ЛАБОРАТОРНА РОБОТА 1

Технології розробки системних програм для POSIX сумісних (сертифікованих) операційних систем

Мета лабораторної роботи полягає у набутті знань, умінь та навичок з технології розроблення системного програмного забезпечення (ПЗ) з використанням мови С і С++ для POSIX сумісних (сертифікованих) операційних систем. Під час використання С (С++) як мови програмування застосовуються знання архітектури мікропроцесора, загальної будови операційної системи та прийнятного стилю вихідного коду.

Також лабораторна робота дає основні навички користування ОС Linux на рівні оболонки.

Робота виконується з використанням бібліотек мови С (С++) для POSIX ОС та автоматизації розробки з використанням Makefile. Для виконання курсу ЛР потрібен комп'ютер, на якому встановлена POSIX операційна система (ОС) або віртуальна машина з POSIX OS або система контейнерізації. Особливих вимог до потужності персонального комп'ютера (ПК) не визначається.

Перелік обладнання для виконання курсу лабораторних робіт 1–5. Для підготовки ЛР № 1–5 може бути використаний будь-який сучасний ПК на основі мікропроцесора AMD64 (Intel® 64) або ARM. Передбачається для використання POSIX сумісна (сертифікована) операційна система. Вихідний код протестований для операційної системи Linux.

Завдання для лабораторної роботи 1

1. Відповідно до завдання і тематичної спрямованості, що надано у додатку 1, користуючись вихідним кодом додатку 2, створити класи, що реалізують тематичну спрямованість додатку 1.

2 Під час розробки навчальної програми для кожного класу створити заголовочний файл і файл реалізації.

3 Створити файл main.cpp для екземплярів класу на купі пам'яті і на стеку. Провести тестування класів на етапі виконання. Для цього скористатися утилітою Makefile, текстовим редактором або середовищем розробки. Запуск реалізувати у POSIX сумісній ОС.

Програма проведення експерименту лабораторної роботи 1

1. Створення вихідного коду. Користуючись одним з текстових редакторів, створити файл вихідного коду лабораторної роботи. Зберегти його у робочому каталозі, що містить програмну. Доробити вихідний код відповідно до поданого завдання.
2. У разі появи помилок внести зміни до вихідного коду.
3. Після успішного виконання п. 1, 2 запустити програму на виконання.
4. Після покрокового виконання програми зробити скриншоти екранів і створити звіт, додаючи до звіту скриншоти екранів.
5. Зробити висновки щодо особливостей виконання лабораторної роботи, наявності специфічних особливостей роботи програми.

Теоретичні відомості для ЛР 1

Системне програмування є широкою галуззю програмування, один з напрямків якої передбачає розробку операційних систем. Також до цього напрямку програмування відноситься розробка програмного забезпечення (ПЗ) або програмних систем (ПС), що є елементами ОС. На теперішній час для вирішення практичних завдань із розробки вбудованих систем, серверів, тощо доцільно використовувати POSIX (Portable Operating System Interface for uniX) ОС.

Операційні системи POSIX є багатозадачними багатокористувацькими системами з розділенням часу. Це можуть бути різні ОС, сертифіковані за стандартами POSIX, або суміжними з цим стандартом, що мають однакову оболоку, інтерфейси ядра, схему інтерфейсів користувача, програмний інтерфейс (інтерфейс системних викликів), а також схожі принципи побудови архітектури ОС у цілому.

До операційних систем POSIX можна віднести: UNIX та її варіанти, Linux, Solaris, тощо. На теперішній час розповсюдження Linux значною мірою зумовлено відкритим програмним кодом. Ця ОС успішно застосовується на персональних комп’ютерах, великих серверах, потужних обчислювальних кластерах, у мережному обладнанні (маршрутизаторах), смартфонах, і навіть у вбудованих пристроях.

Як звичайні ОС, операційні системи POSIX мають у своєму складі ядро ОС, що виконує певний перелік завдань. Крім того ці ОС мають командну оболонку, роль якої можуть виконувати традиційні командні інтерпретатори – sh, ksh , csh, tcsh для bash, тощо.

Галузь системного програмування охоплює розробку ПО на рівні ядра і оболонки. У першому варіанті використовуються мова програмування С(С++), у другому варіанті використовуються мова типова мова оболонки або мова програмування Python.

Теоретичні відомості щодо основ технології програмування GCC

Установка

sudo apt update

sudo apt-get install gcc.

sudo apt install make

sudo apt-get install build-essential

Будуть інсталльовані наступні елементи

1.cpp

2.gcc-5-locales

3.g++-multilib

4.g++-5-multilib

5.gcc-5-doc

6.gcc-multilib

7.autoconf

8.automake

9.libtool

10.flex

11.bison

12.gdb

13.gcc-doc

14.gcc-5-multilib

15.and many.

Послідовність використання компілятору GCC для розробки програми включає кроки, що показані далі. З використанням текстового редактору і створюємо файл hello.c, що містить текст, наприклад:

```
/ * Hello.c * /
#include <stdio.h>
```

```
int main (void){
printf ( "Hello World \ n");}
```

Даний код написаний на мові програмування С [1]. Щоб відкомпілювати hello.c достатньо набрати у командному рядку наступне :

```
$ gcc -o hello hello.c
```

Якщо вихідний код написаний без синтаксичних помилок, то компілятор завершить свою роботу без будь-яких повідомлень, в результаті утворюється файл - hello. Цей файл містить машинний код програми (executable files) або є інша назва - бінарніки (binary files).

Опція -o компілятора GCC вказує на те, яким має бути ім'я вихідного файлу. Якщо не вказати опцію -o, то бінарнику буде присвоєно ім'я a.out. Щоб запустити отриманий бінарник набираємо в командному рядку наступну команду:

```
$ ./hello
Hello World
```

Для мультіфайлового програмування потрібно зробити компонування або лінковку декількох файлів. Отже, щоб відкомпілювати мультіфайловую програму, треба спочатку отримати об'єктний код з кожного вихідного файлу окремо. Кожен такий код буде представляти собою об'єктний модуль. Кожен об'єктний модуль записується в окремий об'єктний файл. Потім об'єктні модулі треба скомпонувати в один бінарник з точкою входу.

У Linux для лінкування використовується програма ld. Програма GCC самостійно викликає компоновщик з потрібними опціями. Для прикладу створимо перший файл з ім'ям main.c:

```
/ * Main.c * /
int main (void){
print_hello ();
}
```

Тепер створимо ще один файл, що містить бібліотечну функцію і назва файлу hello.c:

```
/ * Hello.c * /
#include <stdio.h>
void print_hello (void){
printf ( "Hello World \ n");
}
```

Функція main () викликає функцію print_hello (), що знаходиться в іншому файлі. Опція -c компілятора GCC змушує його відмовитися від лінковки після компіляції. Якщо не вказувати опцію -o, то в імені об'єктного файлу розширення .c буде замінено на .o (звичайні об'єктні файли мають розширення .o):

```
$ gcc -c main.c
$ gcc -c hello.c
$ ls
hello.c hello.o main.c main.o
```

Отже, ми отримали два об'єктних файла. Тепер їх треба об'єднати в один бінарник:

```
$ gcc -o hello main.o hello.o
$ ls
hello * hello.c hello.o main.c main.o
$ ./hello
Hello World
```

Компілятор "побачив", що замість вихідних файлів (з розширенням .c) йому надаються об'єктні файли (з розширенням .o) і відреагував згідно ситуації: викликав лінковщік з потрібними опціями.

Особливість об'єктних файлів полягає у тому, що вони зберігають в таблиці символів імена деяких позицій (глобально оголошених функцій, наприклад). В процесі лінковки відбувається стикування імен та перерахунок позицій, що дозволяє декільком об'єктним файлів об'єднатися до одного бінарника. Якщо викликати nm для файлу hello.o, то побачимо наступну картину:

```
$ Nm hello.o
U printf
00000000 T print_hello
```

Важливим є те, що в об'єктному файлі збереглася інформація про імена. Своя інформація є і в файлі main.o:

```
$ Nm main.o
00000000 T main
        U print_hello
$
```

Розглянемо сутність автоматичної збірки. У попередньому розділі для створення бінарників з двох вихідних файлів потрібно набрати три команди. Утиліта make, що працює за своїми власними сценаріями дає можливість автоматизувати процес виконання команд. Сценарій записується в файлі з ім'ям Makefile і поміщається в репозиторій (робочий каталог) проекту. Формат Makefile використовується не тільки на Unix-системах, а в інших ОС. Процес виконання make, називають складанням проекту, а сама утиліта make відноситься до розряду програм — збирачів проектів.

Файл Makefile складається з трьох елементів: коментарі, макроозначення і цільові зв'язки (або просто зв'язки). У свою чергу зв'язки складаються теж з трьох елементів: мета, залежнсті та правила. Сценарії make використовує однорядкові коментарі, що починаються з літери # (решітка). Макроси Makefile схожі з макроконстантамі мови С. Простий приклад Mikefile у загальному вигляді наданий далі:

мета: залежності

[tab] команда

Разом це можна описати так:

- що потрібно зробити (мету);

- що для цього потрібно (залежності);

- як це зробити (правила). У якості мети виступає ім'я або макроконстанта. Залежності - це список файлів і цілей, розділених між собою. Правила - це команди, що передаються оболонці.

Розглянемо приклад сценарію збірки для розглянутого в попередньому розділі мультіфайлового проекту Hello World. Для цього створіть файл з ім'ям Makefile, зміст якого наданий далі:

```
# Makefile for Hello World project

hello: main.o hello.o
    gcc -o hello main.o hello.o

main.o: main.c
    gcc -c main.c

hello.o: hello.c
    gcc -c hello.c

clean:rm  -f  *.o  hello
```

Для мови програмування C++ аналогічний файл має наступний синтаксис:

```
all: hello

hello: main.o lib1.o lib2.o
	g++ main.o lib1.o lib2.o -o hello

main.o: main.cpp
	g++ -c main.cpp

lib1.o: lib1.o.cpp
	g++ -c lib1.o.cpp

lib2.o: lib2.cpp
	g++ -c lib2.cpp

clean: rm -rf *.o hello
```

Зверніть увагу, що в кожному рядку перед викликом gcc(g++), а також в рядку перед викликом rm стоять табуляції. Формат Makefile вимагає, щоб кожне правило починалося з табуляції.

Makefile може починатися як з великої так і з малої літери. Прийнято починати з великої букви, щоб він не перемішувався з іншими файлами проекту, а стояв "у списку перших". У даному типі файлу може бути використані змінні, наприклад:

```
# Це коментар, де використана змінна CC=g++
#У змінной CFLAGS знаходяться прапори
CFLAGS=-c -Wall
all: hello

hello: main.o lib1.o lib2.o
	$(CC) main.o lib1.o lib2.o -o hello

main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp
```

```
lib1.o: lib1.cpp
	$(CC) $(CFLAGS) lib1.cpp

	lib2.o: lib2.cpp
	   $(CC) $(CFLAGS) lib2.cpp

clean: rm -rf *.o hello
```

Перший рядок - коментар, що починається з символу # (решітка) і закінчується символом нового рядка. Далі по порядку йдуть чотири зв'язки: зв'язка для компонування об'єктних файлів main.o і hello.o; зв'язка для компіляції main.c; зв'язка для компіляції hello.c; зв'язка для очищення проекту.

Перша зв'язка має мету hello. Мета відділяється від списку залежностей двокрапкою. Список залежностей відділяється від правил символом нового рядка. А кожне правило починається на новій рядку з символу табуляції. Користуючись змінними можна реалізувати файл у наступному вигляді:

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp lib2.cpp lib1.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
		$(CC) $(LDFLAGS) $(OBJECTS) -o $@

 .cpp.o:
		$(CC) $(CFLAGS) $< -o $@
```

Очищення проекту буває потрібним в декількох випадках: 1) для очищення готового проекту від всього зайвого; 2) для збирання проекту (коли в проект додаються

нові файли або коли змінюється сам Makefile; 3) в будь-яких інших випадках, коли потрібна повна зібрання (наприклад, для вимірювання часу повного складання).

Формат запуску утиліти make наступний:

make [опції] [мети …].

Опції make нам поки не потрібні. Якщо викликати make без визначення мети, то буде виконана зв'язка (з усіма залежностями) і збірка завершиться:

```
$ make
gcc -c main.c
gcc -c hello.c
gcc -o hello main.o hello.o
$ ls
hello * hello.c hello.o main.c main.o Makefile
$ ./hello
Hello World
$
```

Для очищення проекту потрібно викликати команду:

```
$ Make clean
rm -f * .o hello
$ ls
hello.c main.c Makefile
$
```

В даному випадку ми вказали мету безпосередньо в командному рядку. Цільова зв'язка clean містить порожній список залежностей, виконується тільки одне правило. Потрібно "чистити" проект, коли змінюється список вихідних файлів або коли змінюється сам Makefile.

Питання до ЛР1

1. Опис каталогів POSIX сумісних (сертифікованих) ОС.
2. Основні команди оболонки для управління і роботи у POSIX сумісних (сертифікованих) ОС.
3. Права доступу до файлів у типових POSIX сумісних (сертифікованих) ОС. Способи відображення і описання.
4. Основні команди для всіх етапів розробки програм у GCC компіляторі.
5. Основні правила розробки Makefile.

Тематична спрямованість для створення класів ЛР 1. Додаток 1

№1: Інформаційна система Бібліотеки.

Клас

Працівники (Код співробітника, ПІБ, Вік, Пол, Адреса, Паспортні дані, Код посади) [10 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і на стеку.

№2: Інформаційна система Радіостанції.

Клас

Посади (Код посади, Найменування посади, Оклад, Обов'язки, Вимоги) [5 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і на стеку.

№3: Інформаційна система таксопарку

Клас

Автомобілі (Код автомобіля, Код марки, Реєстраційний номер, Номер кузова, Номер двигуна, Рік випуску, Пробіг, Код шофѐра співробітника, Дата останнього ТО, Код-механіка співробітника, Спеціальні позначки) [10 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і

на стеку.

№4: Інформаційна система Туристичного агентства

Клас

Готелі (Код готелю, Найменування, Країна, Місто, Адреса, Кількість звѐзд, Контактна особа) [10 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і на стеку.

№5: Інформаційна система Страхової компанії

Клас

Поліси (Номер поліса, Дата початку, Дата закінчення, -імость Сто, Сума виплати, Код виду поліса, Відмітка про виплату, Відмітка про закінчення, Код клієнта, Код співробітника) [10 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і на стеку.

№6: Інформаційна система Шлюбного агентства

Клас

Клієнти (Код клієнта, ПІБ, Пол, Дата народження, Вік, Зростання, вага, Кількість дітей, Сімейний стан, Шкідливі при-вичкі, Хобі, Опис, Код знака, Код відносини, Код націо-нальності, Адреса, Телефон, Паспортні дані , Інформація про партнѐре) [10 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і на стеку.

№7: Інформаційна система Сервіс центру

Клас

Види несправностей (Код виду, Код моделі, Опис, Симптоми, Методи ремонту, запчастини Код 1, Код запчастини 2, 3 Код запчастини, Ціна роботи) [5 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і на стеку.

№8: Інформаційна система Школи

Клас

Класи (Код класу, Код співробітника-класного керівника, Код виду, Кількість учнів, Буква, Рік навчання, Рік створення) [5 записів].

Передбачити функціональність додавання інформації, вивід інформації на консоль.

Передбачити функціональність поведінки класу, в залежності від значення певних членів класу.

Під час тестування на етапі виконання створити екземпляри класу на купі пам'яті і на стеку.

Додаток 2. Інформаційна система. Містить 5 файлів.

// Database.h

```
#include <iostream>
#include "Employee.h"

namespace Records {

  const int kMaxEmployees = 100;
  const int kFirstEmployeeNumber = 1000;
    class Database
      {
      public:
          Database();
          ~Database();

          Employee& addEmployee(std::string inFirstName, std::string inLastName);
          Employee& getEmployee(int inEmployeeNumber);
          Employee& getEmployee(std::string inFirstName, std::string inLastName);
          void       displayAll();
          void       displayCurrent();
          void       displayFormer();
      protected:
          Employee    mEmployees[kMaxEmployees];
          int         mNextSlot;
          int         mNextEmployeeNumber;
      };
}
```

// Database.cpp

```
#include <iostream>
#include <stdexcept>
#include <string>

#include "Database.h"

using namespace std;

namespace Records {

  Database::Database()
  {
    mNextSlot = 0;
    mNextEmployeeNumber = kFirstEmployeeNumber;
  }
  Database::~Database()
  {
  }
  Employee& Database::addEmployee(string inFirstName, string inLastName)
  {
    if (mNextSlot >= kMaxEmployees) {
      cerr << "There is no more room to add the new employee!" << endl;
      throw exception();
```

```
  }

  Employee& theEmployee = mEmployees[mNextSlot++];
  theEmployee.setFirstName(inFirstName);
  theEmployee.setLastName(inLastName);
  theEmployee.setEmployeeNumber(mNextEmployeeNumber++);
  theEmployee.hire();

  return theEmployee;
}
Employee& Database::getEmployee(int inEmployeeNumber)
{
  for (int i = 0; i < mNextSlot; i++) {
    if (mEmployees[i].getEmployeeNumber() == inEmployeeNumber) {
        return mEmployees[i];
    }
  }

  cerr << "No employee with employee number " << inEmployeeNumber << endl;
  throw exception();
}

Employee& Database::getEmployee(string inFirstName, string inLastName)
{
  for (int i = 0; i < mNextSlot; i++) {
    if (mEmployees[i].getFirstName() == inFirstName &&
          mEmployees[i].getLastName() == inLastName) {
        return mEmployees[i];
    }
  }

  cerr << "No match with name " << inFirstName << " " << inLastName << endl;
  throw exception();
}
void Database::displayAll()
{
  for (int i = 0; i < mNextSlot; i++) {
    mEmployees[i].display();
  }
}

void Database::displayCurrent()
{
  for (int i = 0; i < mNextSlot; i++) {
    if (mEmployees[i].getIsHired()) {
        mEmployees[i].display();
    }
  }
}

void Database::displayFormer()
{
```

```
    for (int i = 0; i < mNextSlot; i++) {
      if (!mEmployees[i].getIsHired()) {
          mEmployees[i].display();
      }
    }
  }
}
```

// Employee.h

```
#include <iostream>

namespace Records {
  const int kDefaultStartingSalary = 30000;
  class Employee
    {
    public:

      Employee();

      void     promote(int inRaiseAmount = 1000);
      void     demote(int inDemeritAmount = 1000);
      void     hire();     // hires or re-hires the employee
      void     fire();     // dismisses the employee
      void     display();  // outputs employee info to the console

      // Accessors and setters
      void          setFirstName(std::string inFirstName);
      std::string   getFirstName();
      void          setLastName(std::string inLastName);
      std::string   getLastName();
      void          setEmployeeNumber(int inEmployeeNumber);
      int           getEmployeeNumber();
      void          setSalary(int inNewSalary);
      int           getSalary();
      bool          getIsHired();
    private:
      std::string   mFirstName;
      std::string   mLastName;
      int           mEmployeeNumber;
      int           mSalary;
      bool          fHired;
    };
}
```

// Employee.cpp

```
#include <iostream>
#include <string>

#include "Employee.h"
```

```
using namespace std;

namespace Records {

 Employee::Employee()
 {
  mFirstName = "";
  mLastName = "";
  mEmployeeNumber = -1;
  mSalary = kDefaultStartingSalary;
  fHired = false;
 }
 void Employee::promote(int inRaiseAmount)
 {
  setSalary(getSalary() + inRaiseAmount);
 }

 void Employee::demote(int inDemeritAmount)
 {
  setSalary(getSalary() - inDemeritAmount);
 }
 void Employee::hire()
 {
  fHired = true;
 }

 void Employee::fire()
 {
  fHired = false;
 }
 void Employee::display()
 {
  cout << "Employee: " << getLastName() << ", " << getFirstName() << endl;
  cout << "-------------------------" << endl;
  cout << (fHired ? "Current Employee" : "Former Employee") << endl;
  cout << "Employee Number: " << getEmployeeNumber() << endl;
  cout << "Salary: $" << getSalary() << endl;
  cout << endl;
 }
 // Accessors and setters

 void Employee::setFirstName(string inFirstName)
 {
  mFirstName = inFirstName;
 }

 string Employee::getFirstName()
 {
  return mFirstName;
 }

 void Employee::setLastName(string inLastName)
```

```
  {
    mLastName = inLastName;
  }

  string Employee::getLastName()
  {
    return mLastName;
  }

  void Employee::setEmployeeNumber(int inEmployeeNumber)
  {
    mEmployeeNumber = inEmployeeNumber;
  }

  int Employee::getEmployeeNumber()
  {
    return mEmployeeNumber;
  }

  void Employee::setSalary(int inSalary)
  {
    mSalary = inSalary;
  }

  int Employee::getSalary()
  {
    return mSalary;
  }

  bool Employee::getIsHired()
  {
    return fHired;
  }

}
```

// UserInterface.cpp

```
#include <iostream>
#include <stdexcept>
#include <string>

#include "Database.h"

using namespace std;
using namespace Records;

int displayMenu();
void doHire(Database& inDB);
void doFire(Database& inDB);
void doPromote(Database& inDB);
void doDemote(Database& inDB);
```

```
int main(int argc, char** argv)
{
  Database employeeDB;
  bool done = false;

  while (!done) {
    int selection = displayMenu();

    switch (selection) {
    case 1:
      doHire(employeeDB);
      break;
    case 2:
      doFire(employeeDB);
      break;
    case 3:
      doPromote(employeeDB);
      break;
    case 4:
      employeeDB.displayAll();
      break;
    case 5:
      employeeDB.displayCurrent();
      break;
    case 6:
      employeeDB.displayFormer();
      break;
    case 0:
      done = true;
      break;
    default:
      cerr << "Unknown command." << endl;
    }
  }
}

int displayMenu()
{
  int selection;

  cout << endl;
  cout << "Employee Database" << endl;
  cout << "-----------------" << endl;
  cout << "1) Hire a new employee" << endl;
  cout << "2) Fire an employee" << endl;
  cout << "3) Promote an employee" << endl;
  cout << "4) List all employees" << endl;
  cout << "5) List all current employees" << endl;
  cout << "6) List all previous employees" << endl;
  cout << "0) Quit" << endl;
  cout << endl;
```

```
  cout << "---> ";

  cin >> selection;

  return selection;
}

void doHire(Database& inDB)
{
  string firstName;
  string lastName;

  cout << "First name? ";
  cin >> firstName;
  cout << "Last name? ";
  cin >> lastName;

  try {
    inDB.addEmployee(firstName, lastName);
  } catch (std::exception ex) {
    cerr << "Unable to add new employee!" << endl;
  }
}

void doFire(Database& inDB)
{
  int employeeNumber;

  cout << "Employee number? ";
  cin >> employeeNumber;

  try {
    Employee& emp = inDB.getEmployee(employeeNumber);
    emp.fire();
    cout << "Employee " << employeeNumber << " has been terminated." << endl;
  } catch (std::exception ex) {
    cerr << "Unable to terminate employee!" << endl;
  }
}

void doPromote(Database& inDB)
{
  int employeeNumber;
  int raiseAmount;

  cout << "Employee number? ";
  cin >> employeeNumber;

  cout << "How much of a raise? ";
  cin >> raiseAmount;

  try {
```

```
    Employee& emp = inDB.getEmployee(employeeNumber);
    emp.promote(raiseAmount);
  } catch (...) {
    cerr << "Unable to promote employee!" << endl;
  }
}
```