The `abc` Program

A Simple Test of Simple C/C++ Coroutines

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	July 21, 2021	First version of abc as a literate program.	CWRC

Contents

Introduction

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

The abc program is a simple test program for Simple C/C++ Coroutines using Codecraft's open-source sccor library. abc runs in a Terminal window on macOS or Windows (Cygwin).

The sccor Library's Simple C/C++ Coroutines implementation supports lightweight cooperative multitasking and provides for asynchronous programming through the use of Edison[1]-inspired, single-threaded, non-preemptive, ring-based coroutines.

This version of abc produces an x86_64 executable running on macOS[2] or Windows[3].[4]

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

Background

This abc test program was written to verify 64-bit-mode execution of my Simple C/C++ Coroutines running on both macOS and Windows.

In implementing abc 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.

The coroutines employed in abc 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.

As an example, repeatChar is a coroutine that writes its input character a specified number of times and returns. 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] [pbh-edison] is an edition of *Software Practice and Experience* devoted entirely to the Edison papers.
[2] See [cc-abc-mac] for an abc executable that runs on macOS.
[3] See [cc-abc-win] for an abc.exe executable that runs on Windows.
[4] A Linux version will be available in a future release, with the availability of a Linux version of the sccor library.

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 `abc` program.

abc Execution

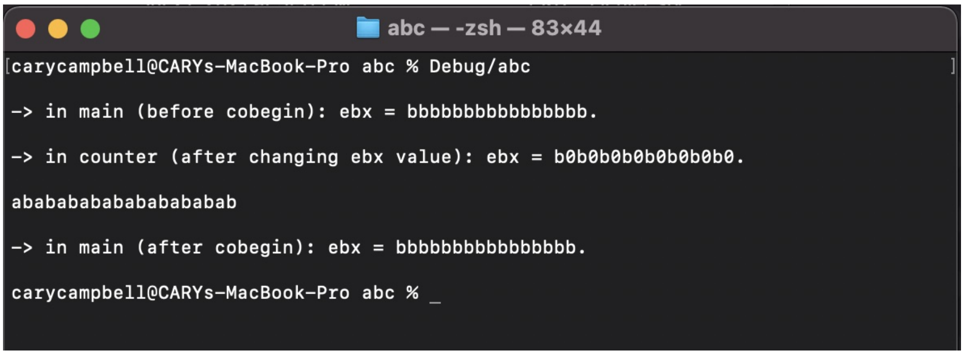
Run the `abc` executable in a macOS[5] Windows 10 Terminal window.

Syntax:

```
% abc
```

There are no options for the `abc` program.

Here is output from a run of the `abc` program on macOS:



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

Implementation

A primary consideration in developing the abc program is to create a simple demonstration using the Simple C/C++ Coroutines in the sccor library.

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

abc's main Routine

The abc 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 something>>
   <<take down>>
}
```

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

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

main's Set Up

A special value is inserted into the rbx register for comparison later, following execution of the coroutines. The rbx register value must be preserved unchanged by coroutine execution.

We are using the sccor library, so we need to include its header.

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

We are using sprintf(), *etc.*, so we need to include the header.

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

```
<<set up>>=
unsigned long _RBX = 0xbbbbbbbbbbbbbbbb ;
char temp[ 200 ] ;
asm ( "movq %0, %%rbx" : /* no outputs */ : "rm" (_RBX) : "%rbx" ) ;
stop = false ;
sprintf( temp, "\n-> in main (before cobegin): rbx = %08lx.\n", _RBX ) ;
strprt( temp ) ;
CR ;
```

main initializes a global boolean variable that signals the writeLetter coroutine instances to stop character outputting and exit. The variable is volatile so the compiler doesn't optimize out asychronous references to it in the coroutines.

```
<<global variables>>=
volatile bool stop ;
```

main uses a convenience "CR" define to generate a newline.

```
<<definitions>>=
#define CR puts( "\r" )
```

main uses the strprt utility routine to display a string.

```
<<forward references>>=
void strprt( const char *str_ptr ) ;
```

```
<<utilities>>=
void strprt( const char *str_ptr ) {
   while ( *str_ptr ) putchar( *str_ptr++ ) ;
}
```

main's Do Something.

main starts two instances of the writeLetter coroutine and one instance of the counter coroutine.

```
<<forward references>>=
void counter( int count ) ;
void writeLetter( char c ) ;
```

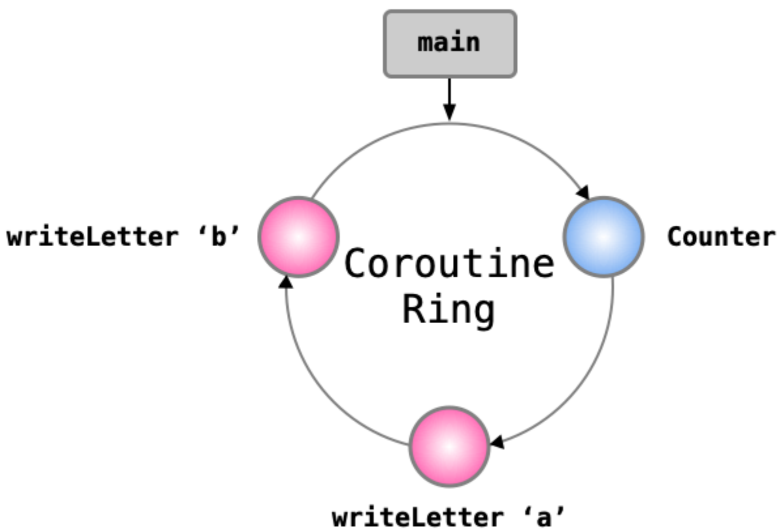
```
<<do something>>=
cobegin( 3,                                  // ❶
            writeLetter,     1, 'b',         // ❷
            writeLetter,     1, 'a',         // ❸
            counter,         1, 10           // ❹
       ) ;
CR ;
```

1. Initial coroutine count ("3").
2. One character parameter ("b").
3. One character parameter ("a").
4. One integer parameter ("10").

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

main's cobegin statement creates a new coroutine ring and places the writeLetter "b" instance, the writeLetter "a" instance, and the counter instance on it, then blocks awaiting completion of all coroutines on the ring.

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



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.

`writeLetter` Coroutine

Each `writeLetter` instance writes its designated character and then yields with a `coresume` statement, continuing until stopped by a global `stop` variable.

Here's the `writeLetter` code:

```
<<coroutines>>=
void writeLetter( char c ) {
   while ( stop == false ) {
      putchar( c ) ;
      coresume() ;
   }
}
```

`counter` Coroutine

The `counter` instance counts to the specified count, sets the global `stop` variable to tell the other coroutines to stop, and exits. Each time after incrementing its count, `counter` yields with a `coresume` statement.

Here's the `counter` code:

```
<<coroutines>>=
void counter( int count ) {
   unsigned long _RBX = 0xb0b0b0b0b0b0b0b0 ;
   char temp[ 200 ] ;
   asm ( "movq %0, %%rbx" : /* no outputs */ : "rm" (_RBX) : "%rbx" ) ;

   asm ( "movq %%rbx, %0" : "=rm" (_RBX) : /* no inputs */ ) ;
   sprintf( temp, "-> in counter (after changing rbx value): rbx = %08lx.\n\n", _RBX ) ;
   strprt( temp ) ;
   for ( long i = 0; i < count; i++ ) {
      coresume() ;
   }
   stop = true ;
}
```

You will note that `counter` changes the value in the rbx register, in order to test that the secor coroutine implementation preserves the value in the rbx register, as required by the x86_64 ABI.

`main`'s Take Down

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

Before exiting, `main` shows the contents of the rbx register. The value should be the same as it was prior to starting the coroutines.

```
<<take down>>=
asm ( "movq %%rbx, %0" : "=rm" (_RBX) : /* no inputs */ ) ;
sprintf( temp, "\n-> in main (after cobegin): rbx = %08lx.\n\n", _RBX ) ;
strprt( temp ) ;
return 0 ;
```

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 is just a single root in this literate program, the abc.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.

abc.cpp

```
<<abc.cpp>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   +abc+ -- a simple test program for coroutines.  <by Cary WR Campbell>
 *
 * Module:
 *   +abc+ executable for macOS or Windows.
 *--
 */
/*
 * Include files
 */
<<include files>>
/*
 * Definitions
 */
<<definitions>>
/*
 * Variables
 */
<<global variables>>
/*
 * Forward References
 */
<<forward references>>
/*
 * Main Routine
 */
<<main routine>>
/*
 * Coroutines
 */
<<coroutines>>
/*
 * Utility Routines
 */
<<utilities>>
```

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.
 */
```

Epilogue

This document has described the `abc` program, its source code, and its execution.

The `abc` program is most certainly a simple program, created as an initial test case for the Simple C/C++ Coroutines using Codecraft's open-source sccor library, running on macOS and Windows.

There are a couple of more interesting and ambitious coroutine test programs available as literate programs.

The `darts` Program

The `darts` program is an interactive visual test program for Simple C/C++ Coroutines using Codecraft's open-source sccor library. The `darts` program can demonstrate the simultaneous execution of hundreds of coroutines with elapsed times in the microseconds.

`darts` runs in a macOS or Windows (Cygwin) Terminal window as a command-line executable.

See [cc-darts-program] for the `darts` literate program.

See [cc-darts-mac] for the macOS `darts` executable that was tangled and compiled from the `darts` literate program.

See [cc-darts-win] for the Windows (Cygwin) `darts` executable that was tangled and compiled from the `darts` literate program.

The elevator Program

The elevator program is a visual and interactive test program that simulates a bank of elevators. It too is a Simple C/C++ Coroutines test program, one which provides a more extensive demonstration of the coroutines' capabilities and performance.

elevator runs in a macOS or Windows (Cygwin) Terminal window as a command-line executable.

See [cc-elevator-program] for the elevator literate program.

See [cc-elevator-mac] for the macOS elevator executable that was tangled and compiled from the elevator literate program.

See [cc-elevator-win] for the Windows (Cygwin) elevator executable that was tangled and compiled from the elevator literate program.

The sccor Library

I hope to have piqued your interest in the multi-platform Simple C/C++ Coroutines available with Codecraft's open-source sccor library.

The sccor library is currently available in macOS and Windows (Cygwin) versions.

See [cc-sccor-mac] for the macOS version.

See [cc-sccor-win] for the Windows (Cygwin) version.

References

Programs

[1] [cc-abc-mac] Cary WR Campbell, The abc Program Executable File—macOS Version, July 2021, https://drive.google.com/file/d/1tgGFy8SNFteQmVTQsem8LcdW8eirYPCf/view?usp=sharing.

[2] [cc-abc-win] Cary WR Campbell, The abc Program Executable File—Windows (Cygwin) Version, July 2021, https://drive.google.com/file/d/1O0sfpd1Q0fHdx1h8xwYYvGjTOOy0sVg5/view?usp=sharing.

[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-program] Cary WR Campbell, The elevator Literate Program, June 2021, https://drive.google.com/file/d/1AIKR9hAKQWzCLRD3tMeD2fknfdw_mq23/view?usp=sharing.

[7] [cc-elevator-mac] Cary WR Campbell, The elevator Program Executable File—macOS Version, June 2021, https://drive.google.com/file/d/1Zk0dyyKOYPEWoDlzvhlVwKkxyC1LrOyx/view?usp=sharing.

[8] [cc-elevator-win] Cary WR Campbell, The elevator Program Executable File—Windows (Cygwin) Version, June 2021, https://drive.google.com/file/d/1RB35oz9nrXgS2x_-vrd6zOPIxXCFaOYv/view?usp=sharing.

[9] [cc-sccor-mac] Cary WR Campbell, The sccor Library Executable File—macOS Version, March 2021, https://drive.google.com/file/d/1v2lkjS8B54feVQeK2uu38P2IwCU6T1lU/view?usp=sharing.

[10] [cc-sccor-win] Cary WR Campbell, The sccor Library Executable File—Windows (Cygwin) Version, July 2021, https://drive.google.com/file/d/1sMbcZihkmOvjrr7DTU8AmKUna1r7NpBA/view?usp=sharing.

Projects

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

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

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

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

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

Articles

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

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