The `elevator` Program

A Visual Test of Simple C/C++ Coroutines

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	June 15, 2021	First version of elevator as a literate program.	CWRC

Contents

Introduction

This document describes the elevator program source code and provides instructions for its execution.

The elevator program is a visual and interactive test program for Simple C/C++ Coroutines using Codecraft's open-source sccor library. elevator runs in an ncurses[1] Terminal window as a command-line executable.

The sccor Library's Simple C/C++ Coroutines implementation supports lightweight cooperative multitasking and provides for asynchronous programming through the use of Edison[2]-inspired, single-threaded, non-preemptive, ring-based coroutines.

This version of elevator is an x86_64 executable running on macOS or Windows (Cygwin). A Linux version will be available in a future release, with the availability of a Linux version of the sccor library.

How to Read This Document

This document is a literate program document. As such it includes a complete description of both the design and implementation of the elevator test program for simple C/C++ coroutines. Further information on the particular literal programming syntax used here is given in Literate Programming.

Background

Twenty years ago, Leon Starr published a case study, *Executable UML - The Elevator Project* [ls-elevator], in which he packaged detailed models and documentation for a working elevator control application. He chose the elevator as the subject matter of his case study because, as he stated, "you already *know* how it works - certainly from a user perspective and to some extent from an engineering perspective." Leon's book is an excellent exploration of XUML modeling techniques, and is highly recommended, although the CDs and provided tooling are somewhat outdated.

At about the same time, I was looking for a good vehicle to demonstrate and test my Simple C/C++ Coroutines. As I read Leon's book, I realized that his elevators would provide a great milieu for demonstrating and testing my coroutines. An elevator simulation demonstration would provide a more realistic and challenging subject matter than the darts and abc programs[3] I had previously used as coroutine test programs. So I developed this elevator program, but utilizing a completely different approach than Leon used, namely using coroutines rather than XUML for the implementation.

The elevator program was initially developed as a test vehicle for the sccor library. Subsequently, I adapted the elavator program to be the unit under test (UUT) in a demonstration of the Repeatable Random Test Generation (RRTGen) framework[4]. Consequently, the elevator program presented here has dual personalities: the ability to execute as a standalone elevator simulatiion, with human interaction driving the simulation, but also as the UUT for an RRTGen application, with commands received from the RRTGen application via named pipes.

In implementing elevator as a literate program, I hope to explain the design and logic of the program in an order and fashion that facilitates your understanding of the implementation, as well as providing all of the code.

Hopefully the elevator program code presented here demonstrates how easy it can be to create a moderately complex application, with many cooperative tasks, through the use of coroutines. Note that these coroutines are just standard C++ procedures, with the simple addition of a couple of coroutine statements from the sccor library:

- cobegin, to initialize coroutine execution and put one or more coroutines on the multitasking ring, and
- coresume, to perform an unconditional task switch to yield execution to the other coroutines on the ring, as appropriate to maintain the behavior and performance profile of the ensemble of executing coroutines.

The cobegin statement blocks further execution of the calling routine (usually main) while the coroutine instances created by cobegin are executing. When all coroutine instances have finished their execution, the routine that issued the cobegin statement continues its execution in a normal manner.

For a trivial example, below is a coroutine, repeatChar, that writes its input character a specified number of times and returns. After each character is written, repeatChar yields via a coresume statement.

[1] new curses, a programming library providing an API that allows the programmer to write text-based user interfaces in a terminal-independent manner.
[2] [pbh-edison] is an edition of *Software Practice and Experience* devoted entirely to the Edison papers.
[3] See [cc-darts-program] for my darts literate program.
[4] A literate program for RRTGen will be published in 2021.

```
void repeatChar( char c, int count ) {
   for ( i = 0; i < count; i++ ) {
      putchar( c ) ;
      coresume() ;
   }
}
```

When executed as the only coroutine instance, with input character ‘a’ and a count of 10, repeatChar produces this string of 10 a’s on stdout:

```
aaaaaaaaaa
```

When two instances of repeatChar are executed together, the first with input ‘a’ and a count of 10 (as before) and a second with input ‘b’ and also a count of 10, their interleaved output is:

```
abababababababababab
```

Each instance of repeatChar acts as an independent task, ouputting its designated character. By issuing a coresume after outputting its character, the instance allows any another instance to do its thing, in this case outputting its character. This leads to the string of interspersed a’s and b’s of the result.

Here’s the cobegin statement to start these two instances:

```
cobegin( 2,                             // start 2 coroutines
            repeatChar, 2, 'b', 10,     // 2 parameters ('b' and '10')
            repeatChar, 2, 'a', 10      // 2 parameters ('a' and '10')
) ;
```

The second instance executes first, since the coroutines are stacked by cobegin until it completes its initialization.

Besides cobegin and coresume, the sccor library provides a few optional statements:

- invoke adds another coroutine to the ring of currently-executing coroutines,
- wait delays a coroutine’s execution for at least a specified number of milliseconds while continuing other coroutines,
- waitEx waits for at least a specified number of milliseconds while continuing other coroutines; the waiting period is interrupted if a specified boolean becomes false, and
- when provides a conditional task switch, continuing other coroutines until a specified boolean becomes true.

Note that there is no need for a special "coroutine exit" or "coroutine return" command to complete execution of a coroutine. Coroutines complete execution by the ordinary C/C++ procedure behavior, either by a return statement or just "falling off" the end of the function.

Tip

In our case a coroutine is just an ordinary C/C++ procedure which contains at least one coresume statement.

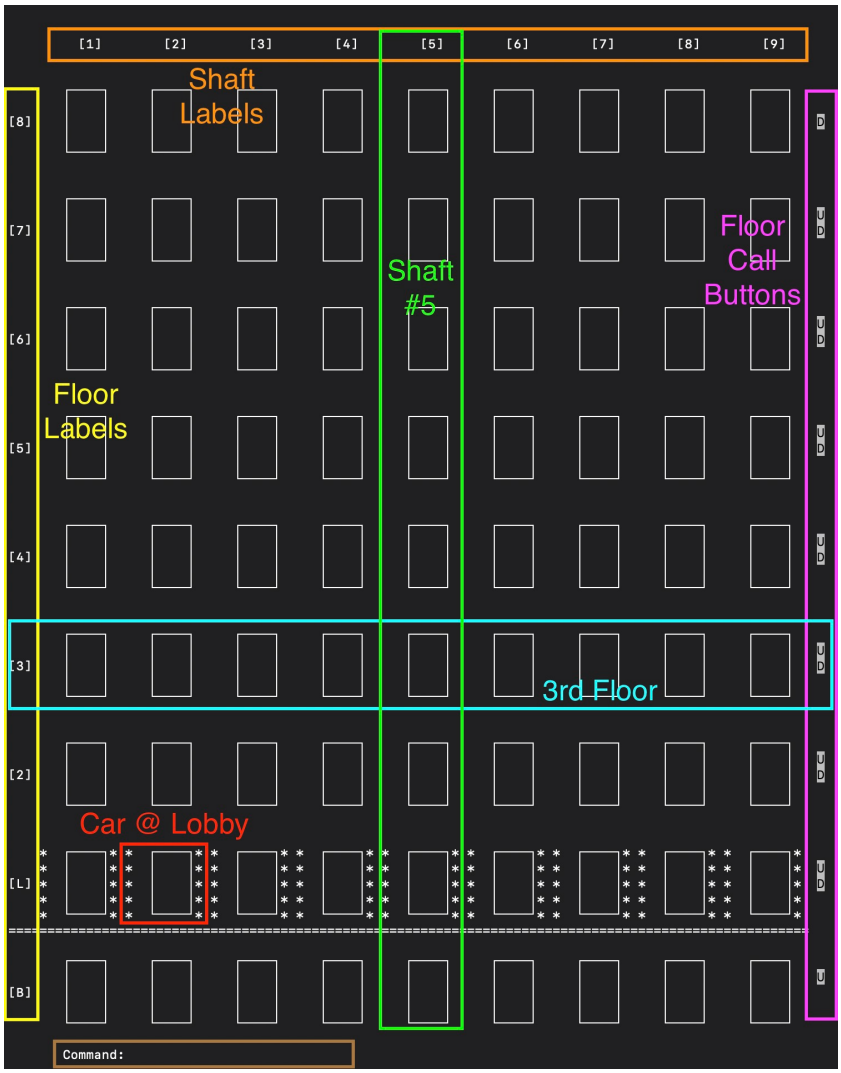
We’ll see examples of these coroutine statements in the implementation of the elevator program.

Overview

The elevator program simulates a bank of elevators in a building having a user-specified number of floors and elevator shafts. The simulation supports from one to nine elevator shafts and from two to nine floors.

Elevator Elements and Basic Terminology

Here is an annotated screenshot[5] of a building with a full complement of floors (9) and shafts (9):



Floors

Floors are depicted as horizontal arrays of elevator doors. Each floor has a label.

The 3rd floor in the illustration is highlighted in blue.

Floor Labels

The floor labels are on the left, highlighted in yellow in the above illustration.

[5]The colored lines and text are just for expository purposes. The actual elevator screen is rather drab.

Floor Call Buttons

Each floor has an associated pair of **floor call buttons**, "U" for up and "D" for down. These are located on the right side and highlighted in pink in the illustration.

A floor call button lights up when pressed by a simulated person waiting for an elevator car to stop at the person's floor.[6]

Shafts

Shafts are depicted as vertical arrays of elevator doors.

Shaft Labels

Each shaft has a label, from 1 to the number of shafts. The shaft labels are on the top, highlighted in orange in the illustration.

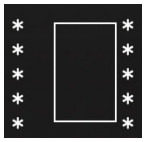
Shaft #5 in the illustration is highlighted in green.

Elevator Cars

Elevator cabins, or "**cars**", run up and down in their respective shafts. We assume a one-to-one relationship between an elevator car and its shaft. We can, therefore, refer to a car by its shaft number, since each car runs exclsively in one shaft.

An elevator car is depicted in a quasi x-ray manner, showing its position via a column of left- and right-side bracketing asterisks that let us "see" the car behind the wall of the building[7]. In the illustration, car number 2 (*i.e.*, in shaft #2) is outlined in red. Car number 2 is currently at the Lobby floor "[L]", as are all of the other elevator cars.

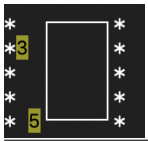
Here is an enlarged look at an elevator car:



The asterisks move as the car travels up and down its shaft to show the car's motion and position.[8]

In-Car Call Buttons

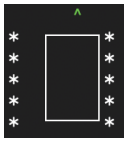
Two left-hand columns inside each elevator car show **in-car call buttons** indicating floors where the car has been requested to stop, in our case by a simulated elevator passenger in the car having pressed a floor button.[9] A car call button "lights up" when the corresponding floor has been requested as a stop destination for that car. For example, here we show two stop requests, for the 3rd floor "[3]" and for the 5th floor "[5]":



[6] We'll see the floor stop request command as part of the elevator simulation user interface.
[7] Simple, but, hey, this is just a character-based graphical presentation.
[8] The (external) doors remain stationary.
[9] We'll see the car call button command as part of the elevator simulation user interface.

Direction Indicator

When a car stops at a floor, a **direction indicator** above the door lights to indicate the car's current direction, a green "^" for up or a red "v" for down. Here we have a car arriving at a floor with the up indicator lit:



The direction indicator stays lit while the car's door opens, passengers leave and enter the car, and the car's door closes. The direction indicator turns off when the door is locked after closing and prior to any movement.

Ground Level

Ground level is shown as a row of equal signs ("===...===="). In the example, the floor above ground level has label "[L]" for Lobby. The floor below ground has label "[B]" for Basement. Successive floors above the Lobby are labeled [2] - [8]. At the beginning of the simulation, all cars start in the floor just above ground ("[L]" in this case).

Command Area

There is a **command area** at the bottom where user-keyed input appears. The command area is highlighted in brown in the illustration.

`elevator` Execution Instructions

1. Run the `elevator` executable[10] in a macOS[11] or Windows (Cygwin) Terminal.

   The size of the Terminal screen must be sufficient to display the entire building. For purposes of this demonstration, a building with 6 floors and 7 elevator shafts will be simulated, requiring a Terminal window size of at least 83 colmns x 45 rows.[12]

   The `elevator` executable requires two parameters, the number of floors and the number of elevator shafts.

   Syntax:

   ```
   % elevator -floors [<count>|<labels>] -n <elevators> [-pipes] [-wait]
   ```

   Options may be abbreviated to one letter following the dash, *e.g.*, "`-f [<count>|<labels>]`".

   Help is available by specifying a "`-h`" option.

   The `-pipes` option causes the simulation to communicate via named pipes (*vs.* using normal console input). We will not be using the `-pipes` option in this demonstration.

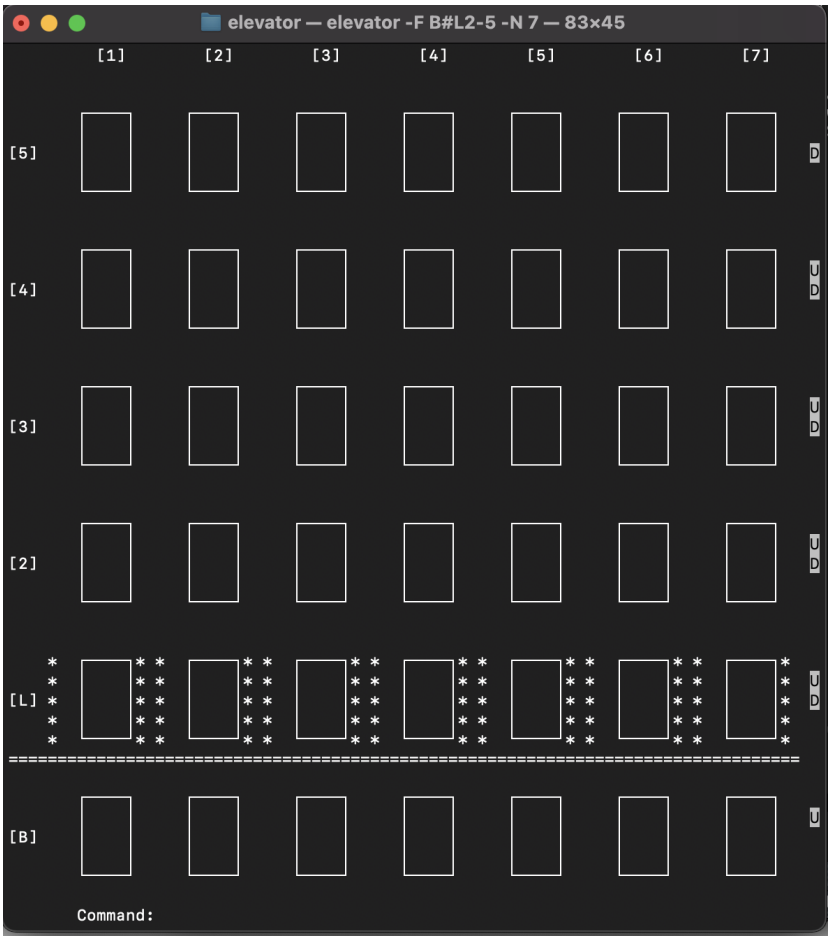
   The `-wait` option causes the simulation to pause prior to the simulation to display some execution details. We will not be using the `-wait` option in this demonstration.

   For this demonstration, enter:

   ```
   % elevator -f B#L2-5 -n 7
   ```

   creating a simulation of 7 elevator shafts in a building having 6 floors, labeled *B*, *L*, *2*, *3*, *4*, and *5*.

[10] See [cc-elevator-mac] and [cc-elevator-win] for macOS and Windows (Cygwin) executables, respectively.
[11] Big Sur (11.0), or later, is supported.
[12] Set Terminal's Window Size in Terminal > Preferences... > Profiles > Window.



Note that the floor labeled *L* is the ground floor. The floor after the "#" in the floor labels string is the ground floor. All elevator cars start the simulation on the ground floor (here the Lobby Floor).

2. You can exercise the `elevator` program with these commands available during the simulation:

`c<floor><car>`	pushes <floor> button in <car>.
`f<floor><up|down>`	pushes <floor> call up or down button.
`xnn<enter>`	speeds up the simulation by a factor of nn (1 - 10).
`q`	ends the simulation.

Examples:

`cb4`	sends car 4 to the basement.
`f3u`	pushes the up button on floor 3.

For example, key "cb4" to send car 4 to the Basement.[13]

You will see the "[B]" car button light up and car 4 will descend to the Basement. Upon arriving, the up direction indicator "^" will light (in green), the car will make a "ding" sound[14], and the elevator doors will open. Once the doors are fully open, in about 2 seconds, they will remain open for about 5 seconds to allow for simulated passengers' entering and exiting the car. Then the doors will close and the direction indicator will turn off.

3. You can see several cars moving by keying multiple commands in succession: "c53" will send car 3 to floor 5, "c42" will send car 2 to floor 4, *etc*.

[13] No Enter key is necessary for your inputs, which appear in the Command area as you type. If you type an invalid key for the command, it is just ignored.
[14] Actually, on my MacBook Pro it is a "tock" rather than a "ding".

4. You can cause a car to stop at multiple floors: key "cl7", then "c27", "c37", "c47", "c57", "cb7". Car 7 will proceed up to floor 5, opening the door at the Lobby (where it is currently), then stopping at floors 2, 3, 4, and 5, before descending to the Basement.

5. You can send all cars to a given floor by keying "*" for the car number. For example, send all cars to the Basement by keying "cb*".

6. You can simulate pressing a floor call button by keying, for example, "f3u" to push the up button of floor 3.

   When a floor button is pushed, the corresponding up or down indicator lights up on the right hand side of the floor. The indicator stays lit until a car arrives at the floor headed in the indicator's direction.

7. Having played with the elevator simulation for a while, the impatient user may enter the command "x5" (followed by the Enter key, in just the case of the "x" command, due to its variable-length one- or two-digit parameter) to speed up the simulation by a factor of 5. For example, you might send all cars to the Basement ("cb*"), speed up the simulation by a factor of 5 ("x5" and Enter), and then send all cars to the 5th floor ("c5*").

   Voila! Turbo elevators !

   In fact the simulation may be sped up by a factor of between 1 and 10.

   To resume normal speed, enter "x1" (remembering to add the Enter key).

8. That's about all there is to the simulation.[15]

   Rinse and repeat *ad nauseam*.

9. Terminate the elevator session by keying "q".

[15] When executing with a test program communicating via pipes, the elevator program can be driven to a sufficient level of activity that it provides a more entertaining performance.

Implementation

A primary consideration in developing the elevator program is to create an interesting demonstration using the Simple C/C++ Coroutines in the sccor library.

Another high-level choice is to implement elevator as a command-line program, primarily for the coding simplicity and to minimize extraneous GUI aspects.

Because we want to show the elevators moving in real-time, we are using the ncurses library, given its excellent performance and relative ease in interfacing with the library's graphical commands.

Besides supporting user keying of commands to drive the elevator simulation, the implementation needs an inter-process communication mechanism to support commands sent from a test program, like the Elevator Verification Test (EVT) program[16] based on the RRTGen framework. For this inter-process communication, the elevator program uses named pipes. Named pipes are available on the principal platforms where elevator will run: macOS, Windows Cygwin, and Linux. Named pipes are fast and simple to use and describe, with file-like characteristics.

Having chosen our implementation platforms, language (C++), the use of coroutines, the ncurses library for character graphics, and named pipes for inter-process communications, we proceed now to some application-specific requirements.

Requirements

There are several requirements that are important in designing and implementing the elevator program. For convenience in referencing them in the discussion below, they are each given an ID, "ERn", for "Elevator Requirement n".

- [ER1] The program shall accept input from keyboard as well as from another process via named pipes.

  The program must support both human interaction as well as machine-driven testing capabilities in order to support both *ad hoc* and RRTGen testing.

- [ER2] Although its elevators shall behave as described in Leon's book, the elevator program shall be designed and written independently from Leon's XUML implementation.

  This is standard practice and allows for better testing than would be possible with the same implementation in the UUT and testing program. Since the elevator program is implemented using coroutines, and the RRTGen test program is an implementation of Leon's XUML models, there is a good chance that an implementation defect in one will not occur in the other, allowing our testing to detect anomolies.

- [ER3] The elevator program shall allow the user to specify the number of floors, a floor label for each floor, the ground floor level, and the number of elevator shafts in the building via command-line parameters.

  When being executed as a stand-alone, keyboard-driven application, the elevator program requires specifying the number of elevator shafts and the number of floors in the simulated building with command-line parameters.

  When being executed as a piped-based UUT, the number of elevators and the number of floors may be specified via command-line parameters, or, if omitted, they will be generated by the test program.

Characteristic	Default Value	Mininum Value	Maximum Value
Number of Elevators	N/A	1	9
Number of Floors	N/A	2	9

[16] A literate program for EVT will be published in 3Q 2021.

- [ER4] For machine-driven testing, these additional physical characteristics shall be settable:

Characteristic	Description	Default Value	Mininum Value	Maximum Value
Block Clear Time	Time to wait after a door is blocked before attempting to close it again (in seconds)	N/A	3 s	10 s
Door Open Close Time	Time required to open or close (without obstruction) the elevator door (in seconds)	2 s	2 s	5 s
Normal Door Wait Time	Time to keep the doors open while waiting for passengers to transfer into or out of a car (in seconds)	5 s	5 s	15 s
Floor Height	Distance between floors (in meters)	3.048 m	3 m	5 m
Max Cabin Velocity	Maximum velocity of the car (in meters/second)	1.048 m/s	0.5 m/s	2 m/s
Max Close Attempts	Maximum number of consecutive attempts allowed when trying to close an obstructed door	N/A	1	20
Min Stopping Distance	Distance required to safely decelerate the car to come to a stop (in meters)	0.1 m	0.1 m	10 m

- [ER5] The program shall accept commands to:
  - "push an up or down floor call button" on any floor
  - "push an in-car call button" for any floor in any elevator car
  - "obstruct a door" in any elevator car to keep it from closing
  - end the simulation
- [ER6] The program shall provide real-time simulation feedback via the pipe interface when running in machine-driven testing mode, in order to allow verification of the elevator program behavior by comparison with values from the test program (likely, the RRTGen model):
  - Car Location
  - Door Is Ajar
  - Door Is Closed
  - Door Is Locked
  - Door Is Open
  - Floor Call Is Up
  - Floor Call Is Down
  - Floor Down Is Cleared
  - Floor Up Is Cleared
  - Indicator Is Down
  - Indicator Is Off
  - Indicator Is Up
  - Stop Is Cleared
  - Stop Is Requested

We will see how each of these requirements is met as we proceed.

First, here is the command-line syntax to execute the elevator program.

elevator Program Execution Command-line Syntax

The **elevator** command executes an elevator simulation in a Terminal window.

```
% elevator -floors [<count>|<labels>] -n <elevators> [-pipes] [-wait] [-help]
```

-floors
is a required option. Either <count> or <labels> may be specified.

<count>
is the number of floors in the simulated building.

<labels>
is a string of characters representing the floor labels. A # character may be included to locate the ground floor.

-n
is a required option.

<elevators>
specifies the number of elevator shafts in the simulated building.

-pipes
causes the simulation to communicate via named pipes (*vs.* using normal console input).

-wait
causes the simulation to pause prior to the simulation to display some execution details for debugging.

-help
produces a simple help output, then exits the application.

Options may be abbreviated to one letter following the dash, *e.g.*, "`-f [<count>|<labels>]`".

For example, "`-f B#L2-5`" specifies six floors, *B*, *L*, *2*. *3*. *4*. and *5*, with *L* as the ground floor.

From the execution syntax you will notice that there are two modes for exercising `elevator`, with keyboard input or with input through pipes from a test process.

elevator's main Routine

The `elevator` program has a rather standard `main` routine: set up, do something, and take down.

```
<<main routine>>=
int main ( int argc, char *argv[] ) {
<<set up>>
<<do simulation>>
<<take down>>
}
```

We'll see the "do something" later, in section main's Elevator Simulation, followed by the "take down", in section main's Take Down.

First, we'll look at the "set up".

main's Set Up

If the user has requested help, we just show our help text and exit.

We allow several different ways to specify help. The help request may appear anywhere in the command-line options, and specifying help will cause `elevator` to ignore any other options.

Processing a Help Request

For uppercasing a string, we have a handy utility function since it isn't included in Unix string.h.

```
<<forward references>>=
char *strupr( char *string ) ;
```

```
<<set up>>=
for ( int i = 1; i < argc; i++ ) {
   if ( !strcmp( strupr( argv[ i ] ), "?" )
            || !strcmp( argv[ i ], "-?" )
            || !strcmp( argv[ i ], "-H" )
            || !strcmp( argv[ i ], "-HELP" )
            || !strcmp( argv[ i ], "HELP" ) ) {
      cout << "\r\nelevator is a simulation of 1 to 9 elevator shafts in\r\n"
              "in a building with from 2 to 9 floors.\r\n\r\n" ;
      cout << "Syntax:" << "\r\n\r\n" ;
      cout << "  elevator -floors [<count>|<labels>] -n <elevators> "
              "[-pipes] [-wait]\r\n\r\n" ;
      cout << "The -pipes option causes the simulation to communicate via "
              "\r\nnamed pipes (vs. using normal console input).\r\n\r\n" ;
      cout << "The -wait option causes the simulation to pause prior to "
              "\r\nthe simulation to display some execution details.\r\n\r\n" ;
      cout << "For example:\r\n\r\n" ;
      cout << "  elevator -floors B#L2-4 -n 3 -pipes\r\n\r\n" ;
      cout << "creates a simulation of 3 elevator shafts in a building\r\n" ;
      cout << "having five floors, labeled 'B', 'L', '2', '3', and '4'.\r\n" ;
      cout << "In this example, the floor labeled 'L' is the ground floor."
           << "\r\n" ;
      cout << "Named pipes are to be used for communications with a testing "
              "process.\r\n\r\n" ;
      cout << "Commands available during the simulation:\r\n\r\n" ;
      cout << "  c<floor><car>      pushes <floor> button in <car>.\r\n" ;
      cout << "  f<floor><up|down>  pushes <floor> call up or down button."
              "\r\n" ;
      cout << "  q                  ends the simulation.\r\n" ;
      cout << "  xnn<enter>         speeds up the simulation by a factor of "
              "nn (1 - 10).\r\n\r\n" ;
      cout << "Examples:\r\n";
      cout << "  cb4                sends car 4 to the basement ('b').\r\n" ;
      cout << "  f3u                pushes the up button on floor 3."
              "\r\n\r\n" ;

      exit( 411 ) ;
   }
}
```

Otherwise, if help is not requested, then there is some housekeeping to do.

Setting Up Name and Version

First, we'll set up our name and version for sharing with a testing process, should we be running in that mode. Our name is taken as the name of the executable on the command line.

`elevator`'s current version is 1.0.

```
<<constants>>=
const char APP_VERSION[] = "1.0" ;
```

We provide an ample limit for the length of our name (taken from the command line) and a version nul-terminated string, "<executable_name> vv.rr".

```
<<pipe constants>>=
const int MAX_NAME_AND_VERSION_LEN = 25 ;
```

We need access to the shared pipe interface.

```
<<include files>>=
#include "pipe_interface.h"
#include "pipe_commands.cpp"
```

```
<<global variables>>=
char ourNameAndVersion[ MAX_NAME_AND_VERSION_LEN ] ;
```

We calculate the maximum name length that can fit in `ourNameAndVersion` as `maxNameLen`.

```
<<main variables>>=
int maxNameLen = MAX_NAME_AND_VERSION_LEN - strlen( APP_VERSION ) - 2 ;
```

We store the name (possibly truncated, if necessary to fit) and version, or give the user an error message and exit if the command-line name of the executable is too long, even with truncation.

```
<<set up>>=
<<main variables>>
if ( maxNameLen > 0 ) {
   if ( strlen( argv[ 0 ] ) > maxNameLen ) {
      argv[ 0 ][ maxNameLen ] = '\0' ;
   }
   sprintf( ourNameAndVersion, "%s %s", argv[ 0 ], APP_VERSION ) ;
} else {
   char str[ 80 ] ; // ample
   sprintf( str, "Error: APP_VERSION string is too long: '%s'.", APP_VERSION ) ;
   cout << str << "\r\n" ;
   exit( 911 ) ;
}
```

Continuing with the command line, we now handle the other options specifiable by the user.

Processing Command Line Options

We'll loop through each string on the command line. For each string, we determine which option it is and then process its parameter, if any. For example, if the user specifies `"-floors B#L2-4"`, we compare `"-F"` with the first two characters (as uppercase), find a match, and set the boolean `waitingForFloors` to true to cause processing of the floor labels string `"B#L2-4"` on the next loop iteration.

Thus, the code is not dependent on character case or on any particular order of command-line options.

If the user specifies `"-floors"` with a digit, we set the `numberOfFloors` variable to the interger value. Otherwise, we copy the floor label string to the `floorSpecification` variable for subsequent processing.

The number of elevator shafts specified by the user with the `"-n"` option is saved in the `numberOfElevators` variable.

If the user specifies using pipes (with `"-p"`), we record that fact in the `useNamedPipes` variable.

If the user specifies waiting (with `"-w"`), we set the `waiting` boolean to true.

Here are the definitions and initial values for these variables.

```
<<constants>>=
const int MAX_NUMBER_OF_FLOORS = 9 ;
```

```
<<global variables>>=
bool useNamedPipes ;
int numberOfElevators ;
int numberOfFloors ;
```

These are local variables for main:

```
<<main variables>>=
bool waitingForFloors = false ;
bool waitingForNumberOfElevators = false ;
bool waiting = false ;
char floorSpecification[ MAX_NUMBER_OF_FLOORS + 1 + 1 ] ;  // ❶
floorSpecification[ 0 ] = '\0';
```

❶ Allowing room for # and trailing nul char.

main initializes the global variables it uses.

```
<<constants>>=
const int MAX_ALERT_MESSAGE_SIZE = 128 ;
const int DEFAULT_DOOR_REMAINS_OPEN_TIME = 5000 ;                    // ❶
const int DEFAULT_DOOR_TRANSIT_TIME = 1000 ;                         // ❷
const int MAX_DOOR_HEIGHT = 7 ;                                      // ❸
const int DEFAULT_POSITION_TRANSITION_TIME = 2000 / MAX_DOOR_HEIGHT ; // ❹
```

❶, ❷ Milliseconds.

❸ Characters.

❹ 285 milliseconds.

```
<<global variables>>=
char alertMessage[ MAX_ALERT_MESSAGE_SIZE ] ;
int doorRemainsOpenTime ;
int doorTransitTime ;
int positionTransitionTime ;
```

When alertMessage is non-empty it contains a text alert message for the user.

```
<<set up>>=
alertMessage[ 0 ] = '\0' ;                                  // ❶
useNamedPipes = false ;
doorRemainsOpenTime = DEFAULT_DOOR_REMAINS_OPEN_TIME ;
doorTransitTime = DEFAULT_DOOR_TRANSIT_TIME ;
numberOfElevators = 0 ;
numberOfFloors = 0 ;
positionTransitionTime = DEFAULT_POSITION_TRANSITION_TIME ;
```

❶ Empty.

Now we process the command-line options:

```
<<set up>>=
for ( int i = 1; i < argc; i++ ) {
   if ( !strncmp( strupr( argv[ i ] ), "-F", 2 ) ) {
      waitingForFloors = true ;
   } else if ( !strncmp( strupr( argv[ i ] ), "-N", 2 ) ) {
```

```
      waitingForNumberOfElevators = true;
   } else if ( !strncmp( strupr( argv[ i ] ), "-P", 2 ) ) {
      useNamedPipes = true ;
   } else if ( !strncmp( strupr( argv[ i ] ), "-W", 2 ) ) {
      waiting = true ;
   } else if ( waitingForFloors ) {
      if ( strlen( argv[ i ] ) >= MAX_NUMBER_OF_FLOORS + 2 ) {
         char str[ 80 ] ; // ample
         sprintf( str, "Error: Too many floor labels: '%s'.", argv[ i ] ) ;
         cout << str << "\r\n" ;
         exit( 911 ) ;
      }
      if ( strlen( argv[ i ] ) == 1 ) {
         if ( isdigit( argv[ i ][ 0 ] ) ) {
            numberOfFloors = atoi( argv[ i ] ) ;
         } else {
            char str[ 80 ] ; // ample
            sprintf( str, "Error: Invalid floor label: '%s'.", argv[ i ] ) ;
            cout << str << "\r\n" ;
            exit( 911 ) ;
         }
      } else {
         strcpy( floorSpecification, argv[ i ] ) ;
      }
      waitingForFloors = false ;
   } else if ( waitingForNumberOfElevators ) {
      numberOfElevators = atoi( argv[ i ] ) ;
      waitingForNumberOfElevators = false ;
   } else {
      char str[ 80 ] ; // ample
      sprintf( str, "Error: Unrecognized parameter specified: '%s'.",
                    argv[ i ] ) ;
      cout << str << "\r\n" ;
      exit( 911 ) ;
   }
}
```

If we are using named pipes to communicate with a testing process, there are a number of `elevator` attributes that the testing process sets during initialization. We'll discuss how we receive configuration data from a testing process below in the next section.

Otherwise, without a testing process, we will be using keyboard input, so we just verify that the inputs provided by the user for "`-floors`" and "`-n`" are acceptable. We'll handle this stand-alone input processing later, in section Input Verification without a Testing Process.

```
<<set up>>=
if ( useNamedPipes ) {
<<set up for named pipes>>
} else {
<<check validity of input without named pipes>>
}
```

Establishing Communications with a Testing Process

If the user has requested that we use named pipes, we establish a pair of named pipes, one pipe (`ElevatorCommandsPipe`) for the testing process to send commands to our `elevator` process and a second pipe (`ElevatorStatusPipe`) for `elevator` to send status information to the testing process.

```
<<pipe definitions>>=
#define ElevatorCommandsPipe "/tmp/ElevatorCommandsPipe"
#define ElevatorStatusPipe   "/tmp/ElevatorStatusPipe"
```

We need the named pipes support.

```
<<include files>>=
#include <sys/stat.h>
```

```
<<set up for named pipes>>=
int retVal = mkfifo( ElevatorStatusPipe, 0666 ) ;
if ( retVal == -1 && errno != EEXIST ) {
   perror( "Error creating the ElevatorStatusPipe" ) ;
   exit( 1 ) ;
}
retVal = mkfifo( ElevatorCommandsPipe, 0666 ) ;
if ( retVal == -1 && errno != EEXIST ) {
   perror( "Error creating the ElevatorCommandPipe" ) ;
   exit( 1 ) ;
}
```

Now we open the pipes, notifying the user that we are going to block while waiting for the testing process to open our status pipe for reading.

We'll need the file controls and standard symbolic constants and types.

```
<<include files>>=
#include <fcntl.h>
#include <unistd.h>
```

```
<<global variables>>=
int commandPipeFd ;
int statusPipeFd ;
```

```
<<set up for named pipes>>=
commandPipeFd = open( ElevatorCommandsPipe, O_RDONLY | O_NONBLOCK ) ;
if ( commandPipeFd == -1 ) {
   perror( "Unable to open the ElevatorCommandPipe" ) ;
   exit( 1 ) ;
}

cout << "Waiting for test program to start." << endl ;

statusPipeFd  = open( ElevatorStatusPipe, O_WRONLY ) ;
if ( statusPipeFd == -1 ) {
   perror( "Unable to open the ElevatorStatusPipe" ) ;
   exit( 1 ) ;
}
```

Note that we read the command pipe and write the status pipe.

Before receiving configuration input from the testing process, let's see what was requested on the command line. We require that both the number of floors and the number of elevators be present if either is specified on the command line. If only one is present, we'll just ignore it so the testing process will be responsible for both values.

If the user specified a floor count, but not the floor labels, we'll provide the floor labels, the integers from 1 to `numberOfFloors`.

```
<<constants>>=
const int MIN_NUMBER_OF_FLOORS = 2 ;
```

```
<<set up for named pipes>>=
if ( strlen( floorSpecification ) != 0 || numberOfFloors != 0
  || numberOfElevators != 0 ) {
   if ( numberOfElevators == 0 ) {
      numberOfFloors = 0 ;
```

```
      floorSpecification[ 0 ] = '\0' ;
   } else if ( numberOfFloors == 0 && strlen( floorSpecification ) == 0 ) {
      numberOfElevators = 0 ;
   } else {
      if ( strlen( floorSpecification ) == 0
        && numberOfFloors >= MIN_NUMBER_OF_FLOORS
        && numberOfFloors <= MAX_NUMBER_OF_FLOORS ) {
         for ( int i = 0; i < MAX_NUMBER_OF_FLOORS + 2; i++ ) {
            if ( i < numberOfFloors ) {
               floorSpecification[ i ] = '0' + i + 1 ;
            } else {
               floorSpecification[ i ] = '\0' ;
            }
         }
      }
      <<check configuration>>
      <<quit if input error>>
   }
}
```

```
<<global variables>>=
char floorLabels[ MAX_NUMBER_OF_FLOORS + 1 ] ;
int groundFloor ;
```

The floorLabels string is sized to allow for an ending nul. The groundFloor value is 0-based, counting up from the bottom floor.

Our checkInput function checks the numberOfElevators and floorSpecification input and returns the numberOfFloors, groundFloor, and floorLabels. The floorLabels are returned in the caller's array.

```
<<forward references>>=
bool checkInput( int numberOfElevators, const char *floorSpecification,
                 int &numberOfFloors, int &grndFloor, char *floorLabels ) ;
```

For debugging, VERBOSE_CONFIGURATION may be defined to display pre- and post-checkInput values of numberOfElevators, floorSpecification, numberOfFloors, groundFloor, and floorLabels.

```
<<check configuration>>=
#ifdef VERBOSE_CONFIGURATION
{
   char str[ 200 ] ; // ample
   sprintf( str, "Before checkInput> numberOfElevators: %i, "
            "floorSpecification: %s, numberOfFloors: %i, "
            "groundFloor: %i, floorLabels: %s",
            numberOfElevators, floorSpecification, numberOfFloors,
            groundFloor, floorLabels ) ;
   cout << str << endl ;
}
#endif // def VERBOSE_CONFIGURATION
bool inputIsOK = checkInput( numberOfElevators, floorSpecification,
                             numberOfFloors, groundFloor,
                             floorLabels ) ;
#ifdef VERBOSE_CONFIGURATION
{
   char str[ 200 ] ; // ample
   sprintf( str, "After checkInput> numberOfElevators: %i, "
            "floorSpecification: %s, numberOfFloors: %i, "
            "groundFloor: %i, floorLabels: %s",
            numberOfElevators, floorSpecification, numberOfFloors,
            groundFloor, floorLabels ) ;
   cout << str << endl ;
}
```

```
#endif // def VERBOSE_CONFIGURATION
```

After checking the command-line values, if there are any errors we'll send a P_END ending message (defined below) to the testing process, wait a moment for the pipes to close, and exit.

Note that an error message would have already been written by checkInput when it found an error.

```
<<constants>>=
const int SAFE_SIZE = 32 ;
const int waitBeforeClosingPipes = 2000 ; // ms
```

sleepMs is one of several coroutine functions and utilities from the Simple C/C++ Coroutines Library.

```
<<include files>>=
#include "sccorlib.h"
```

```
<<quit if input error>>=
if ( !inputIsOK ) {
   char str[ SAFE_SIZE ];
   sprintf( str, "%s\n", P_END ) ;
   write( statusPipeFd, str, strlen( str ) ) ;
   sleepMs( waitBeforeClosingPipes ) ;
   exit( 911 ) ;
}
```

If there are no errors so far, we're ready for the testing process to send us configuration input and requests.

While configuring, the testing process acts as the client and elevator acts as the server. That is, the testing process initiates by sending a request or set message and elevator sends a response message.

Messages between the testing process and elevator are short ASCII strings.

- Requests start with "?".
- Set commands start with "!".
- Each request or set message has a unique two-character identification code.
- Every configuration message from the testing process to elevator receives a response.
- For query requests, elevator responds with the requested data.
- For a command which sets a value, elevator responds with either a "bad" or "OK" indication.

Status Response Messages

There are three status responses that can be sent from elevator to the testing process:

```
<<pipe statuses>>=
const char P_BAD[] = "BAD" ;
const char P_OK[]  = "OK" ;
const char P_END[] = "-30-" ;
```

As one might imagine, P_BAD signals something is amiss, while P_OK says that all is hunky-dory.

Besides the case of input errors discussed above, P_END is sent by elevator in response to a request to stop the test.

Session End Messages

There are two messages that can be sent from the testing process to elevator to stop the test session.

- The quitTest message is sent from the testing process to signal that the allotted test time has expired and the session is ending.

```
<<pipe start and stop commands>>=
const char quitTest[] = "q" ;
```

  The response from elevator is P_END.

- The userRequestedQuitTest message is sent from the testing process to indicate that the testing process user wants to interrupt the test. The userRequestedQuitTest message will cause elevator to set an exit code that can be interpreted by a script running elevator to exit the script's execution loop.

```
<<pipe start and stop commands>>=
const char userRequestedQuitTest[] = "z" ;
```

  The response from elevator is P_END.

Test Configuration Messages

Now we'll look at the main code that processes the test configuration messages.

First, there are some pipe constants that are shared between elevator and a testing process via the pipe_interface.h header file, created as part of this literate program.

We use an ample buffer size for pipe messages between elevator and the testing process.

```
<<pipe constants>>=
const int MAX_BUF_SIZE = 255 ;
```

When reading commands from the testing process, if no data is available from the pipe, we wait briefly for pipeCheckInterval milliseconds and then read again.

```
<<constants>>=
const int pipeCheckInterval = 1 ;
```

While we're processing configuration messages from the testing process, we'll loop reading the command pipe and handling each request or set command received, replying to each message as we go.

```
<<set up for named pipes>>=
bool configuring = true ;
char buf[ MAX_BUF_SIZE ] ;
int  numRead ;
while ( configuring ) {
   numRead = read( commandPipeFd, buf, MAX_BUF_SIZE ) ;
   if ( numRead == -1 ) {
      if ( errno == EAGAIN ) {
         sleepMs( pipeCheckInterval ) ;
      } else {
         // An error has occurred reading the pipe.
         perror( "Error reading the ElevatorStatusPipe" ) ; // ❶
         exit( 3 ) ;
      }
   } else if ( numRead > 0 ) {
      buf[ numRead ] = '\0' ; // for safety
      #ifdef VERBOSE_CONFIGURATION
      {
         char str[ 80 ] ; // ample
         sprintf( str, "Received via pipe (%i chars)> buf: '%s'",
```

```
                    numRead, buf ) ;
            cout << str << endl ;
        }
        #endif // def VERBOSE_CONFIGURATION
        <<process configuration message>>
    }
}
```

❶ Probably should also try to send an alert message to the test process.

Defining VERBOSE_CONFIGURATION will enable debugging output containing the configuration messages received from the testing process over the pipe.

hello message

- A helloMessage is sent from the testing process to initiate a testing session and start configuration processing.

```
<<pipe start and stop commands>>=
const char helloMessage[] = "Ola!" ;
```

```
<<process configuration message>>=
if ( !strcmp( buf, helloMessage ) ) {
    char str[ SAFE_SIZE ] ;
    sprintf( str, "%s%s\n", P_OK, ourNameAndVersion ) ;
    write( statusPipeFd, str, strlen( str ) ) ;
    #ifdef VERBOSE_CONFIGURATION
    {
        char str[ 80 ] ; // ample
        sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
                 strlen( P_OK ), P_OK ) ;
        cout << str << endl ;
    }
    #endif // def VERBOSE_CONFIGURATION
}
```

The response from elevator to the testing process is P_OK concatenated with our name and version. For example, we would write the string "OKrelease/elevator 1.0" should our executable be named "elevator", executing from the "release" directory, and having a version of "1.0".

end configuration message

- Barring errors, configuration processing will conclude with elevator receiving an endConfiguration message, which will stop the configuration processing loop by setting the local configuring variable to false.

```
<<pipe start and stop commands>>=
const char endConfiguration[] = "!EC" ;
```

```
<<process configuration message>>=
else if ( !strcmp( buf, endConfiguration ) ) {
    configuring = false ;
}
```

query max dimensions message

- The queryMaxDimensions message is sent from the testing process to get the maximum dimensions allowed by the Terminal screen dimensions elevator is running in.

```
<<pipe configuration commands>>=
const char queryMaxDimensions[] = "?MD" ;
```

The MAX_DOOR_HEIGHT constant is the elevator door height on the screen. The door height is 7 characters.

THe MAX_FLOOR_LABEL_WIDTH constant is the number of characters in a floor label string, *e.g.*, "[3] ". The maximum floor label width is 4.

The MAX_DOOR_WIDTH constant is the elevator door width on the screen. The door width is 11 characters.

elevator uses these values to determine the maximum possible number of elevator shafts and floors in the building.

```
<<constants>>=
const int MAX_FLOOR_LABEL_WIDTH = 4 ;
const int MAX_DOOR_WIDTH = 11 ;
```

The global variable w is our ncurses window.

The global variables nrows and ncols hold the number of rows and columns, respectively, in the window.

Since we're using ncurses, we'll need its prototypes.

```
<<include files>>=
#include <curses.h>
```

```
<<global variables>>=
WINDOW *w ;
int nrows ;
int ncols ;
```

```
<<process configuration message>>=
else if ( !strcmp( buf, queryMaxDimensions ) ) {
   int  maxNoOfFloors ;
   int  maxNoOfElevators ;
   char isFixed ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   if ( numberOfFloors != 0 && numberOfElevators != 0 ) {
      isFixed          = groundFloor + '0' ;
      maxNoOfFloors    = numberOfFloors + '0' ;
      maxNoOfElevators = numberOfElevators + '0' ;
   } else {
      w = initscr() ;
      getmaxyx( w, nrows, ncols ) ;
      endwin() ;
      isFixed = '*' ;
      maxNoOfFloors    = ( nrows - 2 ) / MAX_DOOR_HEIGHT + '0' ;
      maxNoOfElevators = ( ncols - MAX_FLOOR_LABEL_WIDTH - 2 )
                                          / MAX_DOOR_WIDTH + '0' ;
   }
   sprintf( buf, "%s%c%c%c\n", commandSignature,
            maxNoOfFloors, maxNoOfElevators, isFixed ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
```

```
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

If the numberOfFloors and numberOfElevators are positive, then the dimensions are fixed as the user specified on command line.

Otherwise, we briefly check with curses for console dimensions and tell the testing process the maximum dimensions for the current console screen size.

The response from elevator to the testing process is "MD"yxg, where *y* is the ASCII numeral for the number of floors, *x* is the ASCII numeral for number of elevator shafts, and *g* is either the ASCII numeral for the ground floor number or "*", meaning not determined yet.

query floor labels message

- The queryFloorLabels message is sent from the testing process to get any floor labels that have already been specified by elevator's user on the command line.

```
<<pipe configuration commands>>=
const char queryFloorLabels[] = "?FL" ;
```

```
<<process configuration message>>=
else if ( !strcmp( buf, queryFloorLabels ) ) {
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s%s\n", commandSignature, floorLabels ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The response from elevator is "FLabc..", where *abc..* are the ASCII character floor labels assigned by the user (or elevator, if not specified on the command line).

set block clear time message

- The setBlockClearTime message is sent from the testing process to set the amount of time (in milliseconds) to wait after a door is blocked before attempting to close it again.

```
<<pipe configuration commands>>=
const char setBlockClearTime[] = "!BC" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setBlockClearTime,
                    strlen( setBlockClearTime ) ) ) {
   blockClearTime = atoi( buf + strlen( setBlockClearTime ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
```

```
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The testing process appends a block clear time to the message as ASCII numerals, *e.g.*, "!BC13000" would set the block clear time to 13,000 milliseconds (13 seconds).

We save the block clear time (in milliseconds) as `blockClearTime`.

```
<<global variables>>=
int blockClearTime ;
```

The response from `elevator` to the testing process is either "BC#OK", if the value is acceptable, or "BC#BAD", if unacceptable.

set door open close time message

- The `setDoorOpenCloseTime` message is sent from the testing process to set the open / close time, the time required for the elevator car door to open or close.

```
<<pipe configuration commands>>=
const char setDoorOpenCloseTime[] = "!DO" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setDoorOpenCloseTime,
                    strlen( setDoorOpenCloseTime ) ) ) {
   doorOpenCloseTime = atoi( buf + strlen( setDoorOpenCloseTime ) ) ;
   doorTransitTime   = doorOpenCloseTime / 2 ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The door open / close time (in milliseconds) is appended to the message as ASCII numerals, *e.g.*, "!DO4500" would set the door open / close time to 4,500 ms, or 4.5 seconds.

We save the door open / close time (in milliseconds) as `doorOpenCloseTime` and a `doorTransitTime` (half the `doorOpenCloseTime`, used in displaying door openness state, also in milliseconds) as global variables.

```
<<global variables>>=
int doorOpenCloseTime ;
```

The response from `elevator` to the testing process is either "DO#OK", if the value is acceptable, or "DO#BAD", if unacceptable.

set floor height message

- The setFloorHeight message is sent from the testing process to set the height (in meters) of the floors in the building.

```
<<pipe configuration commands>>=
const char setFloorHeight[] = "!FH" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setFloorHeight,
                    strlen( setFloorHeight ) ) ) {
   floorHeight = atof( buf + strlen( setFloorHeight ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
      sprintf( str, "floorHeight (received): '%f'", floorHeight ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The floor height (which is the distance between floors, in meters, and applies to all floors in the building) is appended to the message as ASCII numerals (and possibly a decimal point), *e.g.*, "!FH3.048" would set the floor height to 3.048 meters (which is about 10 feet).

We save the floor height (in meters) as floorHeight.

```
<<global variables>>=
double floorHeight ;
```

The response from elevator to the testing process is either "FH#OK", if the value is acceptable, or "FH#BAD", if unacceptable.

set floor labels message

- The setFloorLabels message is sent from the testing process to set the building's floor labels for the simulation in the case that elevator's user did not specify them on the command line.

```
<<pipe configuration commands>>=
const char setFloorLabels[] = "!FL" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setFloorLabels, strlen( setFloorLabels ) ) ) {
   strcpy( floorLabels, buf + strlen( setFloorLabels ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
```

```
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The floor labels (one label for each floor) are appended to the message as an ASCII string, *e.g.*, "!FLBL2345" would set the floor labels for a 6-flor building to "B" for the basement, "L" for the lobby, and "2", "3", "4", and "5" for the remaining floors.

The response from elevator to the testing process is either "FL#OK", if the value is acceptable, or "FL#BAD", if unacceptable.

set ground floor level message

- The setGroundFloorLevel message is sent from the testing process to establish the building's ground floor level for the simulation in the case that elevator's user did not specify it on the command line.

```
<<pipe configuration commands>>=
const char setGroundFloorLevel[] = "!GF" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setGroundFloorLevel,
                    strlen( setGroundFloorLevel ) ) ) {
   groundFloor = atoi( buf + strlen( setGroundFloorLevel ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The ground floor level (0-based, from the bottom floor upwards) is appended to the message as an ASCII numeral, *e.g.*, "!GF1" would set the ground floor to the floor above the basement (the second floor from the bottom of the building).

We save the ground floor level as the groundFloor integer, a 0-based level from the bottom.

The response from elevator to the testing process is either "GF#OK", if the value is acceptable, or "GF#BAD", if unacceptable.

set max cabin velocity message

- The setMaxCabinVelocity message is sent from the testing process to set the maximum velocity (in meters per second) of an elevator car.

```
<<pipe configuration commands>>=
const char setMaxCabinVelocity[] = "!CV" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setMaxCabinVelocity,
                    strlen( setMaxCabinVelocity ) ) ) {
   maxCabinVelocity = atof( buf + strlen( setMaxCabinVelocity ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
```

```
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf ( str, "Written via pipe (%li chars)> buf: '%s'",
                strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The maximum cabin velocity value (in meters/second) is appended to the message as ASCII numerals (and possibly a decimal point), *e.g.*, "!CV0.95" would set the maximum car velocity to 0.95 m/s.

We save the maximum cabin velocity (in m/s) as `maxCabinVelocity`.

```
<<global variables>>=
double maxCabinVelocity ;
```

The response from `elevator` to the testing process is either "CV#OK", if the value is acceptable, or "CV#BAD", if unacceptable.

set max close attempts message

- The `setMaxCloseAttempts` message is sent from the testing process to set the maximum number of successive attempts to close an obstructed elevator door.

  If the door cannot be successfully closed within the specified number of attempts, the elevator will be placed in an "out of service" status.[17]

```
<<pipe configuration commands>>=
const char setMaxCloseAttempts[] = "!CA" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setMaxCloseAttempts,
                    strlen( setMaxCloseAttempts ) ) ) {
   maxCloseAttempts = atoi( buf + strlen( setMaxCloseAttempts ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The maximum close attempts value is appended to the message as ASCII numerals , *e.g.*, "!CA12" would set the maximum close attempts to 12.

We save the maximum close attempts value as `maxCloseAttempts`.

```
<<global variables>>=
int maxCloseAttempts ;
```

The response from `elevator` to the testing process is either "CA#OK", if the value is acceptable, or "CA#BAD", if unacceptable.

[17] "Out of service" is not currently implemented.

set minimum stopping distance message

- The setMinStoppingDistance message is sent from the testing process to establish the minimum distance from a destination floor (in meters) required for an elevator car to come to a complete stop.

```
<<pipe configuration commands>>=
const char setMinStoppingDistance[] = "!MS" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setMinStoppingDistance,
                    strlen( setMinStoppingDistance ) ) ) {
   minStoppingDistance = atof( buf + strlen( setMinStoppingDistance ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The minimum stopping distance is appended to the message as ASCII numerals (and possibly a decimal point), *e.g.*, "!MS1.50" would set the minimum stopping distance to 1.50 meters.

We save the minimum stopping distance (in meters) as setMinStoppingDistance.

```
<<global variables>>=
double minStoppingDistance ;
```

The response from elevator to the testing process is either "MS#OK", if the value is acceptable, or "MS#BAD", if unacceptable.

set normal door wait time message

- The setNormalDoorWaitTime message is sent from the testing process to set the normal door wait time, the time to allow for passengers to exit and enter the elevator car, when the door opens.

```
<<pipe configuration commands>>=
const char setNormalDoorWaitTime[] = "!DW" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setNormalDoorWaitTime,
                    strlen( setNormalDoorWaitTime ) ) ) {
   normalDoorWaitTime = atoi( buf + strlen( setNormalDoorWaitTime ) ) ;
   doorRemainsOpenTime = normalDoorWaitTime ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
```

```
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The normal door wait time (in milliseconds) for the simulation is appended to the message as ASCII numerals, *e.g.*, "!DW11000" would set the normal door wait time to 11,000 ms, or 11 seconds.

We save the normal door wait time (in milliseconds) as `normalDoorWaitTime` and as the global `doorRemainsOpenTime` (also in milliseconds).

```
<<global variables>>=
int normalDoorWaitTime ;
```

The response from `elevator` to the testing process is either "DW#OK", if the value is acceptable, or "DW#BAD", if unacceptable.

set number of elevators message

- The `setNumberOfElevators` message is sent from the testing process to set the number of elevator shafts in the case where `elevator`'s user did not specify it on the command line.

```
<<pipe configuration commands>>=
const char setNumberOfElevators[] = "!NE" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setNumberOfElevators,
                    strlen( setNumberOfElevators ) ) ) {
   numberOfElevators = atoi( buf + strlen( setNumberOfElevators ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The number of elevators for the simulation is appended to the message as an ASCII numeral, *e.g.*, "!NE6" would set the number of elevators to 6.

We save the number of elevators as `numberOfElevators`.

The response from `elevator` to the testing process is either "NE#OK", if the value is acceptable, or "NE#BAD", if unacceptable.

set number of floors message

- The `setNumberOfFloors` message is sent from the testing process to set the number of floors in the case where `elevator`'s user did not specify it on the command line.

```
<<pipe configuration commands>>=
const char setNumberOfFloors[] = "!NF" ;
```

```
<<process configuration message>>=
else if ( !strncmp( buf, setNumberOfFloors,
                    strlen( setNumberOfFloors ) ) ) {
   numberOfFloors = atoi( buf + strlen( setNumberOfFloors ) ) ;
   char commandSignature[ 3 ] ; // ample for "cc"
   strncpy( commandSignature, buf + 1, 2 ) ;
   commandSignature[ 2 ] = '\0' ;
   sprintf( buf, "%s#%s\n", commandSignature, P_OK ) ;
   write( statusPipeFd, buf, strlen( buf ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
               strlen( buf ), buf ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

The number of floors for the simulation is appended to the message as an ASCII numeral, *e.g.*, "!NF4" would set the number of floors to 4.

We save the number of floors as `numberOfFloors`.

The response from `elevator` to the testing process is either "NF#OK", if the value is acceptable, or "NF#BAD", if unacceptable.

unknown message

If the incoming message from the testing process is not recognized, we'll write a `P_BAD` response and otherwise ignore the message we received.

```
<<process configuration message>>=
else {
   char str [SAFE_SIZE ] ;
   sprintf( str, "%s\n", P_BAD ) ;
   write( statusPipeFd, str, strlen( str ) ) ;
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "Written via pipe (%li chars)> buf: '%s'", strlen( P_BAD ), P_BAD ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
}
```

Now that we have received the configuration from the testing process, we'll update `positionTransitionTime`, which will be used in displaying elevator car movement. It is the time required (in milliseconds) to move vertically from one row ("position") on the screen to an adjacent row.

```
<<set up for named pipes>>=
positionTransitionTime = 1000 * floorHeight / ( MAX_DOOR_HEIGHT * maxCabinVelocity ) ;
```

We'll assume a successful result from checking our inputs, and `checkInput` will give us the actual results.

```
<<set up for named pipes>>=
bool configurationIsOK = true ;
char *configResult = ( char * )P_OK ;
```

We should have received a floor labels string from the testing process. If not, we'll indicate that the configuration is bad.

Also, we make sure we have a floorSpecification string before doing checkInput.

Finally, we'll check that the configuration is OK.

```
<<set up for named pipes>>=
if ( !strlen( floorLabels ) ) {
   configurationIsOK = false ;
   configResult = ( char * )P_BAD ;
} else {
   if ( !strlen( floorSpecification ) ) {
      char *pFs = floorSpecification ;
      for ( int i = 0; i < numberOfFloors; i++ ) {
         if ( i == groundFloor ) {
            *pFs++ = '#' ;
         } else {
            *pFs++ = floorLabels[ i ] ;
         }
      }
      *pFs++ = '\0';
   }
   #ifdef VERBOSE_CONFIGURATION
   {
      char str[ 80 ] ; // ample
      sprintf( str, "floorLabels: '%s', floorSpecification: '%s'",
               floorLabels, floorSpecification ) ;
      cout << str << endl ;
   }
   #endif // def VERBOSE_CONFIGURATION
   configurationIsOK = checkInput( numberOfElevators, floorSpecification,
                                   numberOfFloors, groundFloor,
                                   floorLabels ) ;

   if ( !configurationIsOK ) {
      configResult = ( char * )P_BAD ;
   }
}
```

Now we'll tell the testing process how the configuration fares.

If the configuration is bad, we'll exit, with checkInput having already written an error message.

```
<<set up for named pipes>>=
char commandSignature[ 3 ] ; // ample for "cc"
strncpy( commandSignature, endConfiguration + 1, 2 ) ;
commandSignature[ 2 ] = '\0' ;
sprintf( buf, "%s#%s\n", commandSignature, configResult ) ;
write( statusPipeFd, buf, strlen( buf ) ) ;
#ifdef VERBOSE_CONFIGURATION
{
char str[ 80 ] ; // ample
sprintf( str, "Written via pipe (%li chars)> buf: '%s'",
         strlen( buf ), buf ) ;
cout << str << endl ;
}
#endif // def VERBOSE_CONFIGURATION
if ( !configurationIsOK ) {
   exit( 911 ) ;
}
```

A Sample Configuration Sequence Diagram Between Testing Process and elevator

Here is a sequence diagram showing a typical set of configuration exchanges between the testing process and elevator.

Sequence diagram for typical configuration messages and replies

```
@startuml
skinparam sequenceMessageAlign direction

== Hello Message ==

testing_process -> elevator: Ola!
elevator --> testing_process: OKdebug/elevator 1.0

== Query Maximum Dimensions ==

testing_process -> elevator: ?MD
testing_process <-- elevator: MD671

== Query Floor Labels ==

testing_process -> elevator: ?FL
testing_process <-- elevator: FLBL2345

== Set Floor Height ==

testing_process -> elevator: !FH3.048000
testing_process <-- elevator: FH#OK

== Set Door Open / Close Time ==

testing_process -> elevator: !DO4000
testing_process <-- elevator: DO#OK

== Set Maximum Car Velocity ==

testing_process -> elevator: !CV0.900000
testing_process <-- elevator: CV#OK

== Set Maximum Door Close Attempts ==

testing_process -> elevator: !CA11
testing_process <-- elevator: CA#OK

== Set Normal Door Wait Time ==

testing_process -> elevator: !DW12000
testing_process <-- elevator: DW#OK

== Set Minimum Stopping Distance ==

testing_process -> elevator: !MS1.000000
testing_process <-- elevator: MS#OK

== Set Block Clear Time ==

testing_process -> elevator: !BC7000
testing_process <-- elevator: BC#OK

== End Configuration ==

testing_process -> elevator: ?EC
testing_process <-- elevator: EC#OK
@enduml
```

Input Verification without a Testing Process

When we're not interacting with a testing process, we just check that the user supplied valid values for the minimum input we require for the elevator simulation; namely, the number of elevator shafts and the number of floors in the building.

If the user specifies a floor count, but not the floor labels, we'll provide default floor labels, the integers 1..numberOfFloors.

If checkInput finds an error, it writes an error message and returns to us a false value so we will just exit.

```
<<check validity of input without named pipes>>=
if ( argc < 5 ) {
   cout << "Error: elevator requires two parameters, '-floors' and '-n'."
        << "\r\n"
        << "See help option (-?) for details." << "\r\n" ;
   exit( 911 ) ;
}
if ( strlen( floorSpecification ) == 0
  && numberOfFloors >= MIN_NUMBER_OF_FLOORS
  && numberOfFloors <= MAX_NUMBER_OF_FLOORS ) {
   for ( int i = 0; i < MAX_NUMBER_OF_FLOORS + 2; i++ ) {
      if ( i < numberOfFloors ) {
         floorSpecification[ i ] = '0' + i + 1 ;
      } else {
         floorSpecification[ i ] = '\0' ; // nul characters
      }
   }
}
bool inputIsOK = checkInput( numberOfElevators, floorSpecification,
                             numberOfFloors, groundFloor, floorLabels ) ;
if ( !inputIsOK ) {
   exit( 911 ) ;
}
```

Waiting and Showing Configuration if Requested

Now main waits, if the user requested it with the "-w" command line option. The configuration of elevators and floors is displayed and then main waits for the user to enter a keystroke before proceeding. This is intended to be used for debugging purposes.

main checks to see if the user has pressed a key with the kbhit function from ncurses.

```
<<forward references>>=
int kbhit( void ) ;
```

```
<<set up>>
if ( waiting ) {
   char str[ 80 ] ; // ample
   sprintf( str, "Simulation has %i elevators. Building has %i floors, "
            "labeled:\r", numberOfElevators, numberOfFloors ) ;
   cout << str << endl << "   " ;
   const char *pFl = floorLabels ;
   for ( int i = 0; i < numberOfFloors; i++ ) {
      if ( i > 0 ) {
         cout << ", " ;
      }
      cout << "'" << *pFl++ << "'" ;
   }
   cout << "\r" << endl ;
   cout << "\r\n" << "Press the Enter key to continue...\r\n" ;
   while ( kbhit() == false ) {
      SleepMs( 25 ) ; // ms
```

```
    }
    getchar() ; // clean the Enter key from the input buffer
}
```

main continues its setup by assuming an eventual normal exit and indicating that an orderly stop has not been requested.

exitCode and orderlyStop are global variables shared between coroutines. We make orderlyStop volatile so the compiler doesn't optimize any reference to it, since its value may be changed at any time by a different coroutine.

```
<<global variables>>=
int exitCode ;
volatile bool orderlyStop ;
```

```
<<constants>>=
const int NORMAL_EXECUTION_EXIT = 0 ;
```

```
<<set up>>=
exitCode = NORMAL_EXECUTION_EXIT ;
orderlyStop = false ;
```

We initialize our random stream to a repeatable value.

```
<<set up>>=
srandom( 1 ) ;
```

Then we set up ncurses.

```
<<set up>>=
w = initscr() ;
start_color() ; noecho() ; nonl() ;
use_default_colors() ;
init_pair( 1, (short)(COLOR_BLUE    | A_BOLD), -1 ) ;
init_pair( 2, (short)(COLOR_RED     | A_BOLD), -1 ) ;
init_pair( 3, (short)(COLOR_GREEN   | A_BOLD), -1 ) ;
init_pair( 4, (short)(COLOR_YELLOW  | A_BOLD), -1 ) ;
init_pair( 5, (short)(COLOR_BLACK           ), COLOR_CYAN ) ;
init_pair( 6, (short)(COLOR_BLACK           ), COLOR_WHITE ) ;
init_pair( 7, (short)(COLOR_BLACK           ), (short)(COLOR_YELLOW  | A_BOLD) ) ;
#define COLOR_COUNT 7
```

main processes console input in raw mode while running the elevator coroutines.

```
<<set up>>=
cbreak() ;
```

Next it finds the size of the window to enforce that the simulated building fits within the window.

```
<<set up>>=
getmaxyx( w, nrows, ncols ) ;
```

Finally, main's last setup step is to clear the window and hide the cursor.

```
<<set up>>=
int old_visibility ;
old_visibility = curs_set( 0 ) ;
clear() ;
refresh() ;
```

main's Elevator Simulation

Now main starts the initial two coroutines, roundtripCounter and elevatorSimulation.

```
<<forward references>>=
void roundtripCounter( void ) ;
void elevatorSimulation( int numberOfElevators, int numberOfFloors,
                         char floorLabels[] ) ;
```

The cobegin statement blocks after starting the two coroutines, and remains blocked until after all coroutine instances have returned.

```
<<do simulation>>=
cobegin( 2,                                  // ❶
         roundtripCounter,   0,              // ❷
         elevatorSimulation, 3,              // ❸
                             numberOfElevators,
                             numberOfFloors,
                             floorLabels
) ;
```

❶ Initial coroutine count ("2").

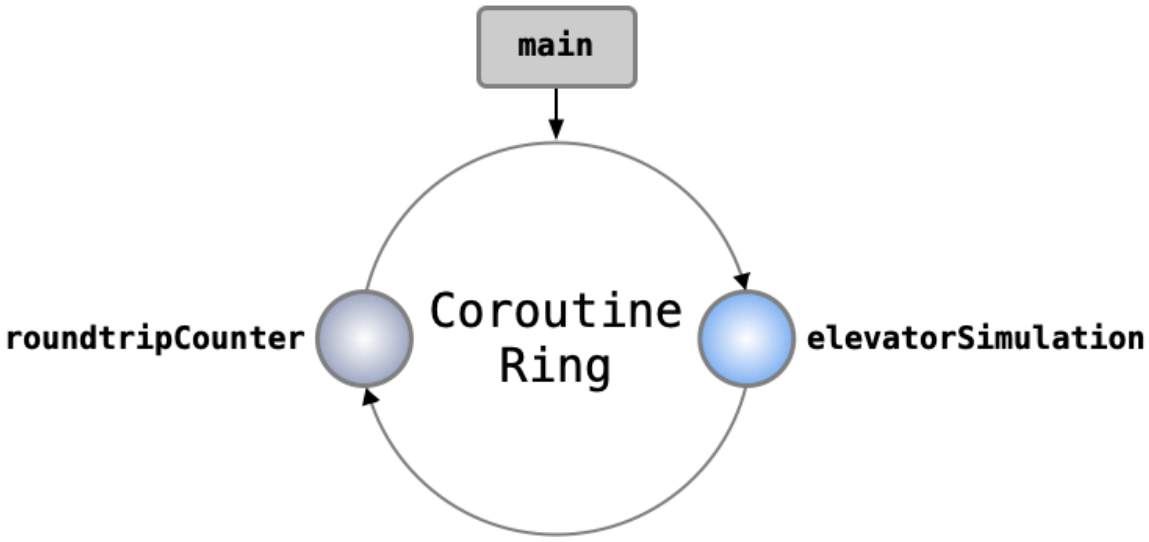
❷ No parameters ("0").

❸ 3 parameters ("3").

The roundtTripCounter coroutine takes no parameters. It provides a count of the number of coroutine cycles, namely, the number of times roundtripCounter executes. It can also collect data for a latency histogram showing the times elapsed between successive executions of roundtripCounter (*i.e.*,from coresume until execution returns to the next statement in roundtripCounter).

The elevatorSimulation coroutine is the controller of the elevator simulation. It takes three parameters: numberOfElevators, numberOfFloors, and floorLabels.

This is what the coresume ring looks like following the cobegin statement:



main's cobegin statement created a new ring and placed the roundtripCounter and elevatorSimulation coroutines on it, then blocked awaiting completion of all coroutines on the ring.

The coroutine ring runs clockwise, with execution passing to the next coroutine when the previous coroutine yields with a coresume call.

A coroutine remains on the ring until it returns (or falls through the end of its routine). So main will not resume its execution (with the statement following the cobegin statement) until all coroutines have finished.

elevatorSimulation draws the simulated building, instantiates a car coroutine for each of the numberOfElevators shafts, instantiates a pipeHandler coroutine if we're using pipes with a testing process, and then continuously reads and handles all notifications for the controller until an orderly stop is signaled.

The car coroutine simulates the elevator car behavior for the specified shaft. It receives notifications from the simulation controller (elevatorSimulation).

main's Take Down

After all coroutines have finished their executions, processing resumes in main at the statement following the cobegin statement.

The cursor is restored and console input is returned to cooked mode with echoing.

```
<<take down>>=
curs_set( old_visibility ) ;
refresh() ;
nocbreak() ; echo() ;
```

Knocking down ncurses.

```
<<take down>>=
endwin() ;
```

We show any alert message not already shown.

alertMessage is non-empty when it contains an alert message for the user.

```
<<take down>>=
if ( strlen( alertMessage ) > 0 ) {
   cout << "\r\n  *** ALERT: " << alertMessage << "\r" << endl ;
}
```

Before exiting, main shows the coroutine roundtrip count and average time per cycle.

```
<<global variables>>=
char roundtripCounterOutputString[ 80 ] ; // ample
```

```
<<take down>>=
cout << "\r\n" << roundtripCounterOutputString << "\r" << endl ;
```

Finally, main shows an optional coroutine roundtrip times histogram and returns.

For the histogram support we need the TimeIntervalHistogram class.

```
<<include files>>=
#include "histospt.h"
```

```
<<global variables>>=
#ifdef SHOW_HISTOGRAM
TimeIntervalHistogram itHistCR( "Coresume roundtrip times (us)",
                                0L, 250, 40 ) ;            // ❶
#endif // def SHOW_HISTOGRAM
```

❶ 40 buckets, each 250 microseconds wide, to display a roundtrip histogram of 0 - 10 milliseconds.

```
<<take down>>=
#ifdef SHOW_HISTOGRAM
itHistCR.show( false ) ;                                    // ❶
itHistCR.reset() ;                                          // ❷
#endif // def SHOW_HISTOGRAM

return( exitCode ) ;
```

❶ Omit log (“false”).

❷ Clear histogram for next time.

Notifications Between Coroutines

The elevator simulation employs Notifications[18] for communications between Components.

This notification (message) passing is accomplished using a Fifo class that is implemented in the fifo.h header file, an output of this literate program.

```
<<include files>>=
#include "fifo.h"
```

Notification Queue

We’ll utilize a Fifo queue, notification_Q, to hold notifications between coroutines.

```
<<global variables>>=
Fifo notification_Q ;
```

The Fifo implements a circular buffer with currentReadIndex pointing to the oldest data in the buffer and nextWriteIndex pointing to the first open slot in the buffer.

```
<<fifo private>>=
unsigned int currentReadIndex ;
unsigned int nextWriteIndex ;
```

No critical section is necessary in this implementation, since only the consumer manipulates the read pointer and only the producer manipulates the write pointer.[19] The queue is empty when the pointers are equal. We use only n - 1 slots so the write pointer can’t reach the read pointer "from the back", unless an overrun occurs.

We use a flag to indicate to the other Fifo methods that an overrun has occurred (when the write pointer reaches the read pointer "from the back")

```
<<fifo private>>=
bool overrunOccurred ;
```

The Fifo constructor just initializes the read and write indices as well as the overrun flag.

```
<<fifo public>>=
Fifo() {
   currentReadIndex = nextWriteIndex = 0 ;
   overrunOccurred = false ;
}
```

[18] To reduce confusion with other "messages" in this document (*e.g.*, error messages, pipe messages, *etc.*), we will use the term "notifications" rather than "messages" specifically for the communications sent between coroutines.

[19] Since the consumer and producer are in different coroutine instances, their executions are not concurrent and manipulations of their respective queue pointers are, consequently, completely independent.

Fifo has a default destructor.

```
<<fifo public>>=
~Fifo() {}
```

elevatorSimulation sends floor call button push events and in-car button push events as notifications to the appropriate car coroutines. These (simulated) button push events are initiated either by keyed user commands or piped testing process commands.

The car coroutines send status information to elevatorSimulation as they carry out their activities deriving from the push button events. The status information is used by elevatorSimulation to graphically present the elevator cars' motion and indicator lights' status and to make elevator car assignment decisions during the simulation.

Components Utilizing Notifications

The Components are implemented so that a car's shaft number is also its Component number. Note that broadcast is 0 and the controller (elevatorSimulation) is 10, both outside the span of possible car numbers.

```
<<notification constants and typedefs>>=
typedef enum Component {
   broadcast = 0,
   car1,
   car2,
   car3,
   car4,
   car5,
   car6,
   car7,
   car8,
   car9,
   controller
} Component ;
```

Notification Types

There are nine Notification types:

```
<<notification constants and typedefs>>=
typedef enum Notification {
   CAR_POSITION = 0,
   CAR_AVAILABLE,
   CAR_NOT_AVAILABLE,
   CAR_SERVICING_FLOOR,
   CAR_NOT_SERVICING_FLOOR,
   DOOR_POSITION,
   FLOOR_CALL,
   PUSH_CAR_BUTTON,
   SHOW_UP_DOWN_INDICATOR
} Notification ;
```

A Notification uses fixed slots for up to four associated data values, data0, data1, data2, and data3[20] (one character each). The number of data values actually used differs by notification type, as is shown in this table:

Notification Type	data0	data1	data2	data3
CAR_POSITION	position above (0-7)	floor label	direction	N/A
CAR_AVAILABLE	N/A	N/A	N/A	N/A

[20] data3 is currently not used in elevator.

Notification Type	data0	data1	data2	data3
CAR_NOT_AVAILABLE	N/A	N/A	N/A	N/A
CAR_SERVICING_FLOOR	N/A	N/A	N/A	N/A
CAR_NOT_SERVICING_FLOOR	N/A	N/A	N/A	N/A
DOOR_POSITION	door state	N/A	N/A	N/A
FLOOR_CALL	floor index	direction	N/A	N/A
PUSH_CAR_BUTTON	floor index	N/A	N/A	N/A
SHOW_UP_DOWN_INDICATOR	floor index	direction	N/A	N/A

The notifications are "transmitted" via a circular buffer, `buf`, containing `NUMBER_OF_FIFO_BUFFER_SLOTS` elements of type `notificationEnvelope`.

```
<<notification constants and typedefs>>=
const int NUMBER_OF_FIFO_BUFFER_SLOTS = 50 ;
```

```
<<notification constants and typedefs>>=
typedef struct {
   Notification   theNotification ;
   Component source ;
   Component destination ;
   char      data0 ;
   char      data1 ;
   char      data2 ;
   char      data3 ;
} notificationEnvelope ;
```

```
<<fifo private>>=
notificationEnvelope buf[ NUMBER_OF_FIFO_BUFFER_SLOTS ] ;
```

`PushNotification` Method

`Notifications` are sent with the `PushNotification` method. `PushNotification` takes a Notification, sender, destination, and exactly four data parameters.

```
<<fifo public>>=
int PushNotification( Notification theNotification, Component sender,
                 Component destination, char data0, char data1,
                 char data2, char data3 ) {
```

We use two constants to reflect possible outcomes of the `PushNotification` method.

```
<<notification constants and typedefs>>=
enum {
   FIFO_SUCCESS = 0,
   OVERRUN_OCCURRED
} ;
```

We assume success.

```
<<fifo public>>=
int rc = FIFO_SUCCESS ;
```

If there is not sufficient room in the ring buffer for this sample, we return `OVERRUN_OCCURRED` and set `overrunOccurred` to inform the other `Fifo` methods to prevent them from using data from the `Fifo` buffer.

```
<<fifo public>>=
if ( ( ( nextWriteIndex == NUMBER_OF_FIFO_BUFFER_SLOTS - 1 )
    && ( currentReadIndex == 0 ) )
```

```
  || ( nextWriteIndex + 1 == currentReadIndex ) ) {
   overrunOccurred = true ;
   rc = OVERRUN_OCCURRED ;
}
```

If room was available, we insert the notification and its envelope in our circular buffer.

```
<<fifo public>>=
else {
   buf[ nextWriteIndex ].theNotification  = theNotification ;
   buf[ nextWriteIndex ].source      = sender ;
   buf[ nextWriteIndex ].destination = destination ;
   buf[ nextWriteIndex ].data0       = data0 ;
   buf[ nextWriteIndex ].data1       = data1 ;
   buf[ nextWriteIndex ].data2       = data2 ;
   buf[ nextWriteIndex ].data3       = data3 ;
```

Next we bump the the write index, resetting it to the start of the buffer if we have reached the buffer's end.

```
<<fifo public>>=
   if ( ++nextWriteIndex == NUMBER_OF_FIFO_BUFFER_SLOTS ) {
      nextWriteIndex = 0 ;
   }
}
```

Finally, we conclude the `PushNotification` method by returning an indication of how things went.

```
<<fifo public>>=
   return rc ;
}
```

`PopNotification` Method

`Notifications` are received with the `PopNotification` method.

```
<<fifo public>>=
int PopNotification( Notification &returnedNotification, Component &returnedSender,
                Component &returnedDestination, char &data0,
                char &data1, char &data2, char &data3 ) {
```

We assume success.

```
<<fifo public>>=
int rc = FIFO_SUCCESS ;
```

If there has been a previous overrun, we just return that indication.

```
<<fifo public>>=
if ( overrunOccurred ) {
   rc = OVERRUN_OCCURRED ;
}
```

If no overrun has occurred, we grab the notification and envelope.

```
<<fifo public>>=
else {
   returnedNotification = buf[ currentReadIndex ].theNotification ;
   returnedSender       = buf[ currentReadIndex ].source ;
   returnedDestination  = buf[ currentReadIndex ].destination ;
   data0                = buf[ currentReadIndex ].data0 ;
```

```
   data1                = buf[ currentReadIndex ].data1 ;
   data2                = buf[ currentReadIndex ].data2 ;
   data3                = buf[ currentReadIndex ].data3 ;
```

Now we remove the notification from the buffer, resetting the read index if we have reached the end of the buffer[21].

```
<<fifo public>>=
   if ( currentReadIndex == NUMBER_OF_FIFO_BUFFER_SLOTS - 1 ) {
      currentReadIndex = 0 ;
   } else {
      currentReadIndex++ ;
   }
}
```

Finally, we conclude the PopNotification method by returning an indication of how things went.

```
<<fifo public>>=
   return rc ;
}
```

IsEmpty Method

IsEmpty is used to see if there are any Notifications available. The buffer is empty if the write and read indices are the same.

```
<<fifo public>>=
bool IsEmpty( void ) {
   return overrunOccurred ? false : nextWriteIndex == currentReadIndex ;
}
```

IsFor Method

IsFor is used to check a Notification's destination.

```
<<fifo public>>=
Component IsFor( void ) {
   return buf[ currentReadIndex ].destination ;
}
```

elevator's Coroutines

elevatorSimulation

elevatorSimulation is the controller for the elevator simulation.

The routine initially draws the external elevator doors and labels.

Subsequently, it provides a graphic presentation of the elevator simulation, including each car's position, the status of floor and elevator buttons, and the state of elevator doors (opened, partially-opened, or closed)[22].

The elevatorSimulation coroutine starts by checking the elevator executable's Terminal window dimensions (obtained earlier from ncurses) to see if the user's building size (as determined by the number of floors and number of elevator shafts the user specified on the command line) fits in the window. If either the number of elevator shafts or the number of floors is too large,

[21] The last slot in the buffer is NUMBER_OF_FIFO_BUFFER_SLOTS - 1, since the buffer is 0-indexed

[22] An actual elevator has a pair of fixed external doors and also inside doors which move with the car. Often the external and car doors are mechanically connected so that they open and close together when the car stops at a floor. In our simulation, the external doors are shown as fixed rectangles. They are unshaded when closed. The car doors are shown as shaded areas inside the rectangles in two positions, half open and fully open.

the user is given an error message (with a recommendation for a minimum appropriate size of window) and orderlyStop is set to true so that the elevatorSimulation coroutine will exit. In this case, the local variable shouldPause is also set to true so elevatorSimulation will wait for the user to view the error message before elevatorSimulation returns.

```
<<elevator coroutines>>=
void elevatorSimulation( int numberOfElevators, int numberOfFloors,
                         char floorLabels[] ) {
   <<check window size>>
   <<define local variables>>
   <<draw the building>>
   if ( !orderlyStop ) {
      <<instantiate the elevators>>
      <<start pipeHandler>>
   }
   while ( !orderlyStop ) {
      <<handle queued notification for controller>>
      coresume() ;
      <<check for commands>>
      coresume() ;
   }
   <<pause for user to read any error message>>
}
```

We check the window size and write an error message to the user if the size is not adequate, as well as indicating we'll do an orderly stop and exit.

```
<<check window size>>=
bool shouldPause = false ;
int curRow = 2 ;
if ( numberOfFloors * MAX_DOOR_HEIGHT + 2 > nrows ) {
   char str[ 90 ]; // ample
   sprintf( str, "Error: %i floors (%s) require a window height "
            "of at least %i rows.", numberOfFloors, floorLabels,
           numberOfFloors * MAX_DOOR_HEIGHT + 2 ) ;
   mvwaddstr( w, curRow++, 3, str ) ;
   sprintf( str, "The simulation is currently running in a window with "
            "a height of %i rows.", nrows ) ;
   mvwaddstr( w, curRow++, 3, str ) ;
   wrefresh( w ) ;
   curRow++ ; // blank line between messages
   orderlyStop = true ;
   shouldPause = true ;
}
if ( numberOfElevators * MAX_DOOR_WIDTH + MAX_FLOOR_LABEL_WIDTH + 2 > ncols ) {
   char str[ 90 ] ; // ample
   sprintf( str, "Error: %i elevators require a window width of at least "
            "%i columns.", numberOfElevators,
            numberOfElevators * MAX_DOOR_WIDTH + MAX_FLOOR_LABEL_WIDTH + 2 ) ;
   mvwaddstr( w, curRow++, 3, str ) ;
   sprintf( str, "The simulation is currently running in a window with "
                 "a width of %i columns.", ncols ) ;
   mvwaddstr( w, curRow++, 3, str ) ;
   wrefresh( w ) ;
   orderlyStop = true ;
   shouldPause = true ;
}
```

With a sufficiently large window, of dimensions nrows by ncols, and unless we have an orderly stop caused by a previous error, we draw the building.

We start by labeling the floors, from the bottom of the window, in columns 0 - 2. The labels are of the form "[0]", "[1]", *etc.* floorLabelRows is an array containing the row numbers where the labels are written. It is static so it can be subsequently accessed by the drawCar routine.

```
<<draw the building>>=
static int floorLabelRows[ MAX_NUMBER_OF_FLOORS ] ;
int currentLabelRow = nrows - 5 ;
for ( int i = 0; i < numberOfFloors; i++ ) {
   floorLabelRows[ i ] = currentLabelRow ;
   currentLabelRow -= MAX_DOOR_HEIGHT ;
}
if ( !orderlyStop ) {
   char str[ 80 ] ; // ample
   for ( int i = 0; i < numberOfFloors; i++ ) {
      sprintf( str, "[%c]", floorLabels[ i ] ) ;
      mvwaddstr( w, floorLabelRows[ i ], 0, str ) ;
   }
```

We next label the elevator shafts, across the top of the window. The shaft labels are also of the form "[1]", "[2]", *etc*.

```
<<draw the building>>=
for ( int i = 0; i < numberOfElevators; i++ ) {
   sprintf( str, "[%c]", '0' + i + 1 );
   mvwaddstr( w, max( 0, floorLabelRows[ numberOfFloors - 1 ] - 5 ),
              MAX_FLOOR_LABEL_WIDTH + MAX_DOOR_WIDTH * i + 5, str ) ;
}
```

Then we draw the elevator doors. The `initDoor` routine draws the initial image of an external door on each floor for each shaft of the building.

```
<<forward references>>=
void initDoor( int y, int x ) ;
```

```
<<draw the building>>=
for ( int i = 0; i < numberOfElevators; i++ ) {
   for ( int j = 0; j < numberOfFloors; j++ ) {
      initDoor( floorLabelRows[ j ] - 2,
                MAX_FLOOR_LABEL_WIDTH + MAX_DOOR_WIDTH * i + 3 ) ;
   }
}
```

The ground level is represented by a row of "=" characters. We remember the row number in `groundFloorRow`, a global variable so it can be shared with other coroutines.

```
<<global variables>>=
int groundFloorRow ;
```

```
<<draw the building>>=
groundFloorRow = floorLabelRows[ groundFloor ] + 3 ;
mvwhline( w, groundFloorRow, 0, '=',
          MAX_FLOOR_LABEL_WIDTH + MAX_DOOR_WIDTH * numberOfElevators ) ;
```

The up / down buttons are drawn on the right hand side of the window.

```
<<define local variables>>=
int udButtonColumn ;
```

```
<<draw the building>>=
udButtonColumn = numberOfElevators * MAX_DOOR_WIDTH
               + MAX_FLOOR_LABEL_WIDTH + 1 ; // on the far right
for ( int i = 0; i < numberOfFloors; i++ ) {
   if ( i != numberOfFloors - 1 ) {
      mvwaddch( w, floorLabelRows[ i ] - 1, udButtonColumn,
                'U' | COLOR_PAIR( 6 ) ) ;  // black on light grey
```

```
    }
    if ( i != 0 ) {
        mvwaddch( w, floorLabelRows[ i ], udButtonColumn,
                  'D'  | COLOR_PAIR( 6 ) ) ; // black on light grey
    }
}
```

Finally, we show the command line and display the initial screen.

```
<<draw the building>>=
mvwaddstr( w, nrows - 1, MAX_FLOOR_LABEL_WIDTH + 3, "Command: " ) ;
wrefresh( w ) ;
}
```

Now that the building has been drawn, we proceed to instantiate the elevator car coroutines (one for each elevator shaft), and a `pipeHandler` coroutine if we are running with a testing process.

```
<<forward references>>=
void car( int shaft ) ;
void pipeHandler( void ) ;
```

But first we need some local variables.

Note that the maximum number of elevators is limited by having only a single digit in the in-car buttons display.

```
<<constants>>=
const int MAX_NUMBER_OF_ELEVATORS = 9 ;
```

The `availableCars` and `servicingFloorCars` variables are 1-indexed bit-wise arrays used to keep track of available cars and cars which are servicing floors, respectively, as their names imply.

```
<<define local variables>>=
int availableCars = 0 ;
int servicingFloorCars = 0 ;
```

The `carButtons` and `carPositionAbove` variables are arrays of integers, with one integer for each elevator car, from 1 to `MAX_NUMBER_OF_ELEVATORS`.

Each integer in the `carButtons` array, except the unused 0th element, contains the state of the corresponding car's in-car buttons. For instance, `carButtons[2]` is the integer holding the in-car button status for elevator car 2. The buttons are bit-mapped in the integer, with the low-order bit representing the lowest floor in the building and bit 8 representing the highest possible floor, floor 9. Bits 9 and above are unused. A 0-bit indicates that the button is "off" and a 1-bit indicates "on".

```
<<define local variables>>=
static int carButtons[ MAX_NUMBER_OF_ELEVATORS + 1 ] ;
```

`carPositionAbove` contains the position above its current floor for each elevator car. There are 7 "positions" for each floor. Position 0 is considered to be "at" a given floor. Positions 1 to 6 are "above" the current floor.

```
<<define local variables>>=
static int carPositionAbove[ MAX_NUMBER_OF_ELEVATORS + 1 ] ;
```

`carPositionLabel` holds the label of the current floor for all elevator cars.

```
<<define local variables>>=
static char carPositionLabel[ MAX_NUMBER_OF_ELEVATORS + 1 ] ;
```

`carButtons`, `carPositionAbove`, and `carPositionLabel` are static so that they can be accessed from the `drawCar` routine. Note that these arrays have `MAX_NUMBER_OF_ELEVATORS` + 1 slots, as we don't use slot 0.

We have several more local variables.

`commandPosition` gives the starting position for user commands, after the "Commands: " prompt.

```
<<define local variables>>=
const int commandPosition = MAX_FLOOR_LABEL_WIDTH + 12 ;
```

commandOffset is the current offset of the user's input in the command string. It is initialized to 0, *i.e.*, no input yet.

```
<<define local variables>>=
int commandOffset = 0 ;
```

commandEraseTimeMs holds the time when a command should be erased.

```
<<define local variables>>=
unsigned int commandEraseTimeMs = 0 ;
```

When displaying a message to the user, we'll wait a second before erasing the message.

```
<<constants>>=
const int DELAY_BEFORE_ERASING_COMMAND = 1000 ; // ms
```

```
<<define local variables>>=
Notification theNotification ;
Component    sender ;
Component    destination ;
char         data0 ;
char         data1 ;
char         data2 ;
char         data3 ;
openness     doorPosition[ MAX_NUMBER_OF_ELEVATORS + 1 ] ;     // 1-indexed
int          floorCallDown[ MAX_NUMBER_OF_ELEVATORS + 1 ] ;    // 1-indexed
int          floorCallUp[ MAX_NUMBER_OF_ELEVATORS + 1 ] ;      // 1-indexed
int          floorsCalledUp     = 0 ;
int          floorsCalledDown   = 0 ;
unsigned int partialTimeMultiplier ;
unsigned int previousTimeMultiplier ;
direction    theDirection ;
char         theFloor ;
bool         waitingForButtonLabel = false ;
bool         waitingForCarNumber   = false ;
bool         waitingForEnter       = false ;
bool         waitingForFirstDigit  = false ;
bool         waitingForSecondDigit = false ;
bool         waitingForDirection   = false ;
bool         waitingForFloorLabel  = false ;
bool         waitingToClearCommand = true ;
struct timeval tv ;
```

OK, that was a *lot* of definitions !

With the preliminaries in place, we can now proceed to instantiate the car coroutines (one for each elevator shaft), and a pipeHandler coroutine if we are running with a testing process.

elevatorSimulation instantiates numberOfElevators car coroutines, starting with car number 1.

```
<<instantiate the elevators>>=
for ( int i = 1; i <= numberOfElevators; i++ ) {
   invoke( (COROUTINE)car, 1, i ) ;
   coresume() ;
```

We initialize our copy of the car's buttons, indicating no buttons pressed, and floor calls, with no up or down floor calls for this car.

```
<<instantiate the elevators>>=
carButtons[ i ] = 0 ;
floorCallUp[ i ] = 0 ;
floorCallDown[ i ] = 0 ;
```

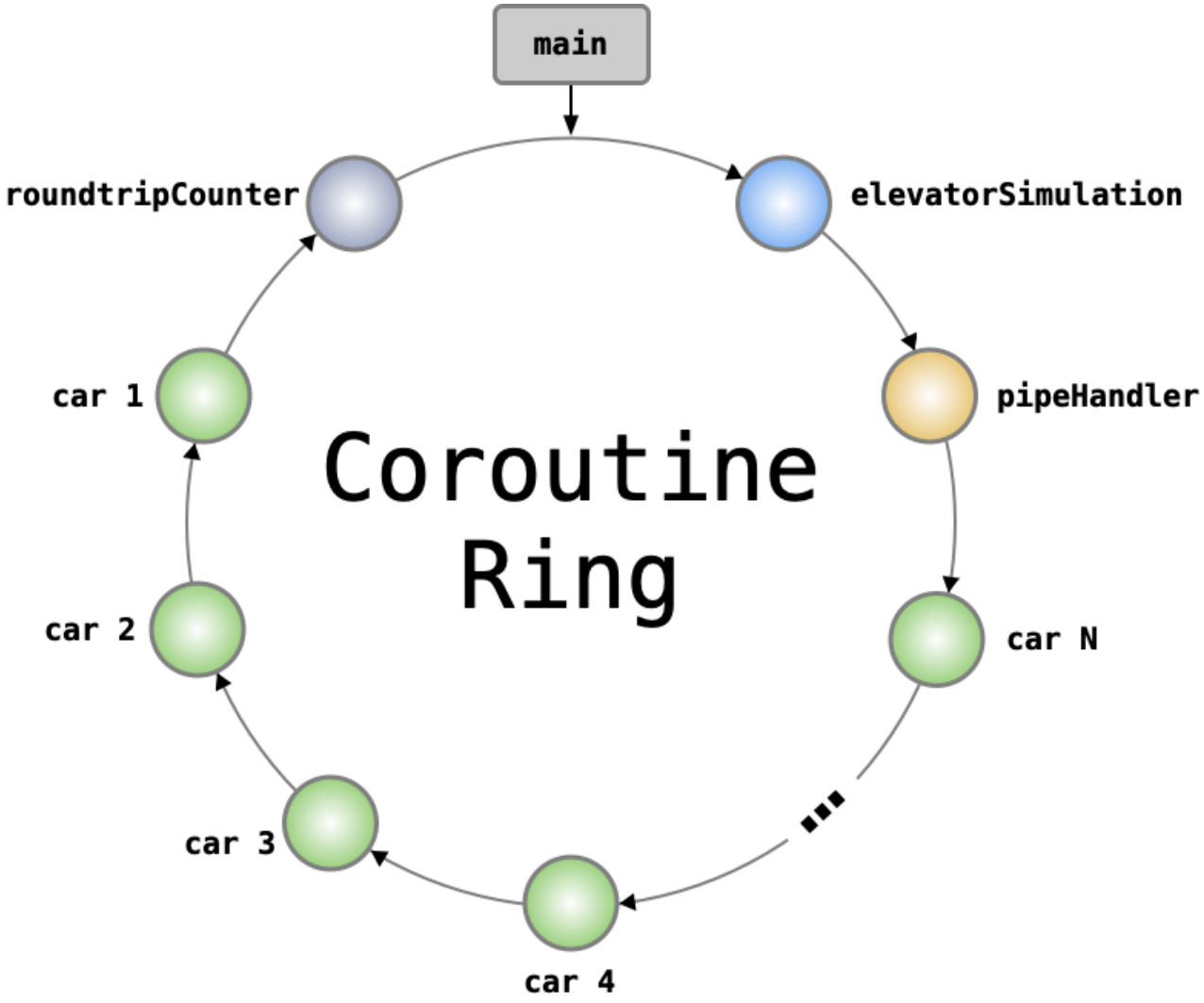
We also initialize our copy of the car's position, at ground level, so we can determine its direction and for our token-allocation purposes. Our door position is initialized to closed.

```
<<instantiate the elevators>>=
   carPositionAbove[ i ] = 0 ;
   carPositionLabel[ i ] = floorLabels[ groundFloor ] ;
   doorPosition[ i ] = doorClosed ;
}
```

If we are running with a testing process, `elevatorSimulation` also instantiates a (single) `pipeHandler` coroutine to handle messages between `elevator` and the testing process.

```
<<start pipeHandler>>=
if ( useNamedPipes ) {
   invoke( ( COROUTINE )pipeHandler, 0 ) ;
   coresume() ;
}
```

This is what the resulting coresume ring now looks like (and here we're assuming we have a `pipeHandler`):



The numberOfElevators instances of the car coroutine and the single instance of the pipeHandler coroutine have been added to the original ring that main created when it issued the cobegin statement for the elevatorSimulation and roundtripCounter coroutines.

The coroutine ring runs clockwise, with execution passing to the next coroutine when the previous coroutine yields with a coresume call.

A coroutine remains on the ring until it returns (or falls through the end of its routine). So main will not resume its execution (with the statement following the cobegin statement that started elevatorSimulation and roundtripCounter) until all coroutines have finished.

You may be wondering about the order of coroutines on the ring. When elevatorSimulation invokes a coroutine, the coroutine is added to the ring "in front" of elevatorSimulation. So car 1 was added between elevatorSimulation and roundtripHandler. The car 2 was added between elevatorSimulation and car 1. Eventually, the last (numberOfElevators'th) car coroutine was added to the ring between elevatorSimulation and the penultimate car's coroutine. And, if it is present, the pipeHandler coroutine is the last to be added and is right after elevatorSimulation.[23]

```
<<forward references>>=
void pushFloorButton( bool isOn, direction theDirection,
                      int udButtonColumn, int floorLabelRow ) ;
```

```
<<forward references>>=
void drawCar( int shaftNumber, int positionAbove, char theFloor,
              char floorLabels[], int floorLabelRows[], int buttonsPressed,
              direction theDirection, int groundFloorRow
              #ifdef SHOW_AVAILABILITY
              , bool available
              #endif // def SHOW_AVAILABILITY
            ) ;
```

```
<<forward references>>=
void drawOpenDoor( int y, int x, openness state ) ;
```

```
<<forward references>>=
void drawUpDownIndicator( int y, int x, direction indicatorDir ) ;
```

```
<<handle queued notification for controller>>=
// Check for any notifications for controller in the queue.
if ( !notification_Q.IsEmpty() && notification_Q.IsFor() == controller ) {
   notification_Q.PopNotification( theNotification, sender, destination,
                        data0, data1, data2, data3 ) ;
   switch ( theNotification ) {
      case CAR_AVAILABLE :
         availableCars |= ( 1 << sender ) ;
         break ;

      case CAR_NOT_AVAILABLE :
         availableCars &= ~( 1 << sender ) ;
         break ;

      case CAR_SERVICING_FLOOR :
         servicingFloorCars |= ( 1 << sender ) ;
         break ;

      case CAR_NOT_SERVICING_FLOOR :
         servicingFloorCars &= ~( 1 << sender ) ;
         break ;

      case CAR_POSITION :
```

[23] Not that it makes any difference, since we don't base any behavior on the position of coroutines on the ring.

```
        {
           carPositionAbove[ sender ] = data0 ;
           carPositionLabel[ sender ] = data1 ;
           direction carDir           = direction( data2 ) ;

           //#define SHOW_INTERMEDIATE_POSITIONS
           #ifdef SHOW_INTERMEDIATE_POSITIONS
           if ( useNamedPipes && carPositionAbove[ sender ] != 0 ) {
              // NB: just for a single car (e.g., 7).
              const int noteBufLen = sizeof carLocation + 2 ;
              static char noteBuf[ noteBufLen ] ; // includes '\n' + '\0'
              noteBuf[ 0 ] = 'a' ;                // note id
              noteBuf[ 1 ] = data1 ;              // floor
              noteBuf[ 2 ] = data0 + '0' ;        // position above
              noteBuf[ 3 ] = '\n' ;
              noteBuf[noteBufLen - 1] = '\0' ;
              write( statusPipeFd, noteBuf, strlen( noteBuf ) ) ;
           }
           #endif // def SHOW_INTERMEDIATE_POSITIONS

           // Check to see if the car has reached a target floor.
           if ( carPositionAbove[ sender ] == 0 ) {
              if ( useNamedPipes ) {
                 const int noteBufLen = sizeof carLocation + 2 ;
                 static char noteBuf[ noteBufLen ] ; // includes '\n' + '\0'
                 noteBuf[ 0 ] = carLocation[ 0 ] ;   // note id
                 noteBuf[ 1 ] = data1 ;              // floor
                 noteBuf[ 2 ] = sender + '0' ;       // car
                 noteBuf[ 3 ] = '\n' ;
                 noteBuf[noteBufLen - 1] = '\0' ;
                 write( statusPipeFd, noteBuf, strlen( noteBuf ) ) ;
                 #ifdef VERBOSE_CONFIGURATION
                 {
                    char str[ 80 ] ; // ample
                    sprintf( str, "Written via pipe (%li chars)> noteBuf: "
                                  "'%s'", strlen( noteBuf ), noteBuf ) ;
                    cout << str << endl ;
                 }
                 #endif // def VERBOSE_CONFIGURATION
              }

              char *floorLabel = strchr( floorLabels,
                                         carPositionLabel[ sender ] ) ;
              int floorIndex   = floorLabel - floorLabels ;
              int tempFloors ;
              if ( carDir == up ) {
                 tempFloors = floorCallUp[ sender ] ;
              } else { // carDir == down
                 tempFloors = floorCallDown[ sender ] ;
              }
              tempFloors |= carButtons[ sender ] ;
              if ( tempFloors & ( 1 << floorIndex ) ) {
                 // Turn off the floor button, just in case it is on. We'll
                 // turn off the 'down' button on the top floor or the 'up'
                 // button on the bottom floor.
                 int  floorLabelRow = floorLabelRows[ floorIndex ] ;
                 direction tempDir = carDir ;
                 if ( floorIndex == numberOfFloors - 1 ) {
                    tempDir = down ;
                 } else if ( floorIndex == 0 ) {
                    tempDir = up ;
                 }
```

```
                pushFloorButton( false, tempDir, udButtonColumn,
                                 floorLabelRow ) ;

                // Turn off our record of the car target and floor call.
                carButtons[sender] &= ~( 1 << floorIndex ) ;
                if ( tempDir == up ) {
                   floorCallUp[ sender ] &= ~( 1 << floorIndex ) ;
                } else { // tempDir == down
                   floorCallDown[ sender ] &= ~( 1 << floorIndex ) ;
                }
             }
          }
          drawCar( sender, carPositionAbove[ sender ],
                   carPositionLabel[ sender ],
                   floorLabels, floorLabelRows, carButtons[ sender ],
                   carDir, groundFloorRow
                   #ifdef SHOW_AVAILABILITY
                   , availableCars & ( 1 << sender )
                   #endif // def SHOW_AVAILABILITY
                ) ;
       }
       break ;

    case DOOR_POSITION :
       {
          doorPosition[ sender ] = openness( data0 ) ;
          char *pFloorLabel = strchr( floorLabels,
                                      carPositionLabel[ sender ] ) ;
          int floorIndex = pFloorLabel - floorLabels ;

          if ( useNamedPipes ) {
             char doorStatus ;
             switch ( openness( data0 ) ) {
                case doorLocked :
                   doorStatus = ' ' ;
                   break ;

                case doorClosed :
                   doorStatus = '|' ;
                   break ;

                case doorOpen :
                case doorOpenWaiting :
                   doorStatus = 'O' ;
                   break ;

                case doorHalfClosed :
                case doorHalfOpen :
                   doorStatus = '-' ;
                   break ;

                default :
                   doorStatus = '?' ;
                   // **??** TBD Put an alert here. **??**
             }
             const int noteBufLen = sizeof doorIsClosed + 2 ;
             static char noteBuf[ noteBufLen ] ; // includes '\n' + '\0'
             noteBuf[ 0 ] = doorIsClosed[ 0 ] ;  // note id
             noteBuf[ 1 ] = doorStatus ;         // door status
             noteBuf[ 2 ] = sender + '0' ;       // car
             noteBuf[ 3 ] = '\n' ;
             noteBuf[noteBufLen - 1] = '\0' ;
```

```
            write( statusPipeFd, noteBuf, strlen( noteBuf ) ) ;
            #ifdef VERBOSE_CONFIGURATION
            {
               char str[ 80 ] ; // ample
               sprintf( str, "Written via pipe (%li chars)> noteBuf: "
                             "'%s'", strlen( noteBuf ), noteBuf ) ;
               cout << str << endl ;
            }
            #endif // def VERBOSE_CONFIGURATION
         }
         if ( openness( data0 ) != doorLocked ) {
            drawOpenDoor( floorLabelRows[ floorIndex ] - 2,
                          MAX_FLOOR_LABEL_WIDTH
                          + MAX_DOOR_WIDTH * ( sender - 1 ) + 3,
                          openness( data0 ) ) ;
         }
      }
      break ;

   case SHOW_UP_DOWN_INDICATOR :
      {
         direction tempDir = direction( data1 ) ;
         if ( tempDir == none ) {
            // Accept the value 'none'.
         } else if ( data0 == numberOfFloors - 1 ) {
            tempDir = down ;
         } else if ( data0 == 0 ) {
            tempDir = up ;
         }
         if ( useNamedPipes ) {
            char dirIndicator ;
            switch ( tempDir ) {
               case none :
                  dirIndicator = ' ' ;
                  break ;

               case down :
                  dirIndicator = 'v' ;
                  break ;

               case up :
                  dirIndicator = '^' ;
                  break ;

               default:
                  dirIndicator = '?' ;
                  // **??** TBD Put an alert here. **??**
            }
            const int noteBufLen = sizeof indicatorIsDown + 2 ;
            static char noteBuf[ noteBufLen ] ;   // includes '\n', '\0'
            noteBuf[ 0 ] = indicatorIsDown[ 0 ] ; // note id
            noteBuf[ 1 ] = dirIndicator ;         // dir indicator
            noteBuf[ 2 ] = sender + '0' ;         // car
            noteBuf[ 3 ] = '\n' ;
            noteBuf[ noteBufLen - 1 ] = '\0' ;
            write( statusPipeFd, noteBuf, strlen( noteBuf ) ) ;
            #ifdef VERBOSE_CONFIGURATION
            {
               char str[ 80 ] ; // ample
               sprintf( str, "Written via pipe (%li chars)> noteBuf: "
                             "'%s'", strlen( noteBuf ), noteBuf ) ;
               cout << str << endl ;
```

```
               }
               #endif // def VERBOSE_CONFIGURATION
            }
            drawUpDownIndicator( floorLabelRows[ data0 ] - 2,
                                 MAX_FLOOR_LABEL_WIDTH
                                  + MAX_DOOR_WIDTH * ( sender - 1 ) + 3,
                                 tempDir ) ;
            if ( tempDir != none ) {
               beep() ;
            }
         }
         break ;

      default :
         break ; // **??** TBD Put an alert here. **??**
   }
}
```

```
<<forward references>>=
bool xkbhit( void ) ;
```

```
<<forward references>>=
int  xgetch( void ) ;          // simulated from pipe input
```

```
<<forward references>>=
void pushCarButton( bool isOn, int shaft, char theCarButton,
                    int positionAbove, int floorLabelRow ) ;
```

When running with a testing process, if the user requests ending the run, we'll exit with a special `USER_REQUESTED_EXIT` to distinguish the exit from a normal exit (0). This will stop the repeated execution of `elevator` by a surrounding script.

```
<<constants>>=
const int USER_REQUESTED_EXIT = 99 ;
```

```
<<check for commands>>=
// We give the user a chance to see the command before erasing it,
// and without delaying the rest of the elevator simulation.
if ( waitingToClearCommand ) {
   gettimeofday( &tv, (struct timezone *)NULL ) ;
   unsigned int currentTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;
   if ( currentTimeMs >= commandEraseTimeMs
      // Stop displaying command if another command arrives from the test
      // so as not to delay execution of the arriving command.
     || ( useNamedPipes && xkbhit() ) ) {
      commandOffset = 0 ;
      wmove( w, nrows - 1, commandPosition ) ;
      wclrtoeol( w ) ; // clear the rest of the command line
      wrefresh( w ) ;
      waitingToClearCommand = false ;
   }
  // Check for any input from the user.
} else if ( ( useNamedPipes && xkbhit() ) || ( !useNamedPipes && kbhit() ) ) {
   int c ;
   if ( useNamedPipes ) {
      c = xgetch( );   // simulated
   } else {
      c = wgetch( w ) ;
   }
   if ( c == 'q' || c == 'z' ) {
      if ( c == 'z' ) {
```

```
         // The user has requested to end test processing, so we'll
         // exit with a non-zero code.  This will stop the repeated
         // execution of the goes.rb script.
         exitCode = USER_REQUESTED_EXIT ;
      }

      mvwaddstr( w, nrows - 1, commandPosition, "quit" ) ;
      wclrtoeol( w ) ; // clear the rest of the command line
      wrefresh( w ) ;
      orderlyStop = true ;
      wait( 1000 ) ;   // give the user a chance to see the quit command
      if ( useNamedPipes ) {
         char str[ SAFE_SIZE ] ;
         sprintf( str, "%s\n", P_END ) ;
         write( statusPipeFd, str, strlen( str ) ) ;
      }
   } else if ( c == 0x7f ) { // ncurses backspace key (by experimentation)
      if ( waitingForButtonLabel ) {
         // Wipe out the command that was entered.
         commandOffset-- ;
         mvwdelch( w, nrows - 1, commandPosition + commandOffset ) ;
         wrefresh( w ) ;

         // Now there is no command: "Command: ".
         waitingForButtonLabel = false ;
      } else if ( waitingForCarNumber ) {
         // Back out the car number that was entered.
         commandOffset-- ;
         mvwdelch( w, nrows - 1, commandPosition + commandOffset ) ;
         wrefresh( w ) ;

         waitingForCarNumber = false ;
         waitingForButtonLabel = true ;
      } else if ( waitingForDirection ) {
         // Back out the floor that was entered.
         commandOffset-- ;
         mvwdelch( w, nrows - 1, commandPosition + commandOffset ) ;
         wrefresh( w ) ;

         // For example, here is the command so far: "Command: F".
         waitingForDirection  = false ;
         waitingForFloorLabel = true ;

      } else if ( waitingForFloorLabel ) {
         // Wipe out the command that was entered.
         commandOffset-- ;
         mvwdelch( w, nrows - 1, commandPosition + commandOffset ) ;
         wrefresh( w ) ;

         // Now there is no command: "Command: ".
         waitingForFloorLabel = false ;
      } else if ( waitingForFirstDigit ) {
         // Wipe out the command that was entered.
         commandOffset-- ;
         mvwdelch( w, nrows - 1, commandPosition + commandOffset ) ;
         wrefresh( w ) ;

         // Now there is no command: "Command: ".
         waitingForFirstDigit = false;
      } else if ( waitingForSecondDigit ) {
         // Back out the first digit that was entered.
         commandOffset-- ;
```

```
         mvwdelch( w, nrows - 1, commandPosition + commandOffset ) ;
         wrefresh( w ) ;

         // For example, here is the command so far: "Command: X".
         waitingForFirstDigit  = true ;
         waitingForSecondDigit = false ;

      } else if ( waitingForEnter ) {
         // Back out the second digit that was entered.
         commandOffset-- ;
         mvwdelch( w, nrows - 1, commandPosition + commandOffset ) ;
         wrefresh( w ) ;

         // Restore the previous time multiplier.
         timeMultiplier = previousTimeMultiplier ;
         partialTimeMultiplier /= 10 ;

         // For example, here is the command so far: "Command: X1".
         waitingForEnter = false ;
         waitingForSecondDigit = true ;

      } else {
         // Tell the user that there isn't anything to backspace over.
         beep() ;
      }
   } else if ( waitingForButtonLabel ) {
      if ( strchr( floorLabels, c ) == NULL
        && strchr( floorLabels, toupper( c ) ) == NULL ) {
         // Tell user that she didn't input a button label.
         beep() ;
      } else {
         if ( strchr( floorLabels, c ) == NULL ) {
            // The button label is uppercase, even though the user used
            // lower case.
            c = toupper( c ) ;
         }
         mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
         wrefresh( w ) ;

         // For example, here is the command so far: "Command: C2".
         theFloor = c ;
         waitingForButtonLabel = false ;
         waitingForCarNumber  = true ;
      }
   } else if ( waitingForCarNumber ) {
      int carNumber ;
      if ( c == '*' ) {
         carNumber = 0 ;
      } else {
         carNumber = c - '0' ;
      }

      // Ensure that the user entered a valid car number.
      if ( carNumber < 0 || carNumber > numberOfElevators ) {
         // Tell user that she didn't input a valid car number.
         beep() ;
      } else {
         mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
         wrefresh( w ) ;

         int startCar ;
         int endCar ;
```

```
         if ( carNumber == 0 ) {
            // We'll send the request to all cars.
            startCar = 1 ;
            endCar   = numberOfElevators ;
         } else {
            // We'll send the request just to the indicated car.
            startCar = carNumber ;
            endCar   = carNumber ;
         }
         for ( int i = startCar; i <= endCar; i++ ) {
            // We'll ignore a pushed button in a stopped car for the
            // floor that the car is currently on if the car is already
            // in the process of serving the floor.
            if ( carPositionAbove[ i ] != 0
              || carPositionLabel[ i ] != theFloor
              || !( servicingFloorCars & ( 1 << i ) ) ) {
               // Send the notification to push a button in the elevator car.
               char *floorLabel = strchr( floorLabels, theFloor ) ;
               int floorIndex  = floorLabel - floorLabels ;
               notification_Q.PushNotification( PUSH_CAR_BUTTON, controller,
                                       Component( i ), floorIndex,
                                       dummy, dummy, dummy ) ;

               // Remember that the we're turning on the button in
               // the car.
               carButtons[ i ] |= ( 1 << floorIndex ) ;

               // Update the elevator simulation display by lighting the
               // button.
               floorLabel = strchr( floorLabels, carPositionLabel[ i ] ) ;
               floorIndex  = floorLabel - floorLabels ;

               int floorLabelRow = floorLabelRows[ floorIndex ] ;
               pushCarButton( true, i, theFloor,
                              carPositionAbove[ i ], floorLabelRow ) ;
            }
            #define DEBUGGING
            #ifdef DEBUGGING
            else {
               // For debugging purposes, we'll send a "CXi" message
               // indicating that a button press wasn't available while
               // the car is on the same floor.
               if ( useNamedPipes ) {
                  const int noteBufLen = sizeof pushCarButtonCommand + 2 ;
                  static char noteBuf[ noteBufLen ] ; // includes "\n\0"
                  noteBuf[ 0 ] = pushCarButtonCommand[ 0 ] ; // note id
                  noteBuf[ 1 ] = 'X' ;                 // 'invalid'
                  noteBuf[ 2 ] = i + '0' ;             // car
                  noteBuf[ 3 ] = '\n' ;
                  noteBuf[ noteBufLen - 1 ] = '\0' ;
                  write( statusPipeFd, noteBuf, strlen( noteBuf ) ) ;
                  #ifdef VERBOSE_CONFIGURATION
                  {
                     char str[ 80 ] ; // ample
                     sprintf( str, "Written via pipe (%li chars)> noteBuf: "
                                   "'%s'", strlen( noteBuf ), noteBuf ) ;
                     cout << str << endl ;
                  }
                  #endif // def VERBOSE_CONFIGURATION
               }
            }
            #endif // DEBUGGING
```

```
        }

        // We'll erase the command in a bit, after the user sees it.
        gettimeofday( &tv, (struct timezone *)NULL ) ;
        commandEraseTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000
                           + DELAY_BEFORE_ERASING_COMMAND ;

        // For example, here is the complete command: "Command: C21".
        waitingForCarNumber  = false ;
        waitingToClearCommand = true ;
     }
  } else if ( waitingForFirstDigit ) {
     int theDigit = c - '0' ;
     if ( theDigit >= 1 && theDigit <= 9 ) {
        mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
        wrefresh( w ) ;

        partialTimeMultiplier = theDigit ;

        // For example, here is the command so far: "Command: X1".
        waitingForFirstDigit  = false ;
        waitingForSecondDigit = true ;
     } else {
        // Tell user that she didn't input a digit (1 to 9).
        beep() ;
     }
  } else if ( waitingForSecondDigit ) {
     if ( c == 0x0d ) { // the enter key (by experimentation)
        mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
        wrefresh( w ) ;
        timeMultiplier = partialTimeMultiplier ;

        // Show the time multiplier if it not 1;
        if ( timeMultiplier == 1 ) {
           mvwprintw( w, nrows - 1, 2, "    " ) ;
        } else {
           mvwprintw( w, nrows - 1, 2, "X%i", timeMultiplier ) ;
        }

        // For example, here is the complete command:
        // "Command: X1<enter>".
        waitingForSecondDigit = false ;
        waitingToClearCommand = true ;
     } else {
        int theDigit = c - '0' ;
        if ( theDigit >= 0 && theDigit <= 9 ) {
           mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
           wrefresh( w ) ;

           previousTimeMultiplier = timeMultiplier ;
           timeMultiplier = partialTimeMultiplier * 10 + theDigit ;

           // For example, here is the command so far: "Command: X10".
           waitingForEnter       = true ;
           waitingForSecondDigit = false ;
        } else {
           // Tell user that she didn't input a digit (0 to 9).
           beep() ;
        }
     }
  } else if ( waitingForEnter ) {
     if ( c == 0x0d ) { // the curses enter key (by experimentation)
```

```
         mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
         wrefresh( w ) ;

         // Show the time multiplier if it not 1;
         if ( timeMultiplier == 1 ) {
            mvwprintw( w, nrows - 1, 2, "    " ) ;
         } else {
            mvwprintw( w, nrows - 1, 2, "X%i", timeMultiplier ) ;
         }

         // For example, here is the complete command:
         // "Command: X10<enter>".
         waitingForEnter = false ;
         waitingToClearCommand = true ;
      } else {
         // Tell user that she didn't input the enter key.
         beep() ;
      }
   } else if ( waitingForDirection ) {
      // Start with the floor label as entered by the user.
      char *floorLabel = strchr( floorLabels, theFloor ) ;
      int floorIndex  = floorLabel - floorLabels ;
      c = toupper( c ) ;
      if ( ( c != 'U' && c != 'D' )
        || ( c == 'U' && floorIndex == ( numberOfFloors - 1 ) )
        || ( c == 'D' && floorIndex == 0 ) ) {
         // Tell user that she didn't input a direction, or the
         // direction isn't valid for the floor.
         beep() ;
      } else {
         theDirection = ( c == 'U' ) ? up : down ;

         mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
         wrefresh( w ) ;

         // Pick a car to handle this floor button request.  We pick the
         // car by priority:
         //   1. closest available car to target floor
         //   2. closest car traveling towards the target floor
         //   3. car traveling in opposite direction with nearest final
         //      target destination
         int selectedCarNumber  = 0 ; // invalid default
         int distanceCarToFloor = 9 ; // worst-case default
         char *pCarFloorLabel ;
         int carFloorIndex ;
         for ( int i = 1; i <= numberOfElevators; i++ ) {
            if ( availableCars & ( 1 << i ) ) {
               pCarFloorLabel = strchr( floorLabels,
                                        carPositionLabel[ i ] ) ;
               carFloorIndex  = pCarFloorLabel - floorLabels ;
               if ( abs( floorIndex - carFloorIndex )
                         < distanceCarToFloor ) {
                  // OK, we'll select the closest available car.
                  selectedCarNumber = i ;
                  distanceCarToFloor = abs( floorIndex - carFloorIndex ) ;
               }
            }
         }
         if ( !selectedCarNumber ) {
            // We didn't find an available car, so we'll try to find
            // one approaching the target floor.
            selectedCarNumber = 3 ; // **??** TBD Just fake it for now. **??**
```

```
         }


         // Now send a notification to the chosen car to add this target to
         // its list.
         notification_Q.PushNotification( FLOOR_CALL, controller,
                                  Component( selectedCarNumber ),
                                  floorIndex, theDirection, dummy, dummy ) ;

         // Remember that the we're requesting that this car handle
         // the floor call request.
         if ( theDirection == up ) {
            floorCallUp[ selectedCarNumber ] |= ( 1 << floorIndex ) ;
         } else { // theDirection == down
            floorCallDown[ selectedCarNumber ] |= ( 1 << floorIndex ) ;
         }

         // Update the elevator simulation display by lighting the button.
         int  floorLabelRow = floorLabelRows[ floorIndex ] ;
         pushFloorButton( true, theDirection, udButtonColumn,
                          floorLabelRow ) ;

         // We'll erase the command in a bit, after the user sees it.
         gettimeofday( &tv, (struct timezone *)NULL ) ;
         commandEraseTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000
                            + DELAY_BEFORE_ERASING_COMMAND ;

         // For example, here is the complete command: "Command: F2D".
         waitingForDirection   = false ;
         waitingToClearCommand = true ;
      }
   } else if ( waitingForFloorLabel ) {
      if ( strchr( floorLabels, c ) == NULL
        && strchr( floorLabels, toupper( c ) ) == NULL ) {
         // Tell user that she didn't input a floor label.
         beep() ;
      } else {
         if ( strchr( floorLabels, c ) == NULL ) {
            // The floor label is uppercase, even though the user used
            // lower case.
            c = toupper( c ) ;
         }
         mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
         wrefresh( w ) ;

         // For example, here is the command so far: "Command: F2".
         theFloor = c ;
         waitingForFloorLabel = false ;
         waitingForDirection  = true ;
      }
   } else if ( ( c = toupper( c ) ) == 'F' ) {
      mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
      wrefresh( w ) ;

      // Here is the command so far: "Command: F".
      waitingForFloorLabel = true ;
   } else if ( ( c = toupper( c ) ) == 'C' ) {
      mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
      wrefresh( w ) ;

      // Here is the command so far: "Command: C".
      waitingForButtonLabel = true ;
```

```
   } else if ( ( c = toupper( c ) ) == 'H' ) {
   } else if ( ( c = toupper( c ) ) == 'X' ) {
      mvwaddch( w, nrows - 1, commandPosition + commandOffset++, c ) ;
      wrefresh( w ) ;

      // Here is the command so far: "Command: X".
      waitingForFirstDigit = true ;
   } else {
      // Tell user that she didn't input a command.
      beep() ;
   }
}
```

```
<<pause for user to read any error message>>=
// Pause for the user to read any error messages.
if ( shouldPause ) {
   curRow++ ; // blank line between messages
   mvwaddstr( w, curRow++, 3, "Press any key to continue..." ) ;
   wrefresh( w ) ;
   while( kbhit() == false ) {
      wait( 500 ) ; // ms
   }
   wgetch( w ) ; // collect the character
}
```

car

The car coroutine simulates the elevator car behavior for the specified shaft. It receives notifications from the simulation controller (elevatorSimulation).

We'll use two type definitions, one for car direction and a second for the open state of a car door.

```
<<type definitions>>=
typedef enum direction { up, down, none } direction ;
typedef enum openness { doorHalfOpen, doorOpen, doorOpenWaiting, doorHalfClosed,
                        doorClosed, doorLocked } openness ;
```

Notifications have varying requirements for data parameters, from none to four. We specify any unused parameters as "dummy" in calls to PushNotification for readability.

```
<<constants>>=
const int dummy = 0 ; // unused parameter
```

The user can speed up the simulation with the "x" command. The multiplier can be an integer between 1 and 10. The multiplier value is held in the global variable timeMultiplier and used to dynamically adjust waiting times to effect the desired speed of the simulation. The default is 1 (realtime).

```
<<global variables>>=
unsigned int timeMultiplier = 1 ;
```

```
<<elevator coroutines>>=
void car( int shaft ) {
   char      atFloor ;
   int       buttons               = 0 ;
   int       floorIndex ;
   int       floorStopRequestsDown = 0 ;
   int       floorStopRequestsUp   = 0 ;
   bool      moving                = false ;
   direction myDir                 = up ;   // initial direction
```

```
openness  nextDoorState         = doorClosed ; // initial state
int       position              = 0 ;
bool      waitingForDoor        = false ;

// We'll start out on the ground floor.
atFloor = floorLabels[ groundFloor ] ;
char *pGroundFloorLabel = strchr( floorLabels, atFloor ) ;
floorIndex              = pGroundFloorLabel - floorLabels ;

// Tell the simulation controller that we are available.
notification_Q.PushNotification( CAR_AVAILABLE, Component( shaft ), controller,
                        dummy, dummy, dummy, dummy ) ;

// Tell the simulation controller that we are not servicing floors.
notification_Q.PushNotification( CAR_NOT_SERVICING_FLOOR, Component( shaft ),
                        controller, dummy, dummy, dummy, dummy ) ;

// Tell the simulation controller (elevatorSimulation) where we are.
notification_Q.PushNotification( CAR_POSITION, Component( shaft ), controller,
                        position, atFloor, myDir, dummy ) ;

// Behave like an elevator until the simulation ends.
unsigned int currentTimeMs ;
unsigned int doorPositionTimeMs ;
unsigned int positionReachedTimeMs ;
struct timeval tv ;
while ( !orderlyStop ) {
   if ( waitingForDoor ) {
      gettimeofday( &tv, (struct timezone *)NULL ) ;
      currentTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;

      if ( currentTimeMs >= doorPositionTimeMs ) {
         if ( nextDoorState != doorOpenWaiting ) {
            // Tell the simulation controller our door position.
            notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                                    controller, nextDoorState,
                                    dummy, dummy, dummy ) ;
         }

         switch ( nextDoorState ) {
            case doorHalfOpen :
               nextDoorState = doorOpen ;
               doorPositionTimeMs = currentTimeMs
                                    + doorTransitTime / timeMultiplier ;
               break ;

            case doorOpen :
               nextDoorState = doorOpenWaiting ;
               doorPositionTimeMs = currentTimeMs
                                    + doorRemainsOpenTime / timeMultiplier ;
               break ;

            case doorOpenWaiting :
               nextDoorState = doorHalfClosed ;
               doorPositionTimeMs = currentTimeMs
                                    + doorTransitTime / timeMultiplier ;
               break ;

            case doorHalfClosed :
               nextDoorState = doorClosed ;
               doorPositionTimeMs = currentTimeMs
                                    + doorTransitTime / timeMultiplier ;
```

```
                break ;

            case doorClosed :
                waitingForDoor = false ;

                // Tell the simulation controller that we are finished
                // servicing this floor.
                notification_Q.PushNotification( CAR_NOT_SERVICING_FLOOR,
                                        Component( shaft ), controller,
                                        dummy, dummy, dummy, dummy ) ;

                // Set the time for the first position arrival if we are
                // going to move as soon as the door closes.
                if ( moving ) {
                    positionReachedTimeMs = currentTimeMs
                                          + positionTransitionTime
                                                              / timeMultiplier ;

                    // Tell the simulation controller our door is now locked.
                    notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                                          controller, doorLocked,
                                          dummy, dummy, dummy ) ;
                }

                // Tell the controller to turn off the direction indicator.
                notification_Q.PushNotification( SHOW_UP_DOWN_INDICATOR,
                                        Component( shaft ),
                                        controller, floorIndex, none,
                                        dummy, dummy ) ;

                if ( !buttons && !floorStopRequestsUp
                  && !floorStopRequestsDown ) {
                    notification_Q.PushNotification( CAR_AVAILABLE, Component( shaft ),
                                          controller,
                                          dummy, dummy, dummy, dummy ) ;
                    #ifdef SHOW_AVAILABILITY
                    // Tell the simulation controller where we are.
                    notification_Q.PushNotification( CAR_POSITION, Component( shaft ),
                                          controller, position, atFloor,
                                          myDir, dummy ) ;
                    #endif // def SHOW_AVAILABILITY
                }
                break ;

            case doorLocked :
                // This should not occur.
                // The case is included to avoid a clang compiler warning.
                break ;
        }
    }
} else if ( moving ) {
    bool targetReached = false ; // default
    bool extentReached = false ; // default
    gettimeofday( &tv, (struct timezone *)NULL ) ;
    currentTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;
    if ( currentTimeMs >= positionReachedTimeMs ) {
        // We've reached a positional change. We'll update our records
        // and notify the simulation controller so it can draw our updated
        // position. We always leave position with a zero value following
        // a transfer (i.e., position == 0 => we're on a floor boundary).
        if ( myDir == up ) {
            if ( ++position == MAX_DOOR_HEIGHT ) {
```

```
                position = 0 ;
                floorIndex++ ;
                atFloor = floorLabels[ floorIndex ];
                if ( buttons & ( 1 << floorIndex )
                  || floorStopRequestsUp & ( 1 << floorIndex ) )
                {
                   targetReached = true ;
                } else {
                   // Check for any targets above.
                   bool found = false ;
                   for ( int i = floorIndex + 1; i < numberOfFloors; i++ ) {
                      if ( buttons & ( 1 << i )
                        || floorStopRequestsUp & ( 1 << i ) ) {
                         found = true ;
                         break ;
                      }
                   }

                   // If there aren't any targets, see if there are
                   // pending floor down requests for me above this floor.
                   if ( !found ) {
                      for ( int i = floorIndex + 1; i < numberOfFloors; i++ ) {
                         if ( floorStopRequestsDown & ( 1 << i ) ) {
                            found = true ;
                            break ;
                         }
                      }
                      if ( !found ) {
                         // There aren't any stops for me above, so this is
                         // the time to turn around, if there are any stops
                         // below.
                         extentReached = true ;
                      }
                   }
                }
             }
          } else { // myDir == down
             if ( position == 0 ) {
                position = MAX_DOOR_HEIGHT - 1 ;
                floorIndex-- ;
                atFloor = floorLabels[ floorIndex ] ;
             } else if ( --position == 0 ) {
                if ( buttons & ( 1 << floorIndex )
                  || floorStopRequestsDown & ( 1 << floorIndex ) ) {
                   targetReached = true ;
                } else {
                   // Check for any targets below.
                   bool found = false ;
                   for ( int i = floorIndex - 1; i >= 0; i-- ) {
                      if ( buttons & ( 1 << i )
                        || floorStopRequestsDown & ( 1 << i ) ) {
                         found = true ;
                         break ;
                      }
                   }

                   // If there aren't any targets, see if there are
                   // pending floor up requests for me below this floor.
                   if ( !found ) {
                      for ( int i = floorIndex - 1; i >= 0; i-- ) {
                         if ( floorStopRequestsUp & ( 1 << i ) ) {
                            found = true ;
```

```
                        break ;
                    }
                }
                if ( !found ) {
                    // There aren't any stops for me below, so this is
                    // the time to turn around, if there are any stops
                    // above.
                    extentReached = true ;
                }
            }
        }
    }

    // Tell the simulation controller where we are.
    notification_Q.PushNotification( CAR_POSITION, Component( shaft ), controller,
                        position, atFloor, myDir, dummy ) ;
    // Continue to move.
    positionReachedTimeMs = currentTimeMs
                        + positionTransitionTime / timeMultiplier ;
}

// If we're at a target floor, we'll stop and open the door.
if ( targetReached ) {
    // Tell the simulation controller that we are going to stop on
    // this floor and open the door.
    notification_Q.PushNotification( CAR_SERVICING_FLOOR,
                        Component( shaft ), controller,
                        dummy, dummy, dummy, dummy ) ;

    // Tell the simulation controller our door is now unlocked.
    notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                        controller, doorClosed,
                        dummy, dummy, dummy ) ;

    // Tell the controller to turn on the direction indicator.
    notification_Q.PushNotification( SHOW_UP_DOWN_INDICATOR, Component( shaft ),
                        controller, floorIndex, myDir,
                        dummy, dummy ) ;

    // Remove the request for this stop.
    buttons &= ~( 1 << floorIndex ) ;
    if ( myDir == up ) {
        floorStopRequestsUp &= ~( 1 << floorIndex ) ;
    } else { // myDir == down
        floorStopRequestsDown &= ~( 1 << floorIndex ) ;
    }

    // Open the door.
    doorPositionTimeMs = currentTimeMs
                    + doorTransitTime / timeMultiplier ;
    moving             = false ;
    waitingForDoor     = true ;
    nextDoorState      = doorHalfOpen ;
} else if ( extentReached ) {
    moving = false ;

    if ( !buttons && !floorStopRequestsUp && !floorStopRequestsDown ) {
        notification_Q.PushNotification( CAR_AVAILABLE, Component( shaft ),
                            controller,
                            dummy, dummy, dummy, dummy ) ;
        #ifdef SHOW_AVAILABILITY
```

```
            // Tell the simulation controller where we are.
            notification_Q.PushNotification( CAR_POSITION, Component( shaft ),
                                    controller, position, atFloor,
                                    myDir, dummy ) ;
            #endif // def SHOW_AVAILABILITY
         }
      }
   }

   coresume() ;

   // Check for any notification.
   if ( !notification_Q.IsEmpty() && notification_Q.IsFor() == shaft ) {
      char         data0 ;
      char         data1 ;
      char         data2 ;
      char         data3 ;
      Component    destination ;
      Component    sender ;
      Notification      theNotification ;
      notification_Q.PopNotification( theNotification, sender, destination,
                           data0, data1, data2, data3 ) ;
      switch ( theNotification ) {
         case FLOOR_CALL :
            {
               direction tempDir = direction( data1 ) ;

               // Add the floor stop call to our list.
               if ( tempDir == up ) {
                  floorStopRequestsUp |= ( 1 << data0 ) ;
               } else { // tempDir == down
                  floorStopRequestsDown |= ( 1 << data0 ) ;
               }
            }
            break ;

         case PUSH_CAR_BUTTON :

            // Add the button to our list.
            buttons |= ( 1 << data0 ) ;
            break ;

         default :
            break ; // **??** TBD Put an alert here. **??**
      }
   }

   coresume() ;

   if ( ( buttons || floorStopRequestsUp || floorStopRequestsDown )
     && !moving && !waitingForDoor ) {
      //#define CHECK_POSITION
      #ifdef CHECK_POSITION
      assert( position == 0 ) ;
      #endif // def CHECK_POSITION

      // We are not moving and we have a request.
      int *tempFloorRequest ;
      if ( myDir == up ) {
         tempFloorRequest = &floorStopRequestsUp ;
      } else { // myDir == down
         tempFloorRequest = &floorStopRequestsDown ;
```

```
        }
        if ( buttons & ( 1 << floorIndex )
          || *tempFloorRequest & ( 1 << floorIndex ) ) {
           // The request is for the floor we're currently on, so we'll just
           // open the door.  First, we'll clear this floor from our buttons
           // and floor stop requests.
           buttons &= ~( 1 << floorIndex ) ;
           *tempFloorRequest &= ~( 1 << floorIndex ) ;

           // Tell the simulation controller that we are going to open the door
           // and service this floor.
           notification_Q.PushNotification( CAR_SERVICING_FLOOR,
                                 Component( shaft ), controller,
                                 dummy, dummy, dummy, dummy ) ;

           // Tell the simulation controller our door is now unlocked.
           notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                                 controller, doorClosed,
                                 dummy, dummy, dummy ) ;

           // Tell the controller to turn on the direction indicator.
           notification_Q.PushNotification( SHOW_UP_DOWN_INDICATOR, Component( shaft ),
                                 controller, floorIndex, myDir,
                                 dummy, dummy ) ;

           // Tell the controller to turn off the car and floor buttons.
           notification_Q.PushNotification( CAR_POSITION, Component( shaft ), controller,
                                 position, atFloor, myDir, dummy ) ;

           // Now, start opening the door.
           gettimeofday( &tv, (struct timezone *)NULL ) ;
           currentTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;
           doorPositionTimeMs = currentTimeMs
                              + doorTransitTime / timeMultiplier ;
           waitingForDoor = true ;
           nextDoorState = doorHalfOpen ;
        } else {
           if ( myDir == up ) {
              // Look for a target above our current position.
              for ( int i = floorIndex + 1; i < numberOfFloors; i++ ) {
                 if ( buttons & ( 1 << i )
                   || floorStopRequestsUp & ( 1 << i )
                   || floorStopRequestsDown & ( 1 << i ) ) {
                    moving         = true ;
                    notification_Q.PushNotification( CAR_NOT_AVAILABLE,
                                          Component( shaft ), controller,
                                          dummy, dummy, dummy, dummy ) ;

                    // Tell the simulation controller our door is now locked.
                    notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                                          controller, doorLocked,
                                          dummy, dummy, dummy ) ;
                    break ;
                 }
              }
           } else { // myDir == down
              // Look for a target below our current position.
              for ( int i = floorIndex - 1; i >= 0; i-- ) {
                 if ( buttons & ( 1 << i )
                   || floorStopRequestsDown & ( 1 << i )
                   || floorStopRequestsUp & ( 1 << i ) ) {
                    moving         = true ;
```

```
                notification_Q.PushNotification( CAR_NOT_AVAILABLE,
                                        Component( shaft ), controller,
                                        dummy, dummy, dummy, dummy ) ;

                // Tell the simulation controller our door is now locked.
                notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                                        controller, doorLocked,
                                        dummy, dummy, dummy ) ;
                break ;
             }
          }
       }

       // If we didn't find a target in the present direction, then
       // we'll reverse direction.
       if ( !moving ) {
          int *tempFloorRequest ;
          if ( myDir == up ) {
             myDir = down ;
             tempFloorRequest = &floorStopRequestsDown ;
          } else {
             myDir = up ;
             tempFloorRequest = &floorStopRequestsUp ;
          }
          if ( buttons & ( 1 << floorIndex )
            || *tempFloorRequest & ( 1 << floorIndex ) ) {
             // The request is for the floor we're currently on, so we'll
             // just open the door.  First, we'll clear this floor from
             // our buttons and floor stop requests.
             buttons &= ~( 1 << floorIndex ) ;
             *tempFloorRequest &= ~( 1 << floorIndex ) ;

             // Tell the simulation controller that we are going to open
             // the door and service this floor.
             notification_Q.PushNotification( CAR_SERVICING_FLOOR,
                                     Component( shaft ), controller,
                                     dummy, dummy, dummy, dummy ) ;

             // Tell the simulation controller our door is now unlocked.
             notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                                     controller, doorClosed,
                                     dummy, dummy, dummy ) ;

             // Tell the controller to turn on the direction indicator.
             notification_Q.PushNotification( SHOW_UP_DOWN_INDICATOR,
                                     Component( shaft ),
                                     controller, floorIndex, myDir,
                                     dummy, dummy ) ;

             // Tell the controller to turn off the car and floor buttons.
             notification_Q.PushNotification( CAR_POSITION, Component( shaft ),
                                     controller, position, atFloor, myDir,
                                     dummy ) ;

             // Now, start opening the door.
             gettimeofday( &tv, (struct timezone *)NULL ) ;
             currentTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;
             doorPositionTimeMs = currentTimeMs
                                + doorTransitTime / timeMultiplier ;
             waitingForDoor = true ;
             nextDoorState = doorHalfOpen ;
          } else {
```

```
                  moving         = true ;
                  notification_Q.PushNotification( CAR_NOT_AVAILABLE,
                                 Component( shaft ), controller,
                                 dummy, dummy, dummy, dummy ) ;

                  // Tell the simulation controller our door is now locked.
                  notification_Q.PushNotification( DOOR_POSITION, Component( shaft ),
                                 controller, doorLocked,
                                 dummy, dummy, dummy ) ;
               }
            }

            // Set the expected time of arrival at the next position.
            gettimeofday( &tv, (struct timezone *)NULL ) ;
            currentTimeMs = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;
            positionReachedTimeMs = currentTimeMs
                                  + positionTransitionTime / timeMultiplier ;
         }
      }

      coresume() ;
   }
}
```

pipeHandler

The pipeHandler coroutine provides a simulated kbhit / getch interface for elevator when it is running with a test process.

This coroutine periodically reads the pipe and provides commands for the elevator simulation.

When a command is available, it is presented character-by-character via the xkbhit and xgetch functions, in the same manner as the interface works when this program runs standalone (i.e., with useNamedPipes == false).

When a given command has been retrieved, we will check the pipe again for possible input, then wait a short time if none is found in order not to flood the pipe with empty requests.

pipeHandler sets the global variable xgetchCharAvailable to true when a character is available for the simulation to obtain with the xgetch function. xgetch sets xgetchCharAvailable to false after the character in xgetchChar is returned to the caller of xgetch.

```
<<global variables>>=
bool xgetchCharAvailable ;
int xgetchChar ;
```

```
<<elevator coroutines>>=
void pipeHandler( void ) {
   char      buf[ MAX_BUF_SIZE ] ;
   int       count ;
   int       numRead ;
   char      *pBuf ;
   bool      pipeInputAvailable = false ;

   xgetchCharAvailable = false ;
   while ( !orderlyStop ) {
      if ( pipeInputAvailable ) {
         if ( !xgetchCharAvailable ) {
            // We'll hand a char to the simulation, unless it's a nul.
            xgetchChar = *pBuf++ ;
            if ( !xgetchChar ) {
               pipeInputAvailable = false ;
            } else {
```

```
                xgetchCharAvailable = true ;
             }
          }
       } else {
          // We don't have any input from the pipe, so let's check for some.
          // Note that this is a non-blocking request.
          numRead = read( commandPipeFd, buf, MAX_BUF_SIZE ) ;
          if ( numRead == -1 ) {
             if ( errno == EAGAIN ) {
                // No data was available from the pipe, so we'll wait a while
                // and then try again.
                wait( pipeCheckInterval ) ;
             } else {
                // An error has occurred with the pipe.
                // **??** TBD send an alert **??**
                orderlyStop = true ;
             }
          } else if ( numRead > 0 ) {
             buf[ numRead ] = '\0' ;
             pBuf = buf ; // initialized to start of data
             pipeInputAvailable = true ;
          }
       }

       coresume() ;
    }
}
```

roundtripCounter

This optional coroutine provides a count of the number of coroutine cycles.

```
<<elevator coroutines>>=
void roundtripCounter( void ) {
   unsigned long coresumeCount = 0 ;
   unsigned int startTime ;
   unsigned int endTime ;
   struct timeval tv ;

   gettimeofday( &tv, (struct timezone *)NULL ) ;
   startTime = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;

   bool otherCoroutinesComplete = false ;
   while ( !otherCoroutinesComplete ) {
      #ifdef SHOW_HISTOGRAM
      itHistCR.tally();
      #endif // def SHOW_HISTOGRAM
      coresumeCount++ ;
      coresume() ;
      if ( orderlyStop && getCoroutineCount() < 2 ) {
         otherCoroutinesComplete = true ;
      }
   }

   gettimeofday( &tv, (struct timezone *)NULL ) ;
   endTime = tv.tv_sec * 1000 + tv.tv_usec / 1000 ;

   when( getCoroutineCount() < 2 ) {
      // This is the last running coroutine. The current coroutine
      // implementation requires returning from cobegin from a coroutine
```

```
      // with no calling parameters. Here we prepare a line to display after
      // leaving curses windowing.
      if ( endTime - startTime < 1000 ) {
         sprintf( roundtripCounterOutputString,
                  ">>> coresume cycle count: N/A." ) ;
      } else {
         sprintf( roundtripCounterOutputString,
                  ">>> coresume cycle count: %lu (%lu / sec).",
                  coresumeCount,
                  coresumeCount / ( ( endTime - startTime ) / 1000 ) ) ;
      }
   }
}
```

elevator's Utility Routines

checkInput

This routine checks the numberOfElevators and floorSpecification input and returns the numberOfFloors, groundFloor, and floorLabels. The floorLabels are returned in the caller's array, which must be at least MAX_NUMBER_OF_FLOORS + 1 bytes long.

In case of an error, a false indication is returned. Otherwise, true is returned.

This routine outputs any error messages to the console.

We insist on at least MIN_NUMBER_OF_ELEVATORS and no more than MAX_NUMBER_OF_ELEVATORS[24].

```
<<constants>>=
const int MIN_NUMBER_OF_ELEVATORS = 1 ;
```

```
<<elevator utilities>>=
bool checkInput( int numberOfElevators, const char *floorSpecification,
                 int &numberOfFloors, int &grndFloor, char *floorLabels ) {
   bool brc        = true ; // assume success
   int  noOfFloors = 0 ; // invalid
   grndFloor       = 0 ; // default level
   if ( strlen( floorSpecification ) == 0 ) {
      cout << "Error: No floor labels specified.\r\n" ;
      return false ;
   }
   if ( numberOfElevators < MIN_NUMBER_OF_ELEVATORS
     || numberOfElevators > MAX_NUMBER_OF_ELEVATORS ) {
      cout << "Error: Elevator count must be between 1 and "
           << int( MAX_NUMBER_OF_ELEVATORS ) << ".\r\n" ;
      return false ;
   }
   const char *pFl          = floorSpecification ;
   bool groundFloorIsMarked = false ;
   bool waitingForEndDigit  = false ;
   int  mostRecentDigit     =  0; // invalid
   char temp[ 17 ] ; // ample for 1 digit
   floorLabels[ 0 ] = '\0' ;
   for ( int i = 0; i < strlen( floorSpecification ); i++ ) {
      if ( waitingForEndDigit ) {
         temp[ 0 ] = *pFl ;
         temp[ 1 ] = '\0' ;
         int endDigit = atoi( temp ) ;
         for ( int i = mostRecentDigit + 1; i <= endDigit; i++ ) {
```

[24] MAX_NUMBER_OF_ELEVATORS is defined above in section elevatorSimulation.

```
            int flLen = strlen( floorLabels ) ;
            if ( flLen < MAX_NUMBER_OF_FLOORS ) {
               sprintf( temp, "%i", i );
               temp[ 1 ] = '\0' ;
               floorLabels[ flLen ] = temp[ 0 ] ;
               floorLabels[ flLen + 1 ] = '\0' ;
            } else {
               cout << "Error: Too many floor labels specified. The maximum is "
                    << int( MAX_NUMBER_OF_FLOORS ) << ".\r\n" ;
               return false ;
            }
         }
         waitingForEndDigit = false ;
      } else if ( isdigit( *pFl ) ) {
         temp[ 0 ] = *pFl ;
         temp[ 1 ] = '\0' ;
         mostRecentDigit = atoi( temp ) ;
         int flLen = strlen( floorLabels ) ;
         if ( flLen < MAX_NUMBER_OF_FLOORS ) {
            floorLabels[ flLen ] = *pFl ;
            floorLabels[ flLen + 1 ] = '\0' ;
         } else {
            cout << "Error: Too many floor labels specified. The maximum is "
                 << int( MAX_NUMBER_OF_FLOORS ) << ".\r\n" ;
            return false ;
         }
      } else if ( *pFl == '-' ) {
         waitingForEndDigit = true ;
      } else if ( *pFl == '#' ) {
         if ( groundFloorIsMarked ) {
            cout << "Error: Multiple ground floor specifications (#)."
                 << "\r\n" ;
            return false ;
         } else {
            grndFloor = strlen( floorLabels ) ;
            groundFloorIsMarked = true ;
         }
      } else if ( isalpha( *pFl ) ) {
         int flLen = strlen( floorLabels ) ;
         if ( flLen < MAX_NUMBER_OF_FLOORS ) {
            floorLabels[ flLen ] = *pFl ;
            floorLabels[ flLen + 1 ] = '\0' ;
         } else {
            cout << "Error: Too many floor labels specified. The maximum is "
                 << int( MAX_NUMBER_OF_FLOORS ) << ".\r\n" ;
            return false ;
         }
      } else {
         char str[ 80 ] ; // ample
         sprintf( str, "Unrecognized floor label specified: '%c'.", *pFl ) ;
         cout << str << "\r\n" ;
         return false ;
      }
      pFl++ ;
   }
   noOfFloors = strlen( floorLabels ) ;
   if ( noOfFloors < MIN_NUMBER_OF_FLOORS
     || noOfFloors > MAX_NUMBER_OF_FLOORS ) {
      cout << "Error: You must specify between 1 and "
           << int( MAX_NUMBER_OF_FLOORS ) << " floors.\r\n" ;
      brc = false ;
   } else {
```

```
      numberOfFloors = noOfFloors ;
   }

   return brc ;
}
```

drawCar

drawCar erases the previous image and then draws the elevator car at positions (0 to MAX_DOOR_HEIGHT - 1) above the specified floor. The buttons that have been pressed in the car are displayed in two columns. If the ground floor was overlaid previously, it is replaced with the appropriate character.

Here is an example of car 1 (depicted by the asterisks, with its left and right sides shown through the building's wall in "x-ray" fashion) at position 2 above floor *2*:

```
    *4        *
    *         *
    *  +----+*
    *  |    |*
[2] *B |    |*
       |    |
       +----+
```

Note that in-car buttons "B" and "4" have been pressed (they would be highlighted on screen).

Note that the available boolean is intended for debugging instrumentation.

The car height, CAR_HEIGHT, is the number of characters high (5).

```
<<constants>>=
const int CAR_HEIGHT = 5 ;
```

```
<<elevator utilities>>=
void drawCar( int shaftNumber, int positionAbove, char theFloor,
              char floorLabels[], int floorLabelRows[], int buttonsPressed,
              direction theDirection, int groundFloorRow
              #ifdef SHOW_AVAILABILITY
              , bool available
              #endif // def SHOW_AVAILABILITY
            ) {
   char *floorLabel = strchr( floorLabels, theFloor ) ;
   int floorIndex   = floorLabel - floorLabels ;
   int topRow       = floorLabelRows[ floorIndex ] - positionAbove - 2 ;
   int leftColumn   = ( shaftNumber - 1 ) * MAX_DOOR_WIDTH
                        + MAX_FLOOR_LABEL_WIDTH ;

   // Erase the previous image.
   if ( theDirection == up ) {
      // Erase the bottom row of the previous image. Replace the ground floor
      // chars as required.
      mvwaddstr( w, topRow + 5, leftColumn,
                 ( topRow + 5 ) == groundFloorRow ? "===" : "   " ) ;
      mvwaddch( w, topRow + 5, leftColumn + MAX_DOOR_WIDTH - 2,
                ( topRow + 5 ) == groundFloorRow ? '=' : ' ' ) ;
   } else { // direction == down
      // Erase the top row of the previous image. Replace the ground floor
      // chars as required.
      mvwaddstr( w, topRow - 1, leftColumn,
                 ( topRow - 1 ) == groundFloorRow ? "===" : "   " ) ;
      mvwaddch( w, topRow - 1, leftColumn + MAX_DOOR_WIDTH - 2,
```

```
                ( topRow - 1 ) == groundFloorRow ? '=' : ' ' ) ;
   }

   // Now display the car at the current position.
   for ( int i = 0; i < CAR_HEIGHT; i++ ) {
      #ifdef SHOW_AVAILABILITY
      mvwaddch( w, topRow + i, leftColumn, ( !i && available ) ? 'a' : '*' ) ;
      #else
      mvwaddch( w, topRow + i, leftColumn, '*' ) ;
      #endif // def SHOW_AVAILABILITY
      if ( buttonsPressed & ( 1 << ( CAR_HEIGHT - 1 - i ) ) ) {
         waddch( w, floorLabels[ CAR_HEIGHT - 1 - i ] | COLOR_PAIR( 7 ) ) ;
      } else {
         waddch( w, ' ' ) ;
      }
      if ( buttonsPressed & ( 1 << ( 2 * CAR_HEIGHT - 1 - i ) ) ) {
         waddch( w, floorLabels[ 2 * CAR_HEIGHT - 1 - i ] | COLOR_PAIR( 7 ) ) ;
      } else {
         waddch( w, ' ' ) ;
      }
      mvwaddch( w, topRow + i, leftColumn + MAX_DOOR_WIDTH - 2, '*' ) ;
   }
   wrefresh( w ) ;
}
```

drawOpenDoor

Draws an elevator door in one of three openness states: doorClosed, doorHalfOpen, or doorOpen.

The "x" and "y" values specify the upper-left corner of the door frame.

Here are examples of a closed door, as drawn by `initDoors`, a half-open door, and an open door (with *X* representing a dark rectangular character):

```
  +----+           +----+           +----+
  |    |           | XX |           |XXXX|
  |    |           | XX |           |XXXX|
  |    |           | XX |           |XXXX|
  +----+           +----+           +----+
doorClosed      doorHalfOpen       doorOpen
```

```
<<elevator utilities>>=
void drawOpenDoor( int y, int x, openness state ) {
   chtype doorPanel[ 4 ] ;

   doorPanel[ 0 ] = ' ' | ( state == doorOpen )   * A_REVERSE ;
   doorPanel[ 1 ] = ' ' | ( state != doorClosed ) * A_REVERSE ;
   doorPanel[ 2 ] = ' ' | ( state != doorClosed ) * A_REVERSE ;
   doorPanel[ 3 ] = ' ' | ( state == doorOpen )   * A_REVERSE ;

   // Draw the lines.
   for ( int i = 1; i <= 3; i++ ) {
      for ( int j = 1; j <= 4; j++ ) {
         mvwaddch( w, y + i, x + j, doorPanel[ j - 1 ] ) ;
      }
   }

   wrefresh( w ) ;
}
```

drawUpDownIndicator

Draws the up / down indicator above an elevator door.

The "x" and "y" values specify the upper-left corner of the door frame.

Here is an example of a door with a down direction indicator:

```
   v
+----+
|    |
|    |
|    |
+----+
```

Note that the direction indicators are colored, green for up and red for down.

```
<<elevator utilities>>=
void drawUpDownIndicator( int y, int x, direction indicatorDir ) {
   chtype c ;

   if ( indicatorDir == up ) {
      c = '^' | COLOR_PAIR( 3 ) ; // bold green on white
   } else if ( indicatorDir == down ) {
      c = 'v' | COLOR_PAIR( 2 ) ; // bold red on white
   } else { //  indicatorDir == none
      c = ' ' ;
   }
   mvwaddch( w, y - 1, x + 3, c ) ;

   wrefresh( w ) ;
}
```

initDoor

Draws the initial image of an elevator door with its associated buttons. The "x" and "y" values specify the upper-left corner of the door.

Here is an example of a (closed) door:

```
+----+
|    |
|    |
|    |
+----+
```

```
<<elevator utilities>>=
void initDoor( int y, int x ) {
   // Draw the top.
   mvwhline( w, y, x + 1, ACS_HLINE, 4 ) ;

   // Draw the left side.
   mvwvline( w, y + 1, x, ACS_VLINE, 3 ) ;

   // Draw the right side.
   mvwvline( w, y + 1, x + 5, ACS_VLINE, 3 ) ;

   // Draw the bottom.
   mvwhline( w, y + 4, x + 1, ACS_HLINE, 4 ) ;
```

```
   // Draw the upper-left corner.
   mvwaddch( w, y, x, ACS_ULCORNER ) ;

   // Draw the upper-right corner.
   mvwaddch( w, y, x + 5, ACS_URCORNER ) ;

   // Draw the lower-right corner.
   mvwaddch( w, y + 4, x + 5, ACS_LRCORNER ) ;

   // Draw the lower-left corner.
   mvwaddch( w, y + 4, x, ACS_LLCORNER ) ;
}
```

pushCarButton

Displays a call button (on or off) in the specified elevator car.

```
<<elevator utilities>>=
void pushCarButton( bool isOn, int shaft, char theCarButton,
                    int positionAbove, int floorLabelRow ) {
   int   buttonColumn ;
   int   buttonRow ;
   chtype theChar ;
   char  *floorLabel   = strchr( floorLabels, theCarButton ) ;
   int   floorIndex    = floorLabel - floorLabels ;
   int   topRow        = floorLabelRow - positionAbove - 2 ;
   int   carLeftColumn = ( shaft - 1 ) * MAX_DOOR_WIDTH + MAX_FLOOR_LABEL_WIDTH ;

   if ( floorIndex < CAR_HEIGHT ) {
      buttonColumn = carLeftColumn + 1 ;
      buttonRow    = topRow + CAR_HEIGHT - 1 - floorIndex ;
   } else { // floorIndex >= CAR_HEIGHT
      buttonColumn = carLeftColumn + 2 ;
      buttonRow    = topRow + 2 * CAR_HEIGHT - 1 - floorIndex ;
   }
   if ( isOn ) {
      theChar = theCarButton | COLOR_PAIR( 7 ) ; // black on bright yellow
   } else {
      theChar = theCarButton | COLOR_PAIR( 6 ) ; // black on light grey
   }
   mvwaddch( w, buttonRow, buttonColumn, theChar ) ;
   wrefresh( w ) ;
}
```

pushFloorButton

Displays a floor call button (on or off).

```
<<elevator utilities>>=
void pushFloorButton( bool isOn, direction theDirection,
                      int udButtonColumn, int floorLabelRow ) {
   chtype theChar ;
   if ( theDirection == up ) {
      if ( isOn ) {
         theChar = 'U' | COLOR_PAIR( 7 ) ; // black on bright yellow
      } else {
         theChar = 'U' | COLOR_PAIR( 6 ) ; // black on light grey
      }
```

```
      mvwaddch( w, floorLabelRow - 1, udButtonColumn, theChar ) ;
   } else { // theDirection is down
      if ( isOn ) {
         theChar = 'D' | COLOR_PAIR( 7 ) ; // black on bright yellow
      } else {
         theChar = 'D' | COLOR_PAIR( 6 ) ; // black on light grey
      }

      mvwaddch( w, floorLabelRow, udButtonColumn, theChar ) ;
   }
   wrefresh( w ) ;
}
```

strupr

A utility (missing in Unix gcc) to uppercase a string.

The input string is changed.

The returned string pointer points to the updated input string.

```
<<elevator utilities>>=
char *strupr( char *string ) {
   int strLen = strlen( string ) ;
   for ( int i = 0; i < strLen; i++ ) {
      string[ i ] = toupper( string[ i ] ) ;
   }
   return string ;
}
```

xgetch

A utility that simulates getch for pipe input.

This convenience function returns the next character transmitted across the named pipe being read to get test commands. Note that the character is returned only once.

`xgetchChar` is controlled by the `pipeHandler` coroutine.

```
<<elevator utilities>>=
int  xgetch( void ) {
   xgetchCharAvailable = false ;
   return xgetchChar ;
}
```

xkbhit

A utility that simulates kbhit for pipe input.

This convenience function indicates if a character is currently available on the pipe being read to get test commands.

`xgetchCharAvailable` is controlled by the `pipeHandler` coroutine.

```
<<elevator utilities>>=
bool xkbhit( void ) {
   return xgetchCharAvailable ;
}
```

Code Layout

In literate programming terminology, a *chunk* is a named part of the final program. The program chunks form a tree and the root of that tree is named * by default. We follow the convention of naming the root the same as the output file name. There are four roots in this literate program, the `elevator.cpp` file, the Notification header and Fifo class file, `fifo.h`, the named pipes header file, `pipe_interface.h`, and the `pipe_commands.cpp` file. The process of extracting the program tree formed by the chunks is called *tangle*. The program, `atangle`, extracts each root chunk to produce the corresponding C/C++ source file.

elevator.cpp

```
<<elevator.cpp>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   +elevator+ -- a test program for coroutines and ncurses.  <by Cary WR Campbell>
 *
 * Module:
 *   +elevator+ executable for macOS.
 *--
 */
/*
 * Configuration
 */
/****************************** Configuration ******************************/
/* Uncomment the following define line for a histogram of coroutine           **
** round-trip latencies.                                                     */
#define SHOW_HISTOGRAM
/* Uncomment the following define line for more info about the configuration  **
** sent from the test program.                                               */
//#define VERBOSE_CONFIGURATION
/* Uncomment the following define line to show elevator car availability.    */
//#define SHOW_AVAILABILITY
/***************************************************************************/
/*
 * Include files
 */
#include <iostream>
//#include <stdio.h>
<<include files>>
using namespace std ;
/*
 * Type Definitions
 */
<<type definitions>>
/*
 * Constants
 */
<<constants>>
/*
 * Variables
 */
<<global variables>>
/*
 * Forward References
 */
<<forward references>>
```

```
/*
 * elevator main Routine
 */
<<main routine>>
/*
 * elevator Coroutines
 */
<<elevator coroutines>>
/*
 * elevator Utility Routines
 */
<<elevator utilities>>
```

fifo.h

```
<<fifo.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   +elevator+ -- a test program for coroutines and ncurses.  <by Cary WR Campbell>
 *
 * Module:
 *   +Notification+ header and +Fifo+ queueing class file for message passing between
 *   +elevator+ simulation +Components+.
 *--
 */
/*
 * Constants and Typedefs
 */
<<notification constants and typedefs>>
/*
 * Fifo Class
 */
class Fifo {
   public:
<<fifo public>>
   private:
<<fifo private>>
} ;
```

pipe_interface.h

```
<<pipe_interface.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   +elevator+ -- a test program for coroutines and ncurses.  <by Cary WR Campbell>
 *
 * Module:
 *   Pipe interface header file for commands and status messages shared
 *   by elevator and a test process, e.g., Elevator Verification Test (EVT).
 *--
 */
/*
```

```
 * Pipe Definitions
 */
<<pipe definitions>>
/*
 * Pipe Constants
 */
<<pipe constants>>
/*
 * Pipe Externs
 */
extern const char endConfiguration[] ;
extern const char helloMessage[] ;
extern const char queryFloorLabels[] ;
extern const char queryMaxDimensions[] ;
extern const char quitTest[] ;
extern const char setBlockClearTime[] ;
extern const char setDoorOpenCloseTime[] ;
extern const char setFloorHeight[] ;
extern const char setFloorLabels[] ;
extern const char setGroundFloorLevel[] ;
extern const char setMaxCabinVelocity[] ;
extern const char setMaxCloseAttempts[] ;
extern const char setMinStoppingDistance[] ;
extern const char setNormalDoorWaitTime[] ;
extern const char setNumberOfElevators[] ;
extern const char setNumberOfFloors[] ;
extern const char userRequestedQuitTest[] ;
extern const char obstructDoorCommand[] ;
extern const char pushCarButtonCommand[] ;
extern const char pushFloorButtonCommand[] ;
extern const char P_BAD[] ;
extern const char P_OK[] ;
extern const char P_END[] ;
extern const char carLocation[] ;
extern const char doorIsClosed[] ;
extern const char doorIsLocked[] ;
extern const char doorIsAjar[] ;
extern const char doorIsOpen[] ;
extern const char indicatorIsDown[] ;
extern const char indicatorIsUp[] ;
extern const char indicatorIsOff[] ;
extern const char stopIsRequested[] ;
extern const char stopIsCleared[] ;
extern const char floorCallIsUp[] ;
extern const char floorCallIsDown[] ;
extern const char floorUpIsCleared[] ;
extern const char floorDownIsCleared[] ;
```

pipe_commands.cpp

```
<<pipe_commands.cpp>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   +elevator+ -- a test program for coroutines and ncurses.  <by Cary WR Campbell>
 *
 * Module:
 *   Pipe commands and status messages shared by elevator and a test process,
```

```
 *   e.g., Elevator Verification Test (EVT).
 *--
 */
/*
 * Test start and stop commands (from EVT -> elevator):
 */
<<pipe start and stop commands>>
/*
 * Test configuration commands (from EVT -> elevator) with responses
 * (from elevator -> EVT):
 *
 * f=float, i=integer, s=string, y=#floors, x=#elevators, g=grndFlr
 * Note: lengths are in meters, times in milliseconds, velocity in meters/second.
 */
<<pipe configuration commands>>
/*
 * Test stimulation commands (from EVT -> elevator) with no response:
 */
<<pipe stimulation commands>>
const char obstructDoorCommand[]    = "Oax"; // "Hold | release shaft x door."
const char pushCarButtonCommand[]   = "Cyx"; // "Push button y in car x."
const char pushFloorButtonCommnd[]  = "Fyi"; // "Push u or d button on floor y."
/*
 * Statuses (from elevator -> EVT):
 */
<<pipe statuses>>
/*
 * Notifications (from elevator -> EVT):
 */
 <<pipe notifications>>
const char carLocation[]            = "@yx"; // "Car x is at floor y."
const char doorIsClosed[]           = "%|x"; // "Car x's door is closed."
const char doorIsLocked[]           = "% x"; // "Car x's door is locked."
const char doorIsAjar[]             = "%-x"; // "Car x's door is half-way open."
const char doorIsOpen[]             = "%Ox"; // "Car x's door is completely
                                             //    open."
const char indicatorIsDown[]        = "~vx"; // "Car x's indicator is 'down'."
const char indicatorIsUp[]          = "~^x"; // "Car x's indicator is 'up'."
const char indicatorIsOff[]         = "~ x"; // "Car x's indicator is off."
const char stopIsRequested[]        = "+yx"; // "Car x's floor y button is lit."
const char stopIsCleared[]          = ".yx"; // "Car x's floor y button is off."
const char floorCallIsUp[]          = "*^y"; // "Floor y up call button is lit."
const char floorCallIsDown[]        = "*vy"; // "Floor y down call button lit."
const char floorUpIsCleared[]       = "-^y"; // "Floor y up call button is off."
const char floorDownIsCleared[]     = "-vy"; // "Floor y down call button off."
```

Edit Warning

We want to make sure to warn readers that the source code is generated and not manually written.

```
<<edit warning>>=
/*
 * DO NOT EDIT THIS FILE!
 * THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
 */
```

Copyright Information

The following is copyright and licensing information.

```
<<copyright info>>=
/*

 * Copyright (c) 2003 - 2021 Codecraft, Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
```

References

Books

[1] [ls-elevator] Leon Starr, Executable UML - The Elevator Project, Model Integration LLC (2001), ISBN 0-9708044-0-7.

Articles

[2] [pbh-edison] Per Brinch Hansen, Edison—a Multiprocessor Language, Software Practice and Experience 11, no. 4 (April 1981): 325 - 362.

Programs

[3] [cc-darts-program] Cary WR Campbell, The darts Literate Program, March 2021, https://drive.google.com/-file/d/1qOwo0P0nbFlritH3ha4FHqa6i1VwOTwS/view?usp=sharing.

[4] [cc-darts-mac] Cary WR Campbell, The darts Program Executable File—macOS Version, March 2021, https://drive.google.com/file/d/1fmRXu69oFyWag_Li-UGkLO-H0-Ufwt_q/view?usp=sharing.

[5] [cc-darts-win] Cary WR Campbell, The darts Program Executable File—Windows (Cygwin) Version, March 2021, https://drive.google.com/file/d/1MaQGIpnH1-i7DltL2f_RQaS0BBzxkOET/view?usp=sharing.

[6] [cc-elevator-mac] Cary WR Campbell, The elevator Program Executable File—macOS Version, March 2021, xxxxxx.

[7] [cc-elevator-win] Cary WR Campbell, The elevator Program Executable File—Windows (Cygwin) Version, March 2021, xxxxxx.

Projects

[8] [cc-darts-git] Cary WR Campbell, The darts Program GitHub Project, August 2021, https://github.com/-Codecraft-Inc-Montpelier-VA/darts.

[9] [cc-elevator-git] Cary WR Campbell, The elevator Program GitHub Project, August 2021, https://github.com/-Codecraft-Inc-Montpelier-VA/elevator.

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source code which is then given to the `gcc` program. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the code in an order suitable for a language processor. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunk definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is in an acceptable order.

Index