```
//   Program:  stack_simulator.c
//   Author:   M. Q. Rieck
//   Last update:  November 3, 2017
//   Purpose:  Simulate a possible run-time stack in a computer system
//
//   Description:
//
//         User-defined functions (procedures) will call each other and
//      use the simulated run-time stack similar to the way things work
//      in a real computer system. Here though, as functions in this C
//      program, they should always be written to be of void return type,
//      and to have no parameters. However, via the simulator's utility
//      functions, and the simulated stack, such user-defined functions
//      can be made to simulate low-level code procedures that have one
//      or more integer parameters, and that are capable of returning an
//      integer value via a simulated CPU register (return_value_register).
//         When a procedure is "called" (in the sense of the simulator),
//      the "return address" for the call is pushed onto the stack, along
//      with an index into the stack, called a "frame index." This frame
//      index serves to reference the caller procedure's frame on the stack,
//      that is, the portion of the stack used for the caller's parameters
//      and local variables. The index into the stack where the caller's
//      frame index is being held becomes the frame index for the callee's
//      frame on the stack. The "return address" pushed onto the stack may
//      be phony (if USE_EXECINFO is undefined), and in any case, the
//      return address on the stack is not really used by the simulator.
//      This is because program flow is not really controlled by the
//      simulator. The host computer's actual stack is used for this.
//         The active frame index, i.e. the callee's frame index, is used
//      dynamically to help locate the active (callee) procedure's parameters
//      and local variables. The parameters are on the stack before the
//      return address for the call that caused the (callee) procedure to be
//      activated. After this comes the caller's frame index. The callee's
//      local variables would come after this. So the picture of the stack
//      would basically be as follows:
//
//        stack:  ... params  ret-addr  caller-frame-index  local-vars
//                                               ^
//                                (active frame index points here)
//
//      Here we are looking at the callee's frame, including the callee's
//      parameters and local variables. The caller's frame would be similar,
//      and would precede the callee's frame on the stack. And so forth.
//         Before a (caller) procedure calls a (callee) procedure, the caller
//      should push the arguments onto the stack. These automatically become
//      the values of the callee's parameters. The callee procedure should
//      begin its execution by reserving stack space for its local variables.
//      Since the parameters and local variables have known positions, rel-
//      ative to the callee's frame index, it becomes easy to access these
//      variables.
//         Before returning, a (callee) procedure should put the value (if any)
//      to be returned to the caller procedure, in the return-value register.
//      The return will cause the frame index to be restored to the value it
//      had before the call was made, and similarly, the stack will be re-
//      stored to its previous size. In an actual computer, the return address
//      that was on the stack would also be used to restore the program counter
//      to its previous value, so that program execution would return back to
//      the caller's code. However, in the simulator, program flow is really
//      controlled by the host computer's hardware and software, and so the
//      return address is safely ignored by the simulator. Still, it is good
//      to imagine it being popped off the stack and put into the program
//      counter.
```

```
////////////// THE RUN-TIME STACK AND SUPPORTING FUNCTIONS ///////////////

#define STACK_CAPACITY 1000   // max number of possible entries in stack
#define TOO_BIG 1000000       // helps distinguish data from addresses
//#define STEP_BY_STEP          // wait for user returns while processing
#define DIAGNOSE              // show stack behavior, etc., during execution
#define INIT_FRAME_INDEX -1   // frame index for main procedure
#define PHONY_ADDR 999999999  // phony return address value
//#define USE_EXECINFO          // when commented out, phony addresses are used

#include <stdio.h>
#include <stdlib.h>
#ifdef USE_EXECINFO
#include <execinfo.h>
#endif

int stack[STACK_CAPACITY];
int stack_size = 0;
int frame_index = INIT_FRAME_INDEX;
int call_level = 0;
int return_value_register;
int num_calls = 0;

void display_num_calls() {
    printf("-> number of procedure calls was %d.\n", num_calls);
}

// Display the value in the "return value register"
void display_return_value() {
    printf("-> return value: %d\n", return_value_register);
}

// Display the current contents of the stack
void display_stack() {
    printf("-> stack(size=%d, frame index=%d): ",
          stack_size, frame_index);
    for(int i=0; i<stack_size; i++)
#ifdef USE_EXECINFO
      if (abs(stack[i]) < TOO_BIG)
        printf("%d ", stack[i]);
      else
        printf("0x%x ", stack[i]);
#else
      printf("%d ", stack[i]);
#endif
    printf("\n");
}

// Simulate calling a procedure.
// Though "call" is a C function here, pretend that it is an
// assembly language (or machine language) instruction instead.
// For convenience, this function is also pushing the frame index
// value onto the stack, and changing the frame index, though this
// would actually be the job of the callee procedure.
void call(void g()) {
#ifdef USE_EXECINFO
    // When f uses this to call g, the return address back to f is this:
    int ret_addr = (int)(__builtin_return_address(0));
#else
    int ret_addr = PHONY_ADDR;           // phony return address
#endif
    stack[stack_size++] = ret_addr;      // Push return address
    stack[stack_size++] = frame_index;   // Push old frame index
    frame_index = stack_size - 1;        // Set frame pointer to point to
#ifdef DIAGNOSE                          //    old frame pointer value
#ifdef USE_EXECINFO
```

```c
    printf("-> calling a procedure at address 0x%x (level %d)\n",
        (int)g, ++call_level);
#else
    printf("-> calling a procedure (level %d)\n", ++call_level);
#endif
    printf("-> pushing frame index, and then changing frame index\n");
    display_stack();
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
    num_calls++;
    g();                                 // Now, call the callee procedure
}

// Simulate returning from a procedure.
// Though "ret" is a C function here, pretend that it is an
// assembly language (or machine language) instruction instead.
// Also popping off the old value of the frame index, though this
// would really be the job of the callee procedure.
void ret() {
    stack_size = frame_index - 1;        // End of stack must be as it was
    int ret_addr = stack[stack_size];    // Return address is now after stack
    frame_index = stack[frame_index];    // Frame index must be as it was
#ifdef DIAGNOSE
    printf("-> popping off old value for frame index (restoring it)\n");
#ifdef USE_EXECINFO
    printf("-> returning from a call, back to 0x%x (level %d)\n",
        ret_addr, call_level--);
#else
    printf("-> returning from a call (level %d)\n", call_level--);
#endif
    display_return_value();
    display_stack();
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
}

// Push a value onto the stack
// Though "push" is a C function here, pretend that it is an assembly
// language (or machine language) instruction instead
void push(int x) {
    stack[stack_size++] = x;
#ifdef DIAGNOSE
    printf("-> pushing %d onto stack\n", x);
    display_stack();
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
}

// Pop a value off of the stack
// Though "pop" is a C function here, pretend that it is an
// assembly language (or machine language) instruction instead
int pop() {
    int x = stack[--stack_size];
#ifdef DIAGNOSE
    printf("-> popping %d off of stack\n", x);
    display_stack();
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
```

```
    return x;
}

// Reserve stack space for several (n) integers
// (These spots might not be zeroed out)
// Though "reserve" is a C function here, pretend that it is an
// assembly language (or machine language) instruction instead
void reserve(int n) {
    stack_size += n;
#ifdef DIAGNOSE
    printf("-> reserving %d spot(s) on stack\n", n);
    display_stack();
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
}

// Utility function to get ordinal number suffix
char* get_ordinal_suffix(int x) {
    if (x < 0) x = -x;
    int m = x%10;      // one's place
    int n = x/10%10;   // ten's place
    if (n == 1) return "th";
    if (m == 1) return "st";
    if (m == 2) return "nd";
    if (m == 3) return "rd";
    return "th";
}

// Get the value of a local variable
// (   get_local(1) gets the value of the first local variable,
//     get_local(2) gets the value of the second local variable,
//     etc.   )
int get_local(int n) {
#ifdef DIAGNOSE
    printf("-> getting %d%s local variable value (= %d)\n",
        n, get_ordinal_suffix(n), stack[frame_index + n]);
#endif
    return stack[frame_index + n];
}

// Set the value of a local variable
void set_local(int n, int x) {
    stack[frame_index + n] = x;
#ifdef DIAGNOSE
    printf("-> setting %d%s local variable to %d\n", n, get_ordinal_suffix(n), x);
    display_stack();
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
}

// Get the value of a parameter
// (   get_param(-1) gets the value of the last parameter,
//     get_param(-2) gets the value of the next-to-last parameter,
//     etc.   )
int get_param(int n) {
#ifdef DIAGNOSE
    printf("-> getting %d%s parameter value (= %d)\n",
        n, get_ordinal_suffix(n), stack[frame_index - 1 + n]);
#endif
    return stack[frame_index - 1 + n];
}
```

```
// Set the value of a parameter (not usually needed)
void set_param(int n, int x) {
    stack[frame_index - 1 + n] = x;
#ifdef DIAGNOSE
    printf("-> setting %d%s parameter to %d\n", n, get_ordinal_suffix(n), x);
    display_stack();
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
}

// Get the contents of the return-value register
int get_ret_value_reg() {
#ifdef DIAGNOSE
    printf("-> getting contents of return-value register (= %d)\n",
        return_value_register);
#endif
    return return_value_register;
}

// Set the contents of the return-value register
void set_ret_value_reg(int x) {
    return_value_register  = x;
#ifdef DIAGNOSE
    printf("-> loading return-value register with the value %d\n",
        return_value_register);
#endif
#ifdef STEP_BY_STEP
    getchar();
#endif
}


///////////////////////// SAMPLE USAGE ////////////////////////

// These are C functions that are to be regarded as representing
// low-level-code procedures being executed in the CPU/stack simulator:
void proc1();
void proc2();
void proc3();
void proc4();
void factorial();
void fibo();
void fibo_helper();
void fibo_horror();

// Get an integer from the user
int get_int() {
    int x;
    scanf("%d", &x);
    return x;
}

// The function main also represents a procedure being used in the
// CPU/stack simulator, but the actual variables here (x,y,n,c)
// are here solely to communicate with the user, and should not
// be regarded as part of the simulated procedure. Ditto for the
// actual variables used in the other simulated procedures.
int main() {
    int x, y, n;
    char c;
#ifdef USE_EXECINFO
    printf("Starting in main function at 0x%x\n\n", (int)main);
#else
    printf("Starting in main function\n");
```

```
#endif

// *** Here is the first test (comment it out if not wanted) ***
/*
    printf("Doing the first test:\n\n");
    printf("The procedure main pushes a number (supplied by the user) onto the stack,
and\n");
    printf("then calls the procedure proc1. The call causes the return address (i.e.
the \n");
    printf("value of the program counter register) to be pushed onto the stack.
Execution\n");
    printf("now proceeds to the start of the procedure proc1, which treats the pushed
number\n");
    printf("as an argument being passed to it, and treats this spot on the stack as a
parameter.\n");
    printf("proc1 pushes the value of the frame pointer register onto the stack, and
copies\n");
    printf("the value of the stack pointer register into the frame pointer register.
(However\n");
    printf("in my demo, instead of using pointers, i.e. memory addresses, I am using
indexes\n");
    printf("into the stack, treating it as an array, in order to make the information
more\n");
    printf("intelligible to us). proc1 reserves more space on the stack, by extending
the stack,\n");
    printf("and uses this for a local variable. Using several machine language
instructions,\n");
    printf("it now multiplies the value of its parameter (i.e. the argument that was
passed)\n");
    printf("by itself, thereby squaring this value, and puts the result in its local
variable.\n");
    printf("It then gets ready to return, by copying the value in its local variable
into the\n");
    printf("register designated for returning the return value from a procedure call.
The\n");
    printf("stack is then restored to its condition prior to reserving the extra room
for the\n");
    printf("local variable. The old frame pointer value is popped off the stack and
put in the\n");
    printf("frame pointer register. A ret instruction is executed, which pops the
return value\n");
    printf("off the stack, and puts this in the program counter register. Execution
continues\n");
    printf("back in the procedure main, which pops the two arguments off the stack,
and then\n");
    printf("displays the value of the return-value register.\n\n");
    printf("x = ");
    x = get_int();
    push(x);          // Push an argument onto the stack
    call(proc1);      // Call the procedure proc1
                      // After returning, value of x is still on the stack
    pop();            // Pop the argument off the stack
    y = get_ret_value_reg();
    printf("The square of %d is %d.\n\n", x, y);
*/
// *** Here is the second test (comment it out if not wanted) ***
/*
    num_calls = 0;    // Prepare to count the number of calls;
    printf("Doing the second test:\n\n");
    printf("This time, main pushes two arguments onto the stack, and\n");
    printf("then calls proc2, which uses two parameters and two local\n");
    printf("variables (after reserving stack space). Some unimportant\n");
    printf("computations are performed. What is important is that to do\n");
    printf("these, I have proc2 calling proc3, and proc3 calling proc4.\n");
    printf("proc4 does some pointless things, and then just returns the\n");
    printf("same number that was passed to it! proc3 receives two argu-\n");
```

```
    printf("ments, and in a somewhat weird way, multiplies these two\n");
    printf("numbers, and returns their product. The quantity that proc2 \n");
    printf("ends up computing is 2*x*y + 3*x + y + 1, which is returned\n");
    printf("back to main. The point here is to watch the stack get a \n");
    printf("good workout, and ultimately get restored. \n\n");
    printf("x = ");
    x = get_int();
    push(x);
    printf("y = ");
    y = get_int();
    push(y);          // Push two arguments onto the stack
    call(proc2);      // Call the procedure proc2
    pop();            // Pop the two arguments off of the stack
    pop();
    y = get_ret_value_reg();
    display_num_calls();
    printf("The answer is %d.\n\n", y);
*/
// *** Here is the third test (comment it out if not wanted) ***
/*
    num_calls = 0;    // Prepare to count the number of calls;
    printf("Doing the third test - factorial.\n");
    printf("Calls a recursive function to compute n!.\n\n");
    printf("What number? ");
    x = get_int();
    push(x);           // Push argument onto the stack
    call(factorial);
    pop();             // Pop argument off of the stack
    y = get_ret_value_reg();
    display_num_calls();
    printf("%d factorial equals %d\n\n", x, y);
*/
// *** Here is the fourth test (comment it out if not wanted) ***

    num_calls = 0;    // Prepare to count the number of calls;
    printf("Doing the fourth test - fibo_horror\n");
    printf("Calls a horribly recursive function to compute a Fibonacci number.\n\n");
    printf("Which Fibonacci number? ");
    x = get_int();
    push(x);           // Push argument onto the stack
    call(fibo_horror); // Call fibo_horror to compute a Fibonacci number
    pop();             // Pop argument off of the stack
    y = get_ret_value_reg();
    display_num_calls();
    printf("%d%s Fibonacci number is %d\n\n", x, get_ordinal_suffix(x), y);

// *** Here is the fifth test (comment it out if not wanted) ***
/*
    num_calls = 0;    // Prepare to count the number of calls;
    printf("Doing the fifth test - fibo\n");
    printf("Calls a nice recursive function to compute a Fibonacci number.\n\n");
    printf("Which Fibonacci number? ");
    x = get_int();
    push(x);          // Push argument onto the stack
    call(fibo);       // Call fibo to compute a Fibonacci number
    pop();            // Pop argument off of the stack
    y = get_ret_value_reg();
    display_num_calls();
    printf("%d%s Fibonacci number is %d\n\n", x, get_ordinal_suffix(n), y);
*/
}

// Simultates this function:
// int proc1(int x) {
//    int y;
//    y = x*x;
```

```
//    return y;
// }
void proc1() {
    // Reserve stack space for one local variable
    reserve(1);
    // Now, get the passed argument, multiply it by itself, and store
    // this in a local variable
    set_local(1, get_param(-1)*get_param(-1));
    // Copy the value of the local variable into the return-value register
    set_ret_value_reg(get_local(1));
    // Execute a ret instruction to return
    ret();
}

// Procedure to compute something, just to demonstrate stuff
//
// Receives: x and y (two integers)
// Returns:  2*x*y + 3*x + y + 1
//
// Plays with the stack just for the heck of it
void proc2() {
    reserve(2);    // Reserve stack space for two local variables
    push(get_param(-2));
    push(get_param(-1));
    call(proc3);  // Use proc3 to compute x*y
    push(2);
    push(get_ret_value_reg());
    call(proc3);  // Use proc3 to compute 2*(x*y)
    set_local(1, return_value_register);   // Save in 1st local var
    push(3);
    push(get_param(-2));
    call(proc3);  // Use proc3 to compute 3*x
    set_local(2, return_value_register);   // Save in 2nd local var
    // Add things up, and put the result in the return-value register
    set_ret_value_reg(get_local(1) + get_local(2) + get_param(-1) + 1);
    ret();
}

// Overly complicated procedure to multiply two numbers
//
// Receives: two integers
// Returns:  their product
//
void proc3() {
    reserve(1);     // Use a local variable, to hold the
                    //   product of the parameters, just
                    //   to unnecessarily complicate things
    set_local(1, get_param(-2) * get_param(-1));
    push(get_local(1));  // Pushes the local variable onto the
    push(get_local(1));  //    stack twice, for no good reason
    pop();               // Pop off the stack once
    call(proc4);    // Pass the number to proc4()
    ret();
}

// An utterly pointless procedure
//
// Receives: an integer
// Returns:  same integer
//
void proc4() {
    reserve(3);    // Use local variables to pass the number around
    set_local(1, get_param(-1));
    push(get_local(1));
    set_local(2, pop());
    set_local(3, get_local(2));
```

```
    push(get_local(3));
    push(get_local(2));
    push(get_local(1));
    pop();
    pop();
    // Ultimately just set return_value register to the received argument!
    set_ret_value_reg(pop());
    ret();
}

// Procedure to compute a factorial, recursively
//
// Receives: n (an integer less than 47)
// Returns:  n factorial (n!)
//
// Basically implements this function:
// int factorial(int x) {
//    if (x == 0) return 1
//    else return factorial(x-1) * x;
// }
void factorial() {
    if (get_param(-1) == 0) {
        return_value_register = 1;
        ret();
    } else {
        push(get_param(-1) - 1);
        call(factorial);
        pop();
        set_ret_value_reg(get_ret_value_reg() * get_param(-1));
        ret();
    }
}

// Procedure to compute a Fibonacci number, but using the full-blown
// recursive method that is usually taught, and that is horribly
// inefficient
//
// Receives: n (an integer less than 47)
// Returns:  n-th Fibonacci number
//
// Basically implements this function:
// int fibo_horror(int x) {
//    int y;
//    if (x == 0) return 0;
//    if (x == 1) return 1;
//    y = fibo_horror(x-2);
//    return fibo_horror(x-1) * y;
// }
void fibo_horror() {
    reserve(1);        // Reserve stack space for one local variable
    if (get_param(-1) == 0) {
        set_ret_value_reg(0);
        ret();
    } else
    if (get_param(-1) == 1) {
        set_ret_value_reg(1);
        ret();
    } else {
        push(get_param(-1) - 2);    // Compute (n-2)-th Fibo number
        call(fibo_horror);
        set_local(1, get_ret_value_reg());  // Hold it in local var
        push(get_param(-1) - 1);    // Compute (n-1)-th Fibo number
        call(fibo_horror);
        set_ret_value_reg(get_ret_value_reg() + get_local(1));
        ret();
    }
```

```
}

// Procedure to compute a Fibonacci number, by using a tail-recursive
// helper procedure
//
// Receives: n (an integer less than 47)
// Returns:  n-th Fibonacci number
//
// Basically implements this function:
// int fibo(int x) {
//     if (x == 0) return 0;
//     else if (x == 1) return 1;
//     else return fibo_helper(0, 1, x-1);
// }
void fibo() {
    if (get_param(-1) == 0) {
        set_ret_value_reg(0);
        ret();
    } else
    if (get_param(-1) == 1) {
        set_ret_value_reg(1);
        ret();
    } else {
        push(0);
        push(1);
        push(get_param(-1) - 1);
        call(fibo_helper);
        ret();
    }
}

// Tail-recursive helper procedure to help compute Fibonacci number
//
// Receives: two successive fibo numbers, and how much further to go
// Returns:  a Fibonacci number
//
// Basically implements this function:
// int fibo_helper(int x, int y, int n) {
//    if (n == 0) return y;
//    else return fibo_helper(y, x+y, n-1);
// }
void fibo_helper() {
    if (get_param(-1) == 0) {
        // If no further, then return the second passed fibo number
        set_ret_value_reg(get_param(-2));
        ret();
    } else {
        // Call recursively, passing next pair of successive Fibo
        // numbers, and how much further to go from there
        push(get_param(-2));
        push(get_param(-3) + get_param(-2));
        push(get_param(-1) - 1);
        call(fibo_helper);
        ret();
    }
}
```