The `darts` Program

A Visual Test of Simple C/C++ Coroutines

Copyright © 2021 Codecraft, Inc.

Legal Notices and Information

This document is copyrighted 2021 Codecraft, Inc. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	March 12, 2021	First version of darts as a literate program.	CWRC
1.1	March 16, 2021	Revised to distribute userInterface keybindings into chunks associated with invoked coroutines.	CWRC

Contents

Introduction

This document describes the `darts` source code and provides instructions for its execution.

The `darts` program is a visual test program for Simple C/C++ Coroutines using Codecraft's open-source sccor library. `darts` runs in an ncurses[1] Terminal window as a command-line executable. This simple 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 `darts` is an x86_64 executable running on macOS and Windows (Cygwin).

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 `darts` test program for simple C/C++ coroutines. Further information on the particular literal programming syntax used here is given in Literate Programming.

Background

The `darts` program was initially developed as a test vehicle for the sccor library. Recently I decided to implement `darts` as a literate program to better explain the `darts` design and logic in an order and fashion that hopefully facilitates your understanding of the implementation, as well as providing all of the code.

I hope that the `darts` program code presented here demonstrates how easy it can be to create an application with many cooperative tasks by the use of coroutines. These coroutines are just standard C/C++ procedures, with the simple addition of two 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.

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

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

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

```
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
```

[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] See *Software Practice and Experience*, Volume 11 No 4, devoted to the Edison papers, Per Brinch Hansen.

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 parms ('b' and 10)
            repeatChar, 2, 'a', 10       // 2 parms ('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" command to complete execution of a coroutine. Coroutines exit by the ordinary C/C++ procedure behavior, either by an `exit` statement or just "falling off" the end of the function.

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

Overview

An interactive and visual representation of multiple coroutines’ execution allows an observation of their speed and consistency of performance as the number of instances increases.

The `darts` program provides the user with a graphical Terminal screen and a keyboard interface with which to start and stop instances of four coroutines:

1. `rightArrows` makes ‘>’ darts from left to right and back on a randomly-chosen row.
2. `leftArrows` makes ‘<’ darts from right to left and back on a randomly-chosen row.
3. `upArrows` makes ‘A’ darts from bottom to top and back on a randomly-chosen column.
4. `downArrows` makes ‘V’ darts from top to bottom and back on a randomly-chosen column.

When the `darts` program is executed, a blank Terminal window is shown initially.

Two coroutines are running, inconspicuously:

1. `userInterface`
2. `roundTripCounter`

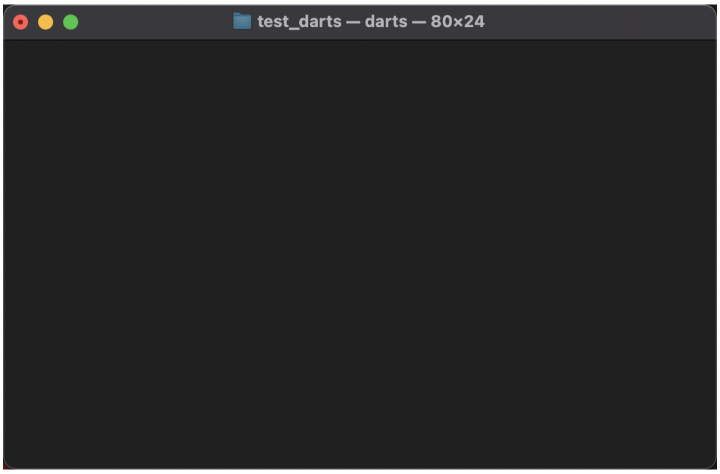
The `userInterface` coroutine reads keystrokes and performs the indicated user actions.

The `roundTripCounter` just counts the number of trips around the coroutine ring. Before exiting, `darts` displays the number of active coroutine instances at exit, the coroutine roundtrip count, and average time per cycle.

darts Execution Instructions

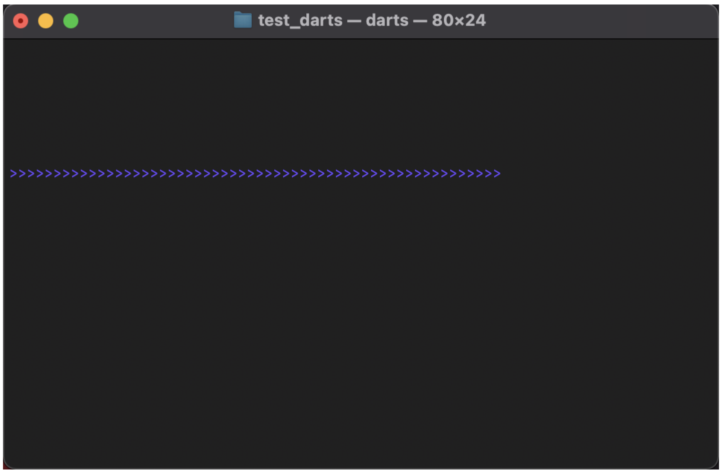
1. Run the darts executable in a macOS[3] or Windows (Cygwin) Terminal.

   darts starts with a blank Terminal screen:

2. Start an instance of one of the four darts coroutines (above) by keying the corresponding ‘>’, ‘<’, ‘A’, or ‘V’ (without the tick marks, and note that ‘A’ and ‘V’ are uppercase). Each instance is invoked with a randomly-selected color (the instance maintains its color during its lifetime).

   Here is (a still shot of) darts with one ‘>’ instance running:

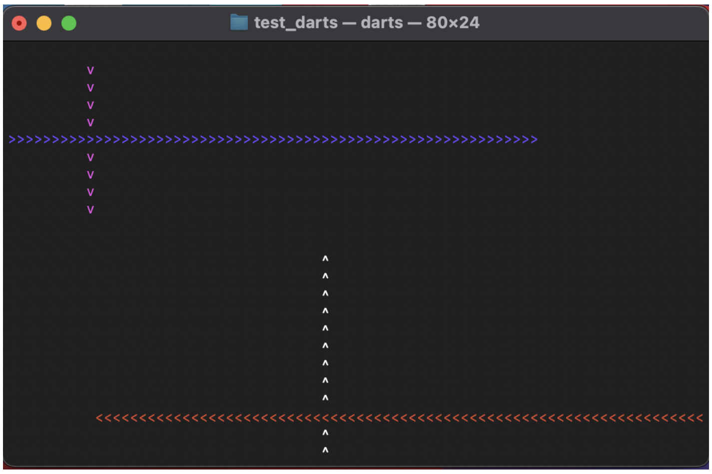
   Obviously, the actual running performance is more spectacular !

3. Continue to add more instances by keying ‘>’, ‘<’, ‘A’, or ‘V’.

   And try interspersing stopping the most recent instance of each type by keying ‘.’, ‘,’, ‘a’, or ‘v’, respectively.

[3] Big Sur (11.0), or later, is supported.

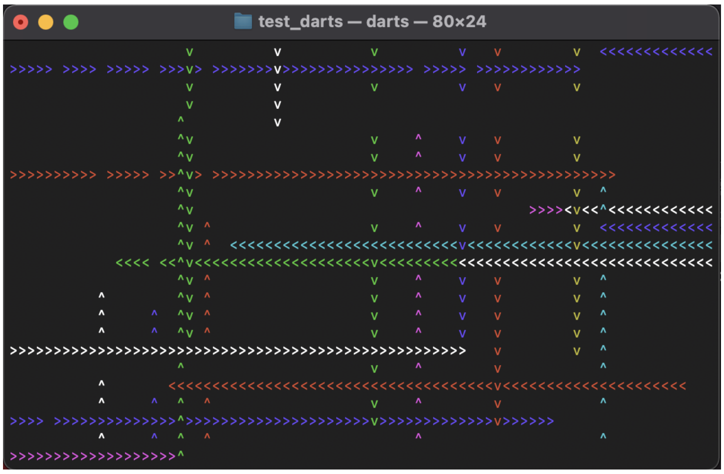
Here four instances are running, one of each persuasion:



Each dart instance chooses its color randomly, then keeps its color throughout its lifetime. For each traverse of the screen, a dart instance randomly picks its row (for ‘>’ and ‘<’ instances) or column (for ‘A’ and ‘V’ instances).

4. Rinse and repeat *ad nauseam*.

   Here are several instances running:



   The "holes" in an instances’s path are a result of another instance not restoring the previous contents while its path is returning to initial position.

5. Terminate the `darts` session by keying uppercase ‘Q’.

   Uppercase ‘Q’ quits `darts` gracefully. Lowercase ‘q’ panic stops, in mid-dart path.

Some Comments on `darts` Behavior

The default behavior of a `darts`’ instance is to take approximately one second to make an edge-to-edge transit across the screen, and then another second to return. This paced behavior may be overridden by keying a lowercase ‘w’, allowing all coroutines to execute at maximum velocity (*i.e.*, without waiting for pacing). Paced execution may be resumed by keying an uppercase ‘W’.

An instance of ‘>’, ‘<’, ‘A’, or ‘V’ may be terminated by keying ‘,’, ‘.’, ‘a’, or ‘v’, respectively. Note that these are the lowercase keys corresponding to the instance activating keys.

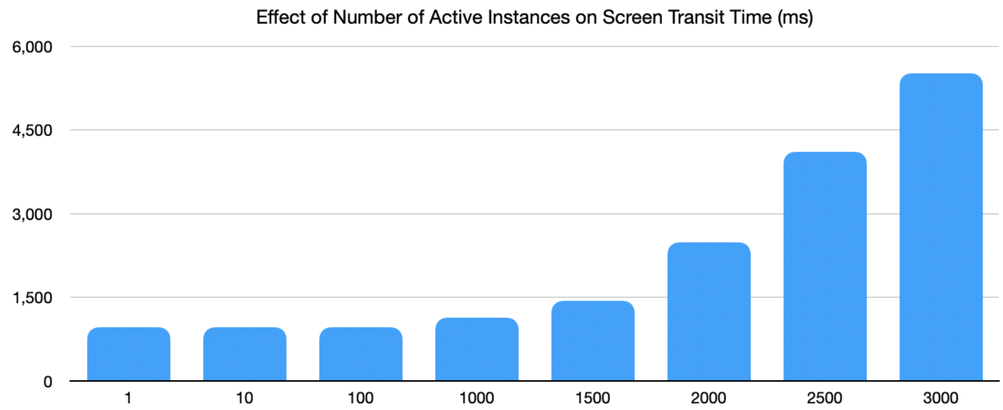
Having invoked a number of the various coroutine instances, the impatient user may key an uppercase ‘C’ to invoke 100 additional coroutine instances, 25 each of ‘>’, ‘<’, ‘A’, and ‘V’. A lowercase ‘c’ terminates the 100 most-recently-invoked instances.

Even more: keying uppercase ‘M’ invokes 1,000 additional instances, 250 each of ‘>’, ‘<’, ‘A’, and ‘V’. A lowercase ‘m’ terminates the 1000 most-recently-invoked instances.

These steps may be repeated as desired to gain an appreciation of the speed of these simple coroutines, especially without pacing. However, with a large number of instances, the visual display may become complicated, if not confusing.

`darts` limits the number of active instances to 3,000. As the number of instances approaches this limit, the time for each instance to transit the screen increases substantially due to the single-threaded nature of the sccor library coroutines implementation.

Here are some representative screen transit times[4] as the count of active dart instances increases:



Instance Count	Screen Transit Time (average, in ms)
1	961
10	961
100	962
1000	1,140
1500	1,437
2000	2,488
2500	4,110
3000	5,512

For this `darts` demonstration, with coroutine pacing implemented to produce target screen transit times of about one second, the performance is flat through at least 100 concurrent darts instances, with a slight performance decrease through the next 900 or so instances. Above that, the increased time required for a screen transit becomes noticeable. However, in the real world, more than 1,000 concurrently executing coroutine instances would be extraordinarily uncommon. For example, the number of concurrent instances employed in testing embedded medical devices with the Repeatable Random Test Generator (RRTGen), which utilizes these simple sccor Library coroutines, is generally less than 20.

In a real application, the performance of these simple coroutines must be analyzed with respect to the application’s performance requirements. In my experience, the performance of these coroutines is more than satisfactory. As a point of reference, with 100 dart instances running, the average elapsed time for an execution cycle (*i.e.*, from a given instance’s yielding with a `coresume` until it starts executing again at the statement after the `coresume`) is 21 microseconds. With 4 instances running, the elapsed time is 1.6 microseconds.

[4] On a MacBook Pro (16-inch, 2019) with 2.4 GHz 8-Core Intel Core I9, running in an 80x24 window.

Implementation

First, the coroutines comprising the darts program.

rightArrows Coroutine

Makes '>' darts from left to right and back on a randomly-chosen row until either orderlyStop or panicStop is true, or instanceId is greater than the global rightArrowsInstanceId value.

```
<<right arrows>>=
void rightArrows( chtype color, int instanceId )
{
   while ( !orderlyStop && !panicStop && instanceId <= rightArrowsInstanceId ) {
      int row = random() % nrows ;

      for ( int column = 0; !panicStop && column < ncols; column++ ) {
         wmove( w, row, column ) ;
         wechochar( w, ACS_RARROW | color ) ;
         if ( withPacing ) {
            wait( 1000 / ncols ) ; // ms
         } else {
            coresume() ;
         }
      }
      for ( int column = ncols - 1; !panicStop && column >= 0; column-- ) {
         wmove( w, row, column ) ;
         wechochar( w, ' ' ) ;
         if ( withPacing ) {
            wait( 1000 / ncols ) ; // ms
         } else {
            coresume() ;
         }
      }
   }
}
```

The rightArrows invocation keybinding to start one instance is '>' in the userInterface coroutine.

If the count of currently active coroutines is less than the maximum, an instance of rightArrows is invoked. If invoked, the instance is assigned a random color, which it keeps for its lifetime. Also, if invoked, the count of all rightArrows instances in increased by one.

```
<<right arrows invocation>>=
case '>':
   {
      if ( getCoroutineCount() <= MAX_DARTS ) {
         randomColor = COLOR_PAIR( random() % COLOR_COUNT ) ;
         invoke( (COROUTINE)rightArrows, 2, randomColor,
                 ++rightArrowsInstanceId ) ;
      }
   }
   break ;
```

The keybinding for stopping the latest rightArrows instance is '.'.

A positive count of currently running rightArrows instances is reduced by 1, causing the most-recently-invoked rightArrows instance to exit after completing its current transits, since its instanceId will be less than rightArrowsInstanceId.

```
<<right arrows termination>>=
case '.':
   {
```

```
      if ( rightArrowsInstanceId > 0 ) {
         --rightArrowsInstanceId ;
      } else {
         rightArrowsInstanceId = 0 ;                     // ❶
      }
   }
   break ;
```

❶ Being safe.

leftArrows Coroutine

Makes '<' darts from right to left and back on a randomly-chosen row until either `orderlyStop` or `panicStop` is true, or `instanceId` is greater than the global `leftArrowsInstanceId` value.

```
<<left arrows>>=
void leftArrows( chtype color, int instanceId )
{
   while ( !orderlyStop && !panicStop && instanceId <= leftArrowsInstanceId ) {
      int row = random() % nrows ;

      for ( int column = ncols - 1; !panicStop && column >= 0; column-- ) {
         wmove( w, row, column ) ;
         wechochar( w, ACS_LARROW | color ) ;
         if ( withPacing ) {
            wait( 1000 / ncols ) ; // ms
         } else {
            coresume() ;
         }
      }
      for ( int column = 0; !panicStop && column < ncols; column++ ) {
         wmove( w, row, column ) ;
         wechochar( w, ' ' ) ;
         if ( withPacing ) {
            wait( 1000 / ncols ) ; // ms
         } else {
            coresume() ;
         }
      }
   }
}
```

The `leftArrows` invocation keybinding to start one instance is '<' in the `userInterface` coroutine.

If the count of currently active coroutines is less than the maximum, an instance of `leftArrows` is invoked. If invoked, the instance is assigned a random color, which it keeps for its lifetime. Also, if invoked, the count of all `leftArrows` instances in increased by one.

```
<<left arrows invocation>>=
case '<':
   {
      if ( getCoroutineCount() <= MAX_DARTS ) {
         randomColor = COLOR_PAIR( random() % COLOR_COUNT ) ;
         invoke( (COROUTINE)leftArrows, 2, randomColor,
                 ++leftArrowsInstanceId ) ;
      }
   }
   break ;
```

The keybinding for stopping the latest `leftArrows` instance is ‘,’.

A positive count of currently running `leftArrows` instances is reduced by 1, causing the most recently invoked `leftArrows` instance to exit after completing its current transits, since its `instanceId` will be less than `leftArrowsInstanceId`.

```
<<left arrows termination>>=
case ',':
   {
      if ( leftArrowsInstanceId > 0 ) {
         --leftArrowsInstanceId ;
      } else {
         leftArrowsInstanceId = 0 ;
      }
   }
   break ;
```

`upArrows` Coroutine

Makes ‘A’ darts from bottom to top and back on a randomly-chosen column until either `orderlyStop` or `panicStop` is true, or `instanceId` is greater than the global `upArrowsInstanceId` value.

```
<<up arrows>>=
void upArrows( chtype color, int instanceId )
{
   while ( !orderlyStop && !panicStop && instanceId <= upArrowsInstanceId ) {
      int column = random() % ncols ;

      for ( int row = nrows - 1; !panicStop && row >= 0; row-- ) {
         wmove( w, row, column ) ;
         wechochar( w, ACS_UARROW | color ) ;
         if ( withPacing ) {
            wait( 1000 / nrows ) ; // ms
         } else {
            coresume() ;
         }
      }
      for ( int row = 0; !panicStop && row < nrows; row++ ) {
         wmove( w, row, column ) ;
         wechochar( w, ' ' ) ;
         if ( withPacing ) {
            wait( 1000 / nrows ) ; // ms
         } else {
            coresume() ;
         }
      }
   }
}
```

The `upArrows` invocation keybinding to start one instance is ‘A’ in the `userInterface` coroutine.

If the count of currently active coroutines is less than the maximum, an instance of `upArrows` is invoked. If invoked, the instance is assigned a random color, which it keeps for its lifetime. Also, if invoked, the count of all `upArrows` instances in increased by one.

```
<<up arrows invocation>>=
case 'A':
   {
      if ( getCoroutineCount() <= MAX_DARTS ) {
         randomColor  = COLOR_PAIR( random() % COLOR_COUNT ) ;
         invoke( (COROUTINE)upArrows, 2, randomColor,
                 ++upArrowsInstanceId ) ;
```

```
      }
   }
   break ;
```

The keybinding for stopping the latest `upArrows` instance is ‘a’.

A positive count of currently running `upArrows` instances is reduced by 1, causing the most recently invoked `upArrows` instance to exit after completing its current transits, since its `instanceId` will be less than `upArrowsInstanceId`.

```
<<up arrows termination>>=
case 'a':
   {
      if ( upArrowsInstanceId > 0 ) {
         --upArrowsInstanceId ;
      } else {
         upArrowsInstanceId = 0 ;
      }
   }
   break ;
```

`downArrows` Coroutine

Makes ‘V’ darts from top to bottom and back on a randomly-chosen column until either `orderlyStop` or `panicStop` is true, or `instanceId` is greater than the global `downArrowsInstanceId` value.

```
<<down arrows>>=
void downArrows( chtype color, int instanceId )
{
   while ( !orderlyStop && !panicStop && instanceId <= downArrowsInstanceId ) {
      int column = random() % ncols ;

      for ( int row = 0; !panicStop && row < nrows; row++ ) {
         wmove( w, row, column ) ;
         wechochar( w, /*ACS_DARROW*/ 'v' | color ) ;                  // ❶
         if ( withPacing ) {
            wait( 1000 / nrows ) ; // ms
         } else {
            coresume() ;
         }
      }
      for ( int row = nrows - 1; !panicStop && row >= 0; row-- ) {
         wmove( w, row, column ) ;
         wechochar( w, ' ' ) ;
         if ( withPacing ) {
            wait( 1000 / nrows ) ; // ms
         } else {
            coresume() ;
         }
      }
   }
}
```

❶ The ACS_DARROW character isn’t a down arrow; it’s more like the symbol for tautology (inverted T). So we’ll improvise with *v*.

The `downArrows` invocation keybinding to start one instance is ‘V’ in the `userInterface` coroutine.

If the count of currently active coroutines is less than the maximum, an instance of `downArrows` is invoked. If invoked, the instance is assigned a random color, which it keeps for its lifetime. Also, if invoked, the count of all `downArrows` instances in increased by one.

```
<<down arrows invocation>>=
case 'V':
   {
      if ( getCoroutineCount() <= MAX_DARTS ) {
         randomColor  = COLOR_PAIR( random() % COLOR_COUNT ) ;
         invoke( (COROUTINE)downArrows, 2, randomColor,
                 ++downArrowsInstanceId ) ;
      }
   }
   break ;
```

The keybinding for stopping the latest `downArrows` instance is ‘v’.

A positive count of currently running `downArrows` instances is reduced by 1, causing the most recently invoked `downArrows` instance to exit after completing its current transits, since its `instanceId` value will be less than `downArrowsInstanceId`.

```
<<down arrows termination>>=
case 'v':
   {
      if ( downArrowsInstanceId > 0 ) {
         --downArrowsInstanceId ;
      } else {
         downArrowsInstanceId = 0 ;
      }
   }
   break ;
```

`userInterface` Coroutine

This coroutine provides the user interface by waiting for a keyboard entry and then doing the corresponding action:

- > (or .) starts (or stops latest) right dart.
- < (or ,) starts (or stops latest) left dart.
- *A* (or *a*) starts (or stops latest) up dart.
- *V* (or *v*) starts (or stops latest) down dart.
- *X* (or *x*) starts (or stops latest) left X dart (for performance analysis).
- *W* (or *w*) starts (or stops pacing.
- *Q* (or *q*) quits gracefully (or panic stops, in mid-dart path).
- *C* (or *c*) starts (or stops latest) 100 dart instances, 25 of each type.
- *M* (or *m*) starts (or stops latest) 1000 dart instances, 250 of each type.

The keyboard hit function (`kbhit`) is a rather expensive action, since it interacts with the system. Consequently, and since there is no need for microsecond response times for the user interface, the `userInterface` coroutine checks for a keyboard hit only every 10th time it executes.

```
<<user interface>>=
#define CHECK_KB_FREQUENCY 10
```

The user interface *per se* consists of a `while` statement enclosing a big `switch` statement to pick out the keyboard entry and do the indicated action(s).

```
<<user interface>>=
void userInterface( void )
{
   char   c ;
   chtype randomColor ;
   int    loopCount = 0 ;

   while ( !orderlyStop && !panicStop ) {
      if ( ++loopCount >= CHECK_KB_FREQUENCY && kbhit() ) {
         loopCount = 0 ;
         c = getch() ;
         switch ( c ) {
            <<ui keybindings>>
         }
      }
      coresume() ;
   }
}
```

The `userInterface` coroutine is started by a `cobegin` statement in the `main` routine when `darts` begins execution.

`roundTripCounter` Coroutine

This coroutine provides a count of the number of coroutine cycles. 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`).

```
<<round trip counter>>=
void roundtripCounter( void )
{
   unsigned int 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 && !panicStop ) {
      #ifdef SHOW_LATENCY_HISTOGRAM
      itHistCR.tally() ;
      #endif
      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 ) {                             // ❶
      sprintf( roundtripCounterOutputString,                     // ❷
               ">>> coresume cycle count: %u (%u / sec).",
               coresumeCount,
               coresumeCount / ( ( endTime - startTime ) / 1000 ) ) ;
      sprintf( instancesAtStopOutputString,                      // ❸
               ">>>  darts count at stop: %u.", instancesActiveAtStop - 2 ) ;
```

```
   }
}
```

1. We wait for this to be the last running coroutine. The current coroutine implementation requires returning from cobegin from a coroutine with no calling parameters (like roundTripCounter).
2. Here we prepare a line to display after leaving ncurses windowing.
3. Another line to display post ncurses.

The roundTripCounter coroutine instance is started with main's cobegin statement when darts begins execution.

leftXs Coroutine

This is a performance testing coroutine, not normally invoked by darts users. Similar in its behavior to the other darts coroutines, leftXs makes *x* darts from right to left and back on a randomly-chosen row until either orderlyStop or panicStop is true, or instanceId is greater than the global leftXsInstanceId value. leftXs can collect data for a histogram showing individual and average screen transit times.

Note: There should be, at most, one leftXs instance executing. More than one instance running would produce incorrect transit times.

```
<<x arrows>>=
void leftXs( chtype color, int instanceId )
{
   while ( !orderlyStop && !panicStop && instanceId <= leftXsInstanceId ) {
      #ifdef SHOW_TRANSIT_TIME_HISTOGRAM
      itHistTT.tally() ;
      #endif
      int row = random() % nrows ;

      for ( int column = ncols - 1; !panicStop && column >= 0; column-- ) {
         wmove( w, row, column ) ;
         wechochar( w, 'x' | color ) ;
         if ( withPacing ) {
            wait( 1000 / ncols ) ; // ms
         } else {
            coresume() ;
         }
      }
      #ifdef SHOW_TRANSIT_TIME_HISTOGRAM
      itHistTT.tally() ;
      #endif
      for ( int column = 0; !panicStop && column < ncols; column++ ) {
         wmove( w, row, column ) ;
         wechochar( w, ' ' ) ;
         if ( withPacing ) {
            wait( 1000 / ncols ) ; // ms
         } else {
            coresume() ;
         }
      }
   }
}
```

The leftXs invocation keybinding to start one instance is 'X' in the userInterface coroutine.

If the count of currently active coroutines is less than the maximum, an instance of leftXs is invoked. If invoked, the instance is assigned a random color, which it keeps for its lifetime. Also, if invoked, the count of all leftXs instances in increased by one.

```
<<left x arrows invocation>>=
case 'X':
   {
      if ( getCoroutineCount() <= MAX_DARTS ) {
         randomColor  = COLOR_PAIR( random() % COLOR_COUNT ) ;
         invoke( (COROUTINE)leftXs, 2, randomColor,
                 ++leftXsInstanceId ) ;
      }
   }
   break ;
```

The keybinding for stopping the latest `leftXs` instance is ‘x’.

A positive count of currently running `leftXs` instances is reduced by 1, causing the most recently invoked `leftXs` instance to exit after completing its current transits, since its `instanceId` will be less than `leftXsInstanceId`.

```
<<left x arrows termination>>=
case 'x':
   {
      if ( leftXsInstanceId > 0 ) {
         --leftXsInstanceId ;
      } else {
         leftXsInstanceId = 0 ;
      }
   }
   break ;
```

Additional User Interface Keybindings

The keybinding in the `userInterface` coroutine for starting pacing (after it has been stopped by the user) is ‘W’ (for "wait").

Pacing causes each `darts` arrows instance to wait briefly after each character has been output to the screen. The wait time is calculated so that each screen traversal takes about one second[5].

```
<<pacing start>>=
case 'W':
   withPacing = true ;
   break ;
```

The keybinding in the `userInterface` coroutine to stop pacing is ‘w’.

When pacing is stopped, all `darts` arrow instances run "flat out", with no waiting between character outputs. There is just a `coresume` statement executed after each character is output, so all other coroutines are given a chance to run.

```
<<pacing stop>>=
case 'w':
   withPacing = false ;
   break ;
```

The keybinding in the `userInterface` coroutine to gracefully quit the execution of the `darts` program is ‘Q’.

Each running `darts` arrow instance will complete its current two screen traversals and exit.

The `main` routine will clean up, return the Terminal to normal (non-`ncurses`) mode, display its `darts` run statistics, and exit.

```
<<quitting gracefully>>=
case 'Q':
   instancesActiveAtStop = getCoroutineCount() ;
   orderlyStop = true ;
   break ;
```

[5] Screen traversal times are greater as the number of `darts` instances increases to the point where there is no wait time remaining before an instance is to output its next character.

The keybinding in the `userInterface` coroutine to panic quit the execution of the `darts` program is ‘q’.

Each running `darts` arrow instance will immediately exit.

The `main` routine will clean up, return the Terminal to normal (non-`ncurses`) mode, display its `darts` run statistics, and exit.

```
<<quitting panic stop>>=
case 'q':
   instancesActiveAtStop = getCoroutineCount() ;
   panicStop = true ;
   break ;
```

A "shorthand" keybinding of ‘C’ in the `userInterface` coroutine allows the user to invoke 100 `darts` arrow instances "at once".

Actually, after each invocation, a `coresume` statement is executed in order to give the just-invoked instance, as well as all other currently running instances, a chance to execute.

Twenty-five instances of each of the four flavors of `darts` arrows are invoked by a single ‘C’ key input.

```
<<century invocation>>=
case 'C':
   {
      for ( int i = 0; i < 25; i++ ) {
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR( random() % COLOR_COUNT ) ;
            invoke( (COROUTINE)rightArrows, 2, randomColor,
                    ++rightArrowsInstanceId ) ;
            coresume() ;
         }
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR(random() % COLOR_COUNT) ;
            invoke( (COROUTINE)leftArrows, 2, randomColor,
                    ++leftArrowsInstanceId ) ;
            coresume() ;
         }
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR(random() % COLOR_COUNT) ;
            invoke( (COROUTINE)upArrows, 2, randomColor,
                    ++upArrowsInstanceId ) ;
            coresume() ;
         }
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR(random() % COLOR_COUNT) ;
            invoke( (COROUTINE)downArrows, 2, randomColor,
                    ++downArrowsInstanceId ) ;
            coresume() ;
         }
      }
   }
   break ;
```

A keybinding of (lower-case) ‘c’ in the `userInterface` coroutine allows the user to stop execution of the 100 most-recently invoked `darts` arrow instances, twenty-five of each of the four flavors of `darts` arrows instances.

The instances are stopped normally, so each will complete its current screen traversals before exiting.

```
<<century termination>>=
case 'c':
   {
      if ( rightArrowsInstanceId > 25 ) {
         rightArrowsInstanceId -= 25 ;
      } else {
         rightArrowsInstanceId = 0 ;
```

```
      }
      if ( leftArrowsInstanceId > 25 ) {
         leftArrowsInstanceId -= 25 ;
      } else {
         leftArrowsInstanceId = 0 ;
      }
      if ( upArrowsInstanceId > 25 ) {
         upArrowsInstanceId -= 25 ;
      } else {
         upArrowsInstanceId = 0 ;
      }
      if ( downArrowsInstanceId > 25 ) {
         downArrowsInstanceId -= 25 ;
      } else {
         downArrowsInstanceId = 0 ;
      }
   }
   break ;
```

A "shorthand" keybinding of ‘M’ in the `userInterface` coroutine allows the user to invoke 1,000 `darts` arrow instances "at once".

Actually, after each invocation, a `coresume` statement is executed in order to give the just-invoked instance, as well as all other currently running instances, a chance to execute.

Two-hundred-fifty instances of each of the four flavors of `darts` arrows are invoked by a single ‘M’ key input.

```
<<millennium invocation>>=
case 'M':
   {
      for ( int i = 0; i < 250; i++ ) {
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR( random() % COLOR_COUNT ) ;
            invoke( (COROUTINE)rightArrows, 2, randomColor,
                    ++rightArrowsInstanceId ) ;
            coresume() ;
         }
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR(random() % COLOR_COUNT) ;
            invoke( (COROUTINE)leftArrows, 2, randomColor,
                    ++leftArrowsInstanceId ) ;
            coresume() ;
         }
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR(random() % COLOR_COUNT) ;
            invoke( (COROUTINE)upArrows, 2, randomColor,
                    ++upArrowsInstanceId) ;
            coresume() ;
         }
         if ( getCoroutineCount() <= MAX_DARTS ) {
            randomColor = COLOR_PAIR(random() % COLOR_COUNT) ;
            invoke( (COROUTINE)downArrows, 2, randomColor,
                    ++downArrowsInstanceId ) ;
            coresume() ;
         }
      }
   }
   break ;
```

A keybinding of (lower-case) ‘m’ in the `userInterface` coroutine allows the user to stop execution of the 1,000 most-recently invoked `darts` arrow instances, 250 of each of the four flavors of `darts` arrows instances.

The instances are stopped normally, so each will complete its current screen traversals before exiting.

```
<<millennium termination>>=
case 'm':
   {
      if ( rightArrowsInstanceId > 250 ) {
         rightArrowsInstanceId -= 250 ;
      } else {
         rightArrowsInstanceId = 0 ;
      }
      if ( leftArrowsInstanceId > 250 ) {
         leftArrowsInstanceId -= 250 ;
      } else {
         leftArrowsInstanceId = 0 ;
      }
      if ( upArrowsInstanceId > 250 ) {
         upArrowsInstanceId -= 250 ;
      } else {
         upArrowsInstanceId = 0 ;
      }
      if ( downArrowsInstanceId > 250 ) {
         downArrowsInstanceId -= 250 ;
      } else {
         downArrowsInstanceId = 0 ;
      }
   }
   break ;
```

`main` Routine

The `main` routine sets up the `ncurses` screen environment, starts the initial coroutines, and handles the usual program administration activities.

First, `main` initializes some counters.

```
<<main routine>>=
int   main ( int argc, char *argv[] )
{
   orderlyStop                = false ;
   panicStop                  = false ;
   withPacing                 = true ;
   downArrowsInstanceId       = 0 ;
   leftArrowsInstanceId       = 0 ;
   leftXsInstanceId           = 0 ;
   rightArrowsInstanceId      = 0 ;
   upArrowsInstanceId         = 0 ;
```

It initializes our random stream to a repeatable value.

```
<<main routine>>=
   srandom( 1 ) ;
```

Then sets up `ncurses`.

```
<<main routine>>=
   w = initscr() ;
   start_color() ; cbreak() ; noecho() ; leaveok( w, TRUE ) ; nonl() ;
   use_default_colors() ;
   init_pair( 1, short(COLOR_BLUE    | A_BOLD), short(-1) ) ;
   init_pair( 2, short(COLOR_RED     | A_BOLD), short(-1) ) ;
   init_pair( 3, short(COLOR_GREEN   | A_BOLD), short(-1) ) ;
   init_pair( 4, short(COLOR_YELLOW  | A_BOLD), short(-1) ) ;
```

```
   init_pair( 5, short(COLOR_MAGENTA | A_BOLD), short(-1) ) ;
   init_pair( 6, short(COLOR_CYAN    | A_BOLD), short(-1) ) ;
   init_pair( 7, short(COLOR_BLACK   | A_BOLD), short(-1) ) ;
   #define COLOR_COUNT 7
```

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

```
<<main routine>>=
   cbreak() ;
```

Next it finds the size of the window to keep darts within the window.

```
<<main routine>>=
   getmaxyx( w, nrows, ncols ) ;
```

Clearing the window and hiding the cursor.

```
<<main routine>>=
   int old_visibility ;
   old_visibility = curs_set( 0 ) ;
   clear() ;
   refresh() ;
```

Now `main` starts the initial two coroutines, `roundTripCounter` and `userInterface`. The `cobegin` statement blocks after starting the two coroutines, and remains blocked until after all coroutine instances have exited.

```
<<main routine>>=
   cobegin( 2,                                                      // ❶
               roundtripCounter, 0,                                 // ❷
               userInterface,    0                                  // ❸
          ) ;
```

❶ Initial coroutine count (“2”).

❷, ❸ No parameters (“0”).

After all coroutines have finished their executions, processing continues here, at the statement following the `cobegin` statement. The cursor is restored and console input is returned to cooked mode with echoing.

```
<<main routine>>=
   curs_set( old_visibility ) ;
   nocbreak() ; echo() ;
```

Knocking down `ncurses`.

```
<<main routine>>=
   endwin() ;
```

Before exiting, `main` shows the coroutine roundtrip count, average time per cycle, and the instance count at stop.

```
<<main routine>>=
   cout << roundtripCounterOutputString << endl << '\r' ;
   cout << instancesAtStopOutputString  << endl ;
```

Finally, `main` shows the optional histograms and returns.

```
<<main routine>>=
   #ifdef SHOW_LATENCY_HISTOGRAM
   itHistCR.show( false ) ;                                         // ❶
   itHistCR.reset() ;                                               // ❷
```

```
    #endif

    #ifdef SHOW_TRANSIT_TIME_HISTOGRAM
    itHistTT.show( false ) ;
    itHistTT.reset() ;
    #endif

    return( 0 ) ;
}
```

1. Omit log (“false”).
2. Clear histogram for next time.

Include Files

The iostream include file is required for the cout and endl functions.

```
<<include files>>=
#include <iostream>
```

The curses.h include file is required for the ncurses library.

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

The sys/time.h include file is required for gettimeofday.

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

The sccorlib.h include file provides access to the sccor library’s coroutine functionality.

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

The histospt.h include file provides histogram support.

```
<<include files>>=
#if defined (SHOW_LATENCY_HISTOGRAM) || defined (SHOW_TRANSIT_TIME_HISTOGRAM)
#include "histospt.h"
#endif
```

Constants

These global constants are shared by the coroutines.

The number of columns (ncols) and rows (nrows) are obtained from ncurses.

```
<<constants>>=
int ncols, nrows ;
```

If histograms are desired, their instantiation is here, one for each histogram type.

```
<<constants>>=
#ifdef SHOW_LATENCY_HISTOGRAM
 TimeIntervalHistogram  itHistCR( "coresume round-trip times (us)",
                                  0L, 250, 40 ) ; // 0 - 10 ms
#endif
```

```
#ifdef SHOW_TRANSIT_TIME_HISTOGRAM
 TimeIntervalHistogram  itHistTT( "screen transit times (us)",
                                  0L, 125000, 40 ) ; // 0 - 5000 ms
#endif
```

Variables

These global variables are shared by the coroutines.

'w' is the `ncurses` canvas that the darts coroutines write on.

```
<<variables>>=
WINDOW *w ;
```

These string buffers are where `roundTripCounter` stores the count of coroutine roundtrips, average time per roundtrip, and the instance count when `darts` is stopped. The allocated size of 80 bytes is ample for the strings. `main` writes these buffers to the console before exiting.

```
<<variables>>=
char roundtripCounterOutputString[80] ;
char instancesAtStopOutputString[80] ;
```

Some boolean flags. They are `volatile` so 'C' doesn't optimize code so that it misses changes to their values.

```
<<variables>>=
volatile bool orderlyStop ;
volatile bool panicStop ;
volatile bool withPacing ;
```

These counters control stopping the latest instance(s) of each kind of dart.

```
<<variables>>=
volatile int downArrowsInstanceId ;
volatile int leftArrowsInstanceId ;
volatile int leftXsInstanceId ;
volatile int rightArrowsInstanceId ;
volatile int upArrowsInstanceId ;
```

This variable allows sharing the final dart coroutine count just before quitting.

```
<<variables>>=
volatile int instancesActiveAtStop ;
```

Forward References

Here are the prototype declarations for all of the routines that comprise `darts`.

```
<<forward references>>=
int  kbhit( void ) ;
void userInterface( void ) ;
void roundtripCounter( void ) ;
void downArrows( chtype color, int instanceId ) ;
void leftArrows( chtype color, int instanceId ) ;
void rightArrows( chtype color, int instanceId ) ;
void upArrows( chtype color, int instanceId ) ;
void leftXs( chtype color, int instanceId ) ;
```

Configuration Options

The following define sets the maximum number of dart instances allowed. It should not be increased without investigating the current limits in the sccor Library.

```
<<configuration>>=
#define MAX_DARTS 3000
```

Besides reporting the coroutine roundtrip count and the average time per cycle, the `roundTripCounter` coroutine can optionally provide a histogram of the coroutine roundtrip cycle times (*i.e.*, the elapsed times, or latencies, recorded for successive executions of the `roundTripCounter` coroutine).

Uncomment the following define line for a histogram of coroutine roundtrip latencies.

```
<<configuration>>=
//#define SHOW_LATENCY_HISTOGRAM
```

In order to measure the screen transit time changes as the number of active dart instances increases, a special ‘X’ arrow dart (`leftXs`) may be started after a given number of normal ‘>’, ‘<’, ‘A’, and ‘V’ instances are running. Letting the ‘X’ instance execute for a few transits, a histogram can be produced showing the ‘X’ instance’s individual transit times and the average for the given number of instances.

The table and graph in the section “Some Comments on `darts` Behavior” were produced by this method, with the given number of instances being 1, 10, 100, 1000, 1500, 2000, 2500, and 3000.

Uncomment the following define line for a histogram of screen transit times for `leftXs` instance.

```
<<configuration>>=
//#define SHOW_TRANSIT_TIME_HISTOGRAM
```

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, in this case `darts.cpp`. The process of extracting the program tree formed by the chunks is called *tangle*. The program, `atangle`, extracts the root chunk to produce the C/C++ source file.

darts.cpp

```
<<darts.cpp>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   +darts+ -- a test program for coroutines and ncurses.  <by Cary WR Campbell>
 *
 * Module:
 *   +darts+ executable for macOS.
 *--
 */
/*
 * Configuration
 */
<<configuration>>
/*
 * Include files
 */
```

```
<<include files>>
using namespace std ;
/*
 * Constants
 */
<<constants>>
/*
 * Variables
 */
<<variables>>
/*
 * Forward References
 */
<<forward references>>
/*
 * darts main routine
 */
<<main routine>>
/*
 * darts Coroutines
 */
<<darts coroutines>>
```

darts Coroutines

In this chunk, we enumerate the darts coroutines.

```
<<darts coroutines>>=
<<right arrows>>
<<left arrows>>
<<up arrows>>
<<down arrows>>
<<x arrows>>
<<user interface>>
<<round trip counter>>
```

darts Keybindings

In this chunk, we enumerate the userInterface keybindings.

```
<<ui keybindings>>=
<<right arrows invocation>>
<<right arrows termination>>
<<left arrows invocation>>
<<left arrows termination>>
<<up arrows invocation>>
<<up arrows termination>>
<<left x arrows invocation>>
<<left x arrows termination>>
<<down arrows invocation>>
<<down arrows termination>>
<<century invocation>>
<<century termination>>
<<millennium invocation>>
<<millennium termination>>
<<pacing start>>
<<pacing stop>>
<<quitting gracefully>>
<<quitting panic stop>>
```

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.
 */
```

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