина команд повинна повторюватися. Це дозволить програмісту зменшити розмір файлу, уникнувши необхідності написання схожого коду кілька разів. Приклад перетворення наведено на рис. 3.11.

```
int numberOne = 4;
for (int index = 1; index < 5; index++)
{
    numberOne += index;
}
```

a) мова C++

```
    int numberOne = 4;
007D4DE6  mov         dword ptr [numberOne],4
    for (int index = 1; index < 5; index++)
007D4DED  mov         dword ptr [ebp-7Ch],1
007D4DF4  jmp         __$EncStackInitStart+63h (07D4DFFh)
007D4DF6  mov         eax,dword ptr [ebp-7Ch]
007D4DF9  add         eax,1
007D4DFC  mov         dword ptr [ebp-7Ch],eax
007D4DFF  cmp         dword ptr [ebp-7Ch],5
007D4E03  jge         __$EncStackInitStart+74h (07D4E10h)
    {
        numberOne += index;
007D4E05  mov         eax,dword ptr [numberOne]
007D4E08  add         eax,dword ptr [ebp-7Ch]
007D4E0B  mov         dword ptr [numberOne],eax
    }
007D4E0E  jmp         __$EncStackInitStart+5Ah (07D4DF6h)
```

b) мова асемблер

Рис. 3.11 – Зразок перетворення циклічного алгоритму

3.3 Тестування та аналіз реалізованого програмного додатку

На фінальному етапі розробки програмного додатку для трансляції коду з мови високого рівня на мову асемблера було проведено тестування готового продукту. Для успішного тестування та отримання об’єктивних результатів при виборі апаратних характеристик робочої станції необхідно враховувати особливості використання програмної системи. Серед обмежень, накладених на програмну систему, варто зазначити: вхідна та вихідна інформація надаватиметься у текстовому форматі, що знижує вимоги до характеристик пристроїв введення даних. Результати роботи повинні бути виготовлені у зручному для користувача форматі.

Вхідні та вихідні дані повинні зберігатися на твердих носіях для можливого повторного використання. Аналіз сучасних апаратних рішень і їх цінових характеристик показав, що для тестування можна використовувати робочу станцію середнього цінового діапазону, яка належить до категорії офісних

комп'ютерів. Ці комп'ютери характеризуються помірними технічними параметрами, але їх вартість відносно невисока.

Випробування проводилися на робочому місці з середніми технічними характеристиками, серед яких:

• Процесор: середнього рівня з достатньою кількістю ядра для паралельної обробки завдань.

• Оперативна пам'ять: Достатньо для зберігання та обробки великих обсягів текстових даних.

• Жорсткий диск: Достатньо для зберігання вхідних і вихідних даних.

• Відеокарта: Основна, оскільки графічні обчислення не є критичними для цього додатка.

• Операційна система: сучасна версія, яка підтримує необхідне тестове програмне забезпечення.

Під час тесту було протестовано кілька ключових аспектів:

1. Швидкість перекод

2. Виправлений вивід: результати перекодування відображаються в зручному для користувача форматі, що ще більше полегшує роботу з кодом.

3. Збереження даних: вхідні та вихідні дані були успішно збережені на жорсткому диску для повторного використання, що підтвердило відповідність системним вимогам.

Тому тести показали, що реалізований програмний додаток відповідає заявленим вимогам і обмеженням. Використання робочої станції середньої дальності дозволило досягти всіх поставлених цілей і забезпечити стабільну роботу програми в реальних умовах експлуатації.

На рис. 3.13 представлено технічні характеристики:

1	Процесор	Intel Core i3-7100
2	Чіпсет	Intel® B250
3	Оперативна пам’ять	8GB (2x4GB) DIMM DDR4 (частота шини 2400MHz) Crucial.
4	Носій інформації	500GB (1x500GB) SATA 6Gb/s 3.5" Seagate
5	Відеокарта	ASUS, серія: GeForce GT 730, об'єм відео-пам'яті 2GB, GT730
6	Мережа	інтегрована на мат. платі: 1Gb/s 1шт.
7	Блок живлення	450W
8	Фірма виробник	Gigabyte

Рис. 3.12 – Специфікації тестової робочої станції

Для даного типу комп’ютерів характерним є використання для простих завдань, які не потребують складних обчислень, наприклад, такі машини можуть використовуватися секретарями, бухгалтерами та звичайними користувачами для роботи з офісними програмами та перегляду інформації в Інтернеті. Особливу увагу слід звернути на те, що розробники програмного забезпечення проводять багато часу за монітором комп’ютера. Тому для перегляду результатів роботи програми та покращення зручності роботи з розробленою програмною системою було обрано монітор середньої цінової категорії з відповідними технічними характеристиками. Специфікації дисплея (DELL P2018H 19.45") представлені на рисунку 3.13 нижче:

	A	B
1	Характеристика	Значення
2	матриця	TN
3	роздільна здатність	1600×900 (16:9)
4	частота поновлення кадрів	75 Гц
5	яскравість	250 кд/м²
6	підсвічування	WLED
7	входи	HDMI 1.4, DisplayPort 1.2, VGA (D-Sub)
8	розміри	570 * 446 * 226 мм
9		

Рис. 3.13 – Специфікації дисплея

Ця конфігурація може бути легко відтворена для підтвердження результатів, що гарантує її надійність під час тестування системи трансляції коду з високорівневої мови на асемблер.

Для перевірки функціональних можливостей розробленого програмного додатку було визначено критерії, на основі яких були обрані типи тестів для оцінки роботи програмного забезпечення. Серед критеріїв, що враховувалися, були наступні:

– тип використовуваного алгоритму;
– кількість необхідних внутрішніх переходів у алгоритмах;
– обробка масивів;
– комбінація різних типів алгоритмів.

Відповідно до цих параметрів було обрано такі типи алгоритмів для трансляції з однієї мови на іншу:

1) **Простий лінійний алгоритм** — характеризується простою структурою без додаткових переходів у коді. Масиви у цих алгоритмах не використовуються.

2) **Алгоритми розгалуження** — включають приклади з елементами розгалуження для прийняття рішень. Розгалуження може бути як одиночним, так і множинним. Використання каскадних розгалужень не передбачено.

3) **Прості циклічні алгоритми** — характеризуються елементами повторення коду. Масиви у цих алгоритмах не використовуються. Тіло циклу містить лише лінійні елементи. Використання вкладених циклів не передбачено.

4) **Прості гібридні алгоритми** — включають приклади, що комбінують різні типи простих алгоритмів. Використання масивів не передбачено.

5) **Алгоритми з використанням масивів** — для оцінки здатності програмного засобу працювати з алгоритмами, які обробляють дані у форматі масиву. Алгоритми можуть бути лише одного типу.

6) **Гібридні алгоритми з обробкою масивів** — найбільша група алгоритмів, які використовуються для вирішення типових завдань під час програмування.

Результати тестування представлені у таблиці 3.1.

Результати тестування показали високу ефективність розробленого програмного додатку, що підтверджується високим відсотком успішних трансляцій та мінімальними часовими витратами на перекодування. Усі типи алгоритмів були успішно протестовані, що дозволяє зробити висновок про надійність та функціональність програмного засобу.

Таблиця 3.1 – Зведена 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

Отримані результати тестування продемонстрували, що використана структура та розроблений набір алгоритмів є ефективними під час проектування та розробки системи для обробки програмного коду.

3.4 Висновки до розділу

Отже, було розроблено і змодельовано програмний модуль для трансляції коду з високорівневої мови на асемблер, що дозволило створити додаток для генерації оптимального коду.

Також було проведено тестування цього додатку, що підтвердило правильність вибору методів, алгоритмів та бібліотек для реалізації програми і розробленого алгоритму. Отримані результати тестування підтвердили доцільність використання вибраної структури та розробленого набору алгоритмів під час проектування та розробки системи для обробки програмного коду.

ВИСНОВКИ

Після аналізу інтегрованих середовищ розробки та алгоритмів трансляції коду на мови високого рівня можна зробити такі висновки:

1. Було проведено детальний аналіз програмних додатків, їх архітектурних елементів і форматів даних.
2. Мови програмування були класифіковані на високому та низькому рівні, з виділенням мови C++ як основної для подальших досліджень.
3. Було проведено аналіз інтегрованих середовищ розробки з фокусом на переклад високого рівня на мову для компіляції.
4. Вивчені алгоритми перекладу коду програмування з мови високого рівня в асемблер з використанням моделей транскодингу.
5. Розроблено інноваційний алгоритм для створення оптимального машинного коду для мов високого рівня.
6. Проведено прикладний тест, який підтвердив ефективність розробленого алгоритму.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. "Engineering a Compiler" by Keith Cooper and Linda Torczon - Книга пропонує огляд технік компіляції та оптимізації, зокрема генерацію машинного коду, з фокусом на практичній реалізації.
2. Carchrae, T. and Beck, J. Applying machine learning to low knowledge control of optimization algorithms. Computational Intelligence 21, 4 (Nov. 2005), 373–387.
3. Cheeseman, P., Kanefsky, B., and Taylor, W.M. Where are the really difficult problems. In the Proceedings of the 12th Joint International Conference on Artificial Intelligence. Morgan Kaufmann, San Mateo, CA, 1991, 331-337.
4. 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 Calculus. ACM Press, New York, 2008, 913-920.
5. 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 the Proceedings of the 14th International IFIP/IEEE Workshop on Distributed Systems: Operations and Management, Vol. 2867 LNCS. Springer-Verlag, Berlin/Heidelberg, 2003, 3-15.
6. Gomes, C.P. and Selman, B. Portfolio of algorithms. Artificial intelligence 126, 1-2 (Feb. 2001), 43-62.
7. Guerri, A. and Milano, M. Learning techniques for the automatic selection of the portfolio of algorithms. Proceedings of the 16th European Conference on Artificial Intelligence. IOS Press, Amsterdam, 2004, 475-479.
8. Hamerly, G. and Elkan, C. Learning k in k-means. In Proceedings of the Conference on Advances in Neural Information Processing Systems. MIT Press, Cambridge, MA, 2004, 281-288.
9. Hoos, H.H. Automated algorithm configuration and parameter tuning. In Autonomous Research, Y. Hamadi and F. Saubion, Eds. Springer-Verlag, 2011.
10. Hoos, H. Programming by optimization. Technical report TR-2010-14. Department of Computer Science, University of British Columbia, Vancouver, 2010.

11. . Hoos, H. Computer-aided design of high-performance algorithms. TR 2008-16 Technical Report. Department of Computer Science, University of British Columbia, Vancouver, 2008. Hoos, H.H. and Stützle, T. Stochastic Local Search: Foundations and Applications. Morgan Kaufmann Publishers, San Francisco, 2004.

12. Huberman, B., Lukose, R., and Hogg, T. An economics approach to hard computational problems. Science 275, 5296 (Jan. 1997), 51–54.

13. 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.

14. 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.

15. 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.

16. 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.

17. Assembly Language for x86 Processors" by Kip R. Irvine

18. 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.

19. 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.

20. 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.

21. 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.

22. "Compiler Construction: Principles and Practice" by Kenneth C. Louden

23. 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.

24. 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.

25. 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.

26. 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.

27. Mohan, M.; Greer, D. A survey of search-based refactoring for software maintenance. J. Softw. Eng. Res. Dev. 2018, 6, 3–55.

28. Leeson, M.; Kandola, G. A comparative analysis of software prediction methodologies using simulation techniques. IEEE Software Engineering Transactions 2022, 27, 1014–1022.

29. Zafar, M.I.; Sanders, P.P.; and Schultes, D. Enhancing the efficiency of route planning algorithms in engineering contexts. In Proceedings of the Sixth International

Workshop on Experimental Algorithms, Vol. 4525 LNCS. Springer-Verlag, Berlin/Heidelberg, 2007, 23–36.

30. Spall, J. A primer on Stochastic Search and Optimization. John Wiley & Sons, Inc., New York, 2003.

31. Vallati, M., Fawcett, C., Gerevini, A., Hoos, H.H., and Saetti, A. Deriving domain-specific optimized planners from general parametrized planning algorithms. In Proceedings of the Eighth RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, 2011;

32. Simon, F.; Steinbruckner, F.; Lewerentz, C. Refactoring strategies based on metrics. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, 14–16 March 2001; pp. 30–38.

33. Baqais, A.; Alshayeb, M. Systematic review of automated software refactoring techniques. Software Quality Journal 2020, 28, 459–502.

34. Westfold, S.J. and Smith, D.R. Developing efficient constraint-satisfaction programs. Knowledge Engineering Review 16, 1 (Mar. 2001), 69–84.

35. Whaley, R.C., Petitet, A., and Dongarra, J.J. Optimization strategies for software with the ATLAS project. Parallel Computing 27, 1–2 (Jan. 2001), 3–35.

36. Xu, L., Hoos, H., and Leyton-Brown, K. Hydra: Dynamic configuration of algorithms for portfolio-based decision making. In Proceedings of the 24th AAAI Conference on Artificial Intelligence. AAAI Press, Palo Alto, CA, 2010, 210–216.

37. Xu, L., Hutter, F., Hoos, H.H., and Leyton-Brown, K. SATzilla: Algorithm selection for SAT problems based on portfolio strategies. Journal of Artificial Intelligence Research 32 (MayAug. 2008), 565–606.

38. Tsantalis, N.; Chatzigeorgiou, A. Detection of opportunities for refactoring by introducing polymorphism. Journal of Systems and Software 2022, 83, 391–404.

39. Heged ″us, P.; Kádár, I.R. Ferenc, Gyimóthy T. Empirical assessment of software maintainability using a validated refactoring dataset. Information and Software Technology 2018, 95, 313–327.

40. Gharehchopogh, F.; Gholizadeh, H. An extensive survey: Whale Optimization Algorithm and its practical applications. Swarm and Evolutionary Computation 2019, 48, 1–24.

41. Rosli, M.; Teo, N.H.I.; Yusop, N.S.M.; Mohamad, N.S. Predictive model for web application faults utilizing genetic algorithms. International Conference on Computer and Software Modeling (IPCSIT) 2011, 14, 71–77.