Модуль №2 "Компілятори та утиліти"

Лабораторна робота 2.1
Реалізація граматичного розбору за заданою формальною граматикою

Мета роботи: ознайомитись з основними поняттями теорії регулярних граматик, з призначенням і принципами роботи лексичних аналізаторів (сканерів). Отримати практичні навички побудови сканера.

Завдання роботи: спроектувати та реалізувати лексичний аналізатор на прикладі заданої простої вхідної мови.

Короткі теоретичні відомості

В теорії формальних мов *алфавітом* називається скінченна множина символів. Об'єднання, перетин, різниця алфавітів також є алфавітом.

Формальна мова – це множина скінчених послідовностей символів заданого алфавіту, які описуються правилами певного виду. В теорії формальних мов ці правила отримали назву *формальна граматика*, або просто *граматика*. Формальна граматика – це спосіб опису формальної мови, тобто спосіб виділення деякої підмножини з множини всіх ланцюжків символів (слів, речень), що можуть бути утворені на основі деякого скінченного алфавіту.

Відомі три способи визначення граматик: визначення граматики прямим перерахуванням усіх ланцюжків, що входять до мови; визначення граматики шляхом породження; визначення граматики шляхом розпізнавання (або аналітично). Спосіб перерахування придатний лише для найпростіших мов і тому не має широкого практичного застосування. Визначення граматики шляхом породження полягає у формулюванні правил, за допомогою яких можна побудувати будь-який ланцюжок мови. Аналітичне визначення граматики надає певні алгоритми, які дозволяють в

результаті розгляду будь-якого ланцюжка визначити, належить він до даної мови чи ні.

Формальні граматики і формальні мови можна класифікувати за характером формального апарату, що застосовується для їхнього опису. Відому класифікацію граматик за типами запропонував відомий американський вчений Ноам Хомський (Chomsky).

Побудова лексичних аналізаторів (сканерів). Скінченні автомати.

Компонент транслятора, який виконує лексичний аналіз вхідного тексту, називають *лексичним аналізатором, розпізнавачем*, або *сканером*. Алгоритм роботи простого сканера можна описати так:

- відбувається перегляд вхідного потоку символів програми на початковій мові до виявлення чергового символу, що обмежує лексему;
- для вибраної частини вхідного потоку виконується функція розпізнавання лексеми;
- при успішному розпізнаванні інформація про виділену лексему заноситься в таблицю лексем, і алгоритм повертається до першого етапу;
- при неуспішному розпізнаванні видається повідомлення про помилку, а подальші дії залежать від реалізації сканера: або його виконання припиняється, або робиться спроба розпізнати наступну лексему (йде повернення до першого етапу алгоритму).

Робота програми-сканера продовжується до тих пір, поки не будуть проглянуті всі символи програми на початковій мові з вхідного потоку. Важливо відзначити, що знаки операцій самі також є лексемами і необхідно не пропустити їх при розпізнаванні тексту.

Розпізнавачами для регулярних мов є скінченні автомати (СА; англ. – finite automata). Існують правила, за допомогою яких для будь-якої регулярної граматики може бути побудований СА, що розпізнає ланцюжки мови, заданої цією граматикою.

Будь-який СА може бути заданий за допомогою п'яти параметрів: $M(Q,\Sigma,\delta,q_0,F)$, де:

Q – скінченна множина станів автомата;

Σ – скінченна множина допустимих вхідних символів (вхідний алфавіт СА);

δ – задане відображення множини Q·Σ в множину підмножин P(Q): δ: Q·Σ → P(Q) (іноді δ називають функцією переходів автомата);

$q_0 \in Q$ – початковий стан автомата;

$F \subseteq Q$ – множина заключних (термінальних) станів автомата.

Іншим способом опису СА є граф переходів – графічне представлення множини станів і функції переходів СА. Граф переходів СА – це навантажений односпрямований граф, в якому вершини представляють стани СА, дуги відображають переходи з одного стану в інший, а символи навантаження (позначки) дуг відповідають функції переходу СА. Якщо функція переходу СА передбачає перехід з стану q в q' по декількох символах, то між ними будується одна дуга, яка позначається всіма символами, по яких відбувається перехід з стану q в q'.

Недетермінований СА незручний для аналізу ланцюжків, оскільки в ньому можуть зустрічатися стани, що допускають неоднозначність. Тобто, на графі такого автомата присутні такі стани-вершини, з яких виходить дві або більш дуги, помічені одним і тим же символом. Очевидно, що програмування роботи такого СА – нетривіальна задача. Для простоти програмування функціонування СА $M(Q,\Sigma,\delta,q_0,F)$ він повинен бути детермінованим – в кожному з можливих станів цього СА для будь-якого вхідного символу функція переходу повинна містити не більше ніж один стан:

$$\forall a \in V,\ \forall q \in Q\text{: або } \delta(a,q) = \{r\},\ r \in Q\text{, або } \delta(a,q) = \varnothing.$$

Доведено, що будь-який недетермінований СА може бути перетворений в детермінований СА так, щоб їх мови співпадали (говорять, що ці СА еквівалентні).

Порядок виконання роботи

1. Ознайомтесь з теоретичними відомостями. Отримайте у викладача варіант завдання – специфікацію граматики, на основі якої необхідно буде побудувати розпізнавач.

2. Відповідно до заданої граматики побудуйте опис СА, який лежить в основі лексичного аналізатора (у вигляді набору множин і функції переходів, або у вигляді графа переходів).

3. Підготуйте у вигляді звіту і представити викладачу (захистити) теоретичні результати виконання.

4. Використовуючи отримані результати, напишіть і відладьте програму, яка виконує лексичний аналіз вхідного тексту відповідно до завдання і породжує таблицю лексем з вказівкою їх типів і значень. Вхідна інформація (текст на вхідній мові) задається у вигляді символьного (текстового) файлу. Програма повинна видавати повідомлення про наявність у вхідному тексті помилок, які можуть бути виявлені на етапі лексичного аналізу.

5. Включіть текст написаної програми як додаток до звіту. Додайте до коду програми відповідні коментарі, які пояснюють кожну дію.

6. Продемонструйте працюючу програму викладачеві і здайте звіт.

7. Дайте відповіді на контрольні запитання.

Контрольні запитання

1. Що таке формальна граматика?

2. Яким чином класифікують формальні граматики?

3. Що таке «контекстно-вільна граматика», «контекстно-залежна граматика»?

4. До якого типу належить граматика, що використовується у вашому завданні? Доведіть свою відповідь.

5. Що таке термінальні і нетермінальні символи граматики?

6. Поясніть, які дії виконує ваша програма в цілому і кожна з її частин.

Література: [3], [6].

Лабораторна робота 2.2
Організація таблиць ідентифікаторів

Мета роботи: вивчити основні методи організації таблиць ідентифікаторів.

Завдання роботи: підготувати програму, яка отримує на вході набір ідентифікаторів, організовує таблиці ідентифікаторів за допомогою заданих методів.

Короткі теоретичні відомості

Принципи організації таблиць ідентифікаторів

Компілятор поповнює записи в таблиці ідентифікаторів по мірі аналізу початкової програми і виявлення в ній нових елементів, що вимагають розміщення в таблиці. Пошук інформації в таблиці виконується всякий раз, коли компілятору необхідні відомості про той або інший елемент програми. Слід відзначити, що пошук елементу в таблиці виконуватиметься компілятором набагато частіше, ніж поміщення в неї нових елементів. Так відбувається тому, що описи нових елементів в початковій програмі, як правило, зустрічаються набагато рідше, ніж ці елементи використо-вуються. Крім того, кожному додаванню елементу в таблицю ідентифікаторів у будь-якому випадку передуватиме операція пошуку – щоб переконатися, що такого елементу в таблиці немає.

На кожну операцію пошуку елементу в таблиці компілятор витрачатиме час, і оскільки кількість елементів в початковій програмі велика (від кількох одиниць до сотень тисяч, залежно від об'єму програми), цей час істотно впливатиме на загальний час компіляції. Тому таблиці ідентифікаторів повинні бути організовані так, щоб компілятор мав можливість максимально швидко виконувати пошук потрібного йому запису таблиці по імені елементу, з яким пов'язаний цей запис.

Можна виділити наступні способи організації таблиць ідентифікаторів:

- □ прості і впорядковані списки;
- □ бінарне дерево;
- □ хеш-адресація з рехешуванням;
- □ хеш-адресація по методу ланцюжків;

□ комбінація хеш-адресації із списком або бінарним деревом.

Побудова таблиць ідентифікаторів за методом бінарного дерева

У цьому методі таблиця зберігається у вигляді бінарного дерева. Кожен елемент таблиці є вузлом дерева. Кореневим вузлом стає перший елемент, зустрінутий компілятором при заповненні таблиці. Дерево називається бінарним, оскільки кожна вершина в ньому може мати не більше двох гілок. Для визначеності називатимемо дві гілки «права» і «ліва».

Розглянемо алгоритм заповнення бінарного дерева. Вважатимемо, що алгоритм працює з потоком вхідних даних, що містить ідентифікатори. Перший ідентифікатор, як вже було сказано, поміщається у вершину дерева. Всі подальші ідентифікатори потрапляють в дерево за наступним алгоритмом:

1. Вибрати черговий ідентифікатор з вхідного потоку даних. Якщо чергового ідентифікатора немає, то побудова дерева закінчена.
2. Зробити поточним вузлом дерева кореневу вершину.
3. Порівняти ім'я чергового ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.
4. Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, якщо дорівнює – припинити виконання алгоритму (двох однакових ідентифікаторів бути не повинно!), інакше – перейти до кроку 7.
5. Якщо у поточного вузла є ліва вершина-нащадок, то зробити її поточним вузлом і повернутися до кроку 3, інакше – перейти до кроку 6.
6. Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину лівою вершиною поточного вузла і повернутися до кроку 1.
7. Якщо у поточного вузла існує права вершина-нащадок, то зробити її поточним вузлом і повернутися до кроку 3, інакше – перейти до кроку 8.
8. Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину правою вершиною поточного вузла і повернутися до кроку 1.

Хеш-функції і хеш-адресація

У реальних початкових програмах кількість ідентифікаторів буває настільки великою, що навіть логарифмічну залежність часу пошуку від їх кількості не можна визнати задовільною. Необхідні більш ефективні методи пошуку інформації в таблиці ідентифікаторів. Кращих результатів можна досягти, якщо застосувати методи, пов'язані з використанням хеш-функцій і хеш-адресації.

Хеш-функцією F називається деяке відображення множини вхідних елементів R на множину цілих невід'ємних чисел Z: F(r) = n, r R, n Z.

Хеш-адресація полягає у використанні значення, повернутого хеш-функцією, в якості адреси елементу деякого масиву даних.

Проблема вибору хеш-функції не має універсального розв'язання. Хешування зазвичай відбувається за рахунок виконання над ланцюжком символів деяких простих арифметичних і логічних операцій.

Найпростішою хеш-функцією для символу є код внутрішнього представлення в комп'ютері літери символу. Цю хеш-функцію можна використовувати і для ланцюжка символів, обираючи перший символ в ланцюжку.

Очевидно, що така примітивна хеш-функція не буде задовільною, бо при її використанні виникне проблема – двом різним ідентифікаторам, що починаються з однієї і тієї ж букви, відповідатиме одне і те ж значення хеш-функції. Тоді при хеш-адресації в один і той же елемент таблиці ідентифікаторів потрібно буде помістити два різні ідентифікатори, що неможливо. Ситуація, коли двом або більшій кількості ідентифікаторів відповідає одне і те ж значення хеш-функції, називається *колізією*.

Порядок виконання роботи

1. Отримайте варіанти завдання у викладача.
2. Виберіть і опишіть відповідну хеш-функцію.
3. Опишіть структури даних, використовувані для заданих методів організації таблиць ідентифікаторів.

4. Побудуйте опис скінченного автомата, який лежить в основі лексичного аналізатора (у вигляді набору множин і функції переходів або у вигляді графа переходів).

5. Підготуйте у вигляді звіту і представте викладачу на захист теоретичні результати виконання пп. 2, 3, 4.

6. Використовуючи отримані результати, напишіть і відладьте програми на ЕОМ.

7. Внесіть до коду програм відповідні коментарі, які пояснюють кожну дію.

8. Здайте працюючі програми викладачеві.

9. Включіть тексти написаних програм як додаток до звіту.

Контрольні запитання та завдання

1. Що називають ідентифікатором?
2. Що називають таблицею ідентифікаторів? Яке її призначення?
3. Які способи організації таблиць ідентифікаторів Вам відомі?
4. Поясніть поняття «хеш-функція».
5. Що таке хеш-адресація?

Література: [3], [6].

Лабораторна робота 2.3
Переклад виразів у зворотній польський запис

Мета роботи: ознайомитись з поданням алгебраїчних виразів у прямій та зворотній польській нотаціях та у вигляді бінарного дерева.

Завдання роботи: розробити, налагодити та дослідити алгоритми трансляції арифметичних виразів в прямий та зворотний польський запис, а також подання їх у вигляді бінарного дерева.

Короткі теоретичні відомості

Відомі три форми запису виразів – *префіксна*, *інфіксна* і *постфіксна*. При використанні префіксної форми запису операція

записується перед своїми операндами (наприклад, + 2 3), при використанні інфіксної – між операндами (наприклад, 2 + 3), а при використанні постфіксної – після операндів (наприклад, 2 3 +).

Загальноприйнятий («шкільний») запис арифметичних виразів використовує інфіксну форму. Префіксна форма запису виразів, у якій всі оператори пишуть безпосередньо перед операндами, отримала назву «прямої польської форми запису». Постфіксну форму запису, у якій всі оператори пишуть безпосередньо перед операндами, стали називати «зворотнім (або інверсним) польським записом». Звичайно, всі три форми запису виражають одну і ту ж послідовність дій, тому будь-який вираз може бути переведений з однієї з цих форм у іншу, наприклад, з інфіксної форми в постфіксну або навпаки.

Структуру арифметичного виразу зручно відображати в ієрархічному вигляді. Традиційною формою подання таких даних стало дерево. При використанні середовищ програмування C++Builder або Embarcadero RAD Studio для візуального подання дерева зручно використати компонент TreeView бібліотеки VCL. Цей компонент можна знайти на вкладці Win32 Палітри Компонентів. Додайте його на форму, а для відображення даних з обраних вузлів дерева створіть компоненту типу Richedit з тієї ж вкладки. Для налаштування властивостей дерева використовуйте редактор властивостей компоненту TreeView (рис.1). На рис. 2 показано приклад вигляду форми після додання у дерево кількох вузлів.

Настроювання дерева можна виконувати і програмним способом. Ось приклад коду, що реалізує додавання вузла:

```
int n = TreeView1->Selected->AbsoluteIndex;
TTreeNode *Node1 = TreeView1->Items->Item[n];
TreeView1->Items->AddChild(Node1,"ChildNode");
Node1->Selected=true;
```

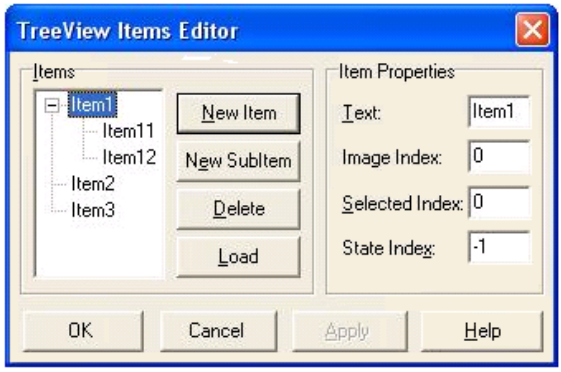

Рис. 1. Редактор властивостей компоненту TreeView.

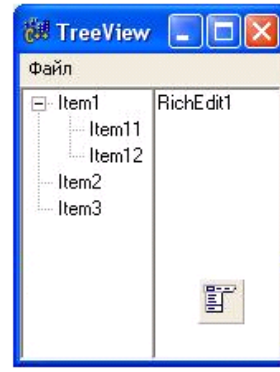

Рис. 2. Приклад використання компоненту TreeView на формі.

Порядок виконання роботи

1. Розробіть граматику, яка породжує арифметичні вирази, що містять числа, знаки арифметичних дій та дужки. Множини термінальних та нетермінальних символів цієї граматики можуть включати, наприклад, такі компоненти: Число; Додавання; Віднімання; Множення; Ділення; Ліва дужка; Права дужка; Вираз. Подайте розроблену граматику у вигляді синтаксичних форм Бекуса-Наура та у вигляді синтаксичних діаграм Вірта.
2. Відповідно до розробленої граматики, створіть алгоритм розпізнавача, який проводитиме аналіз введеного виразу.
3. Створіть проект CBuilder. Розмістсть на формі три поля для виразів у інфіксній, префіксній і постфіксній нотаціях, а також компоненти TreeView та RichEdit для відображення результатів розбору. Додайте також кнопку, що запускатиме виконання алгоритму перекладу.
4. Підключіть до проекту розроблений алгоритм розбору. Додайте до нього блок, який відобразіть результати розбору у компоненті TreeView.

5. Додайте до програми алгоритм, який на основі введених у TreeView даних згенерує рядки, що містять префіксну і постфіксну форми виразу.

6. Оформіть звіт та дайте відповіді на контрольні запитання.

Контрольні запитання

1. Що називається формальною граматикою?
2. Що називається поданням граматики у формі Бекуса-Наура?
3. Що називається діаграмами Вірта? Які правила їх створення Ви знаєте?
4. Яким чином визначають тип формальної граматики? До якого типу належить розроблена Вами граматика? Доведіть свою відповідь.
5. Що називається термінальними і нетермінальними символами граматики?
6. На основі яких міркувань Ви визначили, які з символів Вашої граматики мають належати до термінальних, а які – до нетермінальних?
7. Які форми запису арифметичних виразів Ви знаєте? Чому їх можлива кількість саме така?
8. Як виконується подання арифметичного виразу у зворотному польському записі?
9. Як виконується обчислення виразу, поданого в зворотному польському записі?
10. Як Ви вважаєте, у чому полягає перевага використання польського інверсного запису для подання виразів у компіляторах?

Література: [3], [6].

Лабораторна робота 2.4
Декомпіляція

Мета роботи: ознайомитись з призначенням і принципами роботи декомпіляторів; отримати практичні навички їх використання.

Завдання роботи: провести декомпіляцію програми в середовищах Windows та Linux.

Короткі теоретичні відомості

Декомпілятор – це програма, що транслює виконуваний модуль (отриманий на виході компілятора) у відносно еквівалентний вихідний код на мові програмування високого рівня. *Декомпіляція* – це процес відтворення вихідного коду програми декомпілятором. Частковим випадком декомпіляції є дизасемблювання: дизасемблер транслює виконуваний модуль програми в текст на мові асемблера.

Успішність декомпіляції залежить від обсягу інформації, наявної в коді, що декомпілюється. Найкраще декомпілюються програми, написані на Java, Visual Basic, FoxPro, .NET – мовах, які транслюються у байт-код, тобто у команди, що виконуються не безпосередньо процесором, а віртуальною машиною. Причина цього полягає в тому, що інструкції віртуальної машини у більшості випадків відповідають операторам початкової мови написання програми, тільки з деякою оптимізацією. Зокрема, програми, написані для середовища виконання .NET, можуть бути відновлені із скомпільованих файлів майже із 100% точністю – завдяки неймовірній кількості надлишкової інформації, що зберігається в їх exe-файлах.

На другому місці за простотою декомпіляції знаходяться програми, створені в середовищах Delphi, C++ Builder і RAD Studio. Ці програми у більшості випадків компілюються в звичайний native-код, який не використовує віртуальні машини (за винятком лише платформи .NET). Однак ці середовища програмування широко використовують власні стандартні бібліотеки на кшталт VCL і залишають в exe-файлах багато зайвої інформації; зокрема, там можна знайти всі ідентифікатори з тексту вхідної програми. Використовуючи цю інформацію, можна також досить успішно відновити початковий код.

Оптимізований машинний код вважається найбільш складним джерелом для декомпіляції, оскільки він зазвичай містить значно менше супутньої інформації.

Спеціалісти, що працюють в середовищі Windows, мають в своєму розпорядженні найбагатший вибір засобів декомпіляції.

Такі засоби розроблені практично для кожного формату файлів та середовища розробки. В якості прикладу розглянемо EMS Source Rescuer – це утиліта, яка дозволяє частково відновити загублений вихідний код програми, створеної в середовищі C++ Builder, Borland Delphi, або RAD Studio. Для відновлення початкових текстів програми досить мати її виконуваний модуль. EMS Source Rescuer дозволяє відновити всі модулі і форми складного проекту, а також всі задані події і властивості. Отриманий код буде не повним: фактично EMS Source Rescuer дозволяє відновити лише загальну структуру коду програми. Відновлені процедури – обробники подій не мають тіла, а натомість містять вказівник на відповідну адресу у файлі виконуваного коду. Тим не менше, навіть автоматизація відновлення загальної структури програми дозволяє програмісту заощадити до 90% часу, що витрачається на відновлення коду.

Декомпіляція під Linux має свою специфіку і свої тонкощі, недостатньо висвітлені в доступній літературі. До того ж під Linux не розроблено розвинених інструментів для декомпіляції, як під Windows. Тому навіть простий захист програми під Linux може стати нездоланною проблемою для хакерів. Розглянемо деякі з декомпіляторів під Linux.

Objdump – це утиліта для виконуваних файлів формату ELF (Executable and Linking Format), яка містить також простенький дизасемблер. Вона вимагає обов'язкової наявності section table в коді і не може обробляти упаковані файли.

Dasm2 – це скрипт-дизасемблер, що автоматизує використання утиліти objdump. Отже, за відсутності objdump dasm2 працювати не може. Ще однією обов’язковою вимогою є наявність у системі робочої версії Perl.

Bastard disassembler – дизассемблер, написаний для Linux/Unix платформ. Він працює з різними форматами файлів, має підсвічування синтаксису, відображення рядка зовнішніх посилань, та інші можливості. Цей декомпілятор може використовуватися в інтерактивному режимі або за допомогою командного рядка. При цьому дизасемблер отримує файли для роботи через stdin, зберігає всю отриману інформацію в пам’яті і друкує її в консолі за запитом користувача.

Decomp – декомпілятор, який працює тільки з об’єктними файлами (.o або .obj). Він використовує інформацію таблиці

символів, що є в цих файлах, щоб знайти точки входу до функцій, визначити дані, які використовуються в програмі, і імена цих даних

Порядок виконання роботи

1. Ознайомтесь з теоретичними відомостями.
2. Запропонуйте ідею простої програми, виконуваний файл якої Ви будете декомпілювати. Обговоріть свої плани з викладачем.
3. Напишіть цю програму в середовищі Delphi, C++ Builder або RAD Studio. Отримайте виконуваний файл програми.
4. За допомогою утиліти EMS Source Rescuer виконайте процес декомпіляції виконуваного файла.
5. Збережіть отриманий результат декомпіляції.
6. Розгляньте отриманий код. Спробуйте віднайти у ньому елементи алгоритмів та структур даних, які були Вами закладені в початковий варіант програми.
7. Спробуйте скомпілювати і запустити отриманий код. Чи вдасться це зробити без його допрацювання?
8. З'ясуйте, як впливає на результат декомпіляції опція компілятора «включити в exe-файл відладочну інформацію». Для цього змініть стан відповідного флажка в опціях середовища розробки, ще раз скомпілюйте початкову програму і повторіть з отриманим виконуваним файлом пп. 4–7 завдання.
9. Спробуйте виконати декомпіляцію програми в будь-якому декомпіляторі під Linux.
10. Оформіть звіт про виконану роботу. Включіть до звіту початковий код програми, коди, отримані в результаті декомпіляції, результати використання отриманого коду, а також скриншоти роботи з декомпілятором.
11. Здайте звіт викладачеві і дайте відповіді на контрольні питання.

Контрольні питання

1. Поясніть поняття «декомпіляція»..
2. Яке практичне застосування можуть мати програми, що виконують декомпіляцію?

3. Що спільного має код, отриманий в результаті декомпіляції, з початковим кодом програми? В чому полягає різниця між цими кодами?

4. Чому для запуску коду, отриманого в результаті декомпіляції, може знадобитися його ручне доопрацювання?

5. Чи можливо виконати декомпіляцію програм з візуальним інтерфейсом?

6. Охарактеризуйте юридичні аспекти застосування декомпіляції.

Література: [1], [6].

Лабораторна робота 2.5
Створення інсталятора додатку

Мета роботи: ознайомитись з принципом роботи інсталяторів додатків та основними етапами їх створення.

Завдання роботи: отримання практичних навичок створення інсталяторів за допомогою BitRock InstallBuilder.

Короткі теоретичні відомості

Деякі комп'ютерні програми написані і скомпільовані таким чином, що їх можна встановити простим копіюванням їх файлів в будь-яке зручне розміщення у файловій системі. Про такі програми кажуть, що вони не потребують встановлення. Цей же вираз також застосовується щодо деяких програмних файлів, які самі по собі не є програмами (наприклад, деяких плагінів, драйверів, тощо). Існують навіть дистрибутиви операційних систем (Windows, Linux, FreeBSD та інших), які не вимагають встановлення, а можуть бути безпосередньо запущені з завантажувального CD, DVD або USB, не справляючи при цьому впливу на інші ОС, встановлені на комп'ютері користувача. (Такі завантажувальні диски часто називають Live CD.)

Тим не менше, більшість програмних продуктів і дистрибутивів операційних систем потребують інсталяції.

Інсталяція програмного забезпечення – це процес встановлення програмного забезпечення на комп'ютер кінцевого користувача.

Більшість програм поставляються і розповсюджуються в стиснутому (упакованому) вигляді. Для нормальної роботи вони мають бути розпаковані, після цього компоненти програм мають бути правильно розміщені на комп'ютері, причому з урахуванням можливих відмінностей між різними комп'ютерами, різними операційними системами і налаштуваннями різних користувачів. У процесі встановлення виконуються різні тести на відповідність заданим вимогам, система необхідним чином конфігурується (настроюється) для зберігання файлів і даних, необхідних для правильної роботи програми.

Встановлення програмного забезпечення (ПЗ), як правило, включає в себе розміщення всіх необхідних програмі файлів у відповідних місцях файлової системи, а також модифікацію і створення конфігураційних файлів. В процесі встановлення ПЗ також може виконуватись створення або зміна:

- спільно використовуваних програмних файлів;
- директорій;
- записів конфігураційних файлів, використовуваних однією програмою, або спільно;
- ключів реєстру операційної системи;
- посилань або ярликів.

При встановленні ПЗ також необхідно виконати контроль залежностей, тобто перевірити, чи є в системі необхідні для роботи даної програми пакети чи компоненти. В разі успішного встановлення слід зареєструвати новий пакет у списку доступних пакетів операційної системи.

Описані компоненти та дії є досить складними для виконання вручну, і до того ж можуть розрізнятися для різних програм і для різних комп'ютерних систем. Тому, з метою автоматизації, інсталяція ПЗ виконується спеціальною системною програмою. Така програма може бути компонентом операційної системи (менеджер встановлення ПЗ, пакетний менеджер – наприклад, RPM, apt, Yum в GNU/Linux, Windows Installer в Microsoft Windows), або ж входити до складу пакету того програмного забезпечення, яке встановлюється.

Користувачі операційної системи GNU/Linux часто використовують систему GNU toolchain для компіляції програмного забезпечення безпосередньо перед встановленням. В цьому випадку використовуються засоби встановлення ПЗ, що входять до складу GNU toolchain.

Дистрибутиви операційних систем поставляються разом з універсальним або спеціальним інсталятором.

BitRock InstallBuilder є інструментом для створення кроссплатформенних інсталяторів для настільного і серверного програмного забезпечення. За допомогою InstallBuilder можна швидко створювати інсталятори професійного рівня для Linux, Windows, Mac OS X, Solaris і інших платформ. При цьому використовується спеціальне середовище для створення інсталятора, і, в якості вхідних даних, – файли проекту програми, для якої створюється інсталятор. BitRock InstallBuilder може створювати менеджери встановлення для Windows з підтримкою кількох мов, програми видалення встановленого ПЗ, генерувати rpm- і deb- пакети для GNU/Linux і мультиплатформенні CD/DVD диски. Присутня також функція оновлення, яка дозволяє генерувати пакети для встановлення автоматичних оновлень програмного забезпечення в системах, де це програмне забезпечення уже встановлене.

Порядок виконання роботи

1. Ознайомтесь з теоретичними відомостями.
2. Встановіть BitRock InstallBuilder на свій комп`ютер. Запустіть програму та ознайомтесь з онлайн-довідкою. Рекомендується також ознайомитись з посібником користувача програми BitRock InstallBuilder на сайті продукту [http://installbuilder.bitrock.com]
3. За допомогою середовища розробки C++ Builder створіть просту програму, яка складається з основного виконуваного компонента (exe), динамічно пов’язаної з ним бібліотеки (dll), використовує конфігураційний файл та який-небудь «саморобний» ключ в реєстрі Windows.
4. У програмі BitRock InstallBuilder запустіть процес створення інсталятора. Відповідаючи на запитання майстра, створіть інсталятор написаної Вами програми.

5. Після успішного створення істалятора, перевірте його на працездатність, встановивши програму на іншій системі. Запустіть встановлену програму і перевірте коректність її роботи.

6. Заплануйте і виконайте тести, які доведуть, що інсталятор коректно встановив допоміжні елементи проекту – бібліотеку і конфігураційний файл, коректно створив необхідний ключ реєстра, і що встановлена програма дійсно може їх використовувати.

7. Підготуйте звіт з виконаної роботи. Включіть до звіту: а) відомості про написану програму; б) скріншоти процесу створення інсталятора і пояснення до них; в) скріншоти встановлення програми зі створеного інсталятора; г) опис заходів з перевірки коректності встановлення програми.

8. Здайте звіт викладачеві і дайте відповіді на контрольні питання.

Контрольні питання

1. Що таке інсталяція програмного забезпечення?
2. Якими засобами може виконуватися інсталяція програмного забезпечення?
3. Для чого призначена програма BitRock InstallBuilder?
4. Що є вхідною і вихідною інформацією для програми BitRock InstallBuilder?
5. З яких основних етапів складається створення інсталятора за допомогою програми BitRock InstallBuilder?
6. Яким чином можна перевірити коректність роботи створеного Вами інсталятора?

Література: [1], [2], [3], [4], [7].