CS 2110 Homework 07:
Assembly Subroutines and Recursion

Michael "The Machine" Lin, Austin Adams, Daniel Becker,
Preston Olds, Brandon "The Machine" Whitehead, and Cem Gokmen

Spring 2018

Contents

1 Overview

The goal of this assignment is to familiarize you with the LC-3 calling convention using assembly recursion. This will involve use of the stack, as seen in lecture, to save the return address (RA) and old frame pointer (OFP).

In each section, you'll act as a compiler, converting the provided pseudocode into assembly code that follows the LC-3 calling convention. (For more information on the calling convention, see the lecture slides or the Calling Convention Guide later in this PDF.) We reserve the right to use different tests for grading, but to check your work, we provide tests which you can run as follows:

```
$ lc3test testfile.xml asmfile.asm
```

Try to follow the pseudocode as closely as you can, because the tests expect you to. In particular, the order of arguments and your strategy for recursion need to match the pseudocode, since the tests are hardcoded for them.

1.1 Resources

To tackle this homework, we've provided:

- Lecture slides and example code on Canvas for the LC-3 calling convention under Files→Slides: `L14d_There is nothing magical about recursion.pptx` and `fact.asm`.
- An LC-3 calling convention guide written by Kyle Murray (RIP) later in this document
- An implementation of a recursive multiply subroutine, `mult()`, in `shift_pow.asm`

Feel free to use code from these class resources as you need to, but as always, not from your friends or random sketchy internet sites.

1.2 Debugging

To debug failing tests, open your .asm file in complx and use Debug → Setup Test to set up the same environment as lc3test does when running the failing test. This way, you can step through instruction by instruction and see what goes wrong instead of running the test repeatedly hoping for a new result. If you want to try a method call not covered by a test, use Debug → Simulate Subroutine Call.

To see the stack, click the main complx window, press Control-V to open a new memory view, and press Control-G and enter `xF000` to see the stack. (Remember, the stack grows into lower memory, so values closer to the top of the window, away from `xF000`, are higher on the stack.)

We the TAs ask that before you show us your code saying "the tests don't pass," you try to debug it in complx as described above. As students ourselves with our own fires to put out, we don't have time to debug everyone's code.

2 Subroutines

2.1 Right Bit Shift

The LC-3 lacks a right bit shift instruction, so in this first section, we'll roll our own in `shift_pow.asm`.

The idea is to 'rotate' bits around and mask until we get the result we want. Define 'rotating' as shifting a binary number B left 1 bit and then setting the new least significant bit of B to the old most significant bit of B before the shift. So, for example, if we rotate a 4-bit binary number $abcd$, where a, b, c, d are bits, then we get $bcda$. Notice that if we perform this 16 times for a 16-bit number, we loop around and get the original number back again. And if we 'rotate' only 15 times for a 16 bit number, then we've bitshifted the number right 1 bit, with the most significant bit being the old least significant bit.

So, using this knowledge, if we want to right shift a 16-bit bitstring $b_{15}b_{14}\cdots b_0$ (where b_i are bits) right to get $0b_{15}b_{14}\cdots b_1$, we would 'rotate' the string 15 times: $b_{15}b_{14}\cdots b_0 \rightarrow b_{14}b_{13}\cdots b_0b_{15} \rightarrow b_{13}b_{12}\cdots b_0b_{15}b_{14} \rightarrow \cdots \rightarrow b_0b_{15}\cdots b_1$ then clear out the most significant bit (MSB): $0b_{15}\cdots b_0$. To optionally perform sign extension, we can bitwise OR in the original MSB b_{15} to get: $b_{15}b_{15}b_{14}\cdots b_1$.

2.1.1 rotate_bits()

First, we'll implement a function which 'rotates' a 16-bit LC-3 integer n; that is, it will bit shift n left 1 bit and then OR in the value of the orignal most significant bit as the new least significant bit.

```
// Perform a 'bit rotation' (abcd -> bcda) on the number n
// k times and return the result
rotate_bits(n, k) {
    if k <= 0 {
        return n
    } else {
        msb = n & x8000
        n = n + n       // n = n << 1
        if msb != 0 {
            n = n + 1
        }
        return rotate_bits(n, k-1)
    }
}
```

2.1.2 right_shift_once()

Now that we can 'rotate' bits easily, write a method which shifts a 16-bit LC-3 integer n to the right one bit, performing sign extension if `sext` $\neq 0$.

```
right_shift_once(n, sext) {
    msb = n & x8000               // back up old MSB
    n = rotate_bits(n, 16-1)
    n = n & x7FFF                 // clear out MSB
    if sext != 0 {                // set MSB to old MSB if we're sign extending
        n = n + msb
    }
    return n
}
```

2.1.3 right_shift()

With `right_shift_once()` finished, we can now implement a function which right shifts a 16-bit LC-3 integer n right k bits, performing sign extension if `sext` $\neq 0$:

```
right_shift(n, k, sext) {
    if k <= 0 {
        return n
    } else {
        shifted_once = right_shift_once(n, sext)
        return right_shift(shifted_once, k-1, sext)
    }
}
```

2.2 Fast Exponentiation

We will implement the following $O(\log k)$ algorithm which calculates n^k where $k \geq 0$:

$$\text{pow}(n,k) = \begin{cases} \text{pow}(n, k/2)^2 & k \text{ is even} \\ n \cdot \text{pow}(n, \lfloor k/2 \rfloor)^2 & k \text{ is odd} \end{cases}$$

$$\text{pow}(n, 0) = 1$$

Use the following pseudocode to finish the `pow` subroutine in `shift_pow.asm` (assume $k \geq 0$). We've provided `mult(a,b)`, a subroutine which multiplies `a` and `b`.

```
pow(n,k) {
    if k == 0 {
        return 1
    }

    k_halved = right_shift(k, 1, 1)  // k_halved = k >> 2
    halfpow = pow(n, k_halved)       // find n^(k/2)
    product = mult(halfpow, halfpow) // find n^(k/2) * n^(k/2)
    if (k & 1 != 0) {                // if k is odd
        product = mult(product, n)   // find n^(k/2) * n^(k/2) * n
    }
    return product
}
```

2.3 Recursive Bubble Sort

If you've taken CS 1332, then you are probably familiar with Bubble Sort (don't worry if you haven't — it is pretty simple). Essentially, the idea is to iterate over an array n times, each time pushing the largest element to its correct position. Most implementations of Bubble Sort are done iteratively, where you have two nested loops, but that's just boring. Instead, we are going to do it recursively! In each recursive call, you are going to iterate over the array and push the largest element to the very end. Once you do that, you will recursively call the bubble sort method where you will decrease the size by one. Also, you will have to keep track of the number of swaps that you make and return the grand total. Here is the pseudocode:

```
bubble_sort(array, n) {
    if (n <= 1) {
     return 0
    }

    swapCount = 0

    for (i = 0; i < n-1; i++) {
     if (array[i] > array[i + 1]) {
         temp = array[i];
            array[i] = array[i + 1]
            array[i + 1] = temp
            swapCount++
        }
    }

    return swapCount + bubble_sort(array, n-1)
}
```

3 Rubric

1. `rotate_bits()` – 10%
2. `right_shift_once()` – 10%
3. `right_shift()` – 10%
4. `pow()` – 30%
5. `bubble_sort()` – 40%

4 Deliverables

Please upload the following files to Gradescope under `Homework 7`:

1. `shift_pow.asm`
2. `bubble_sort.asm`

If you are in section A6 or B6, please submit to Autolab instead. We'll send out an announcement with details.

Download and test your submission to make sure you submitted the right files.

5 LC-3 Assembly Programming Requirements

5.1 Overview

1. Your code must assemble with **NO WARNINGS OR ERRORS**. To assemble your program, open the file with Complx. It will complain if there are any issues. **If your code does not assemble you WILL get a zero for that file.**
2. **Comment your code!** This is especially important in assembly, because it's much harder to interpret what is happening later, and you'll be glad you left yourself notes on what certain instructions are contributing to the code. Comment things like what registers are being used for and what less intuitive lines of code are actually doing. To comment code in LC-3 assembly just type a semicolon (;), and the rest of that line will be a comment.
3. Avoid stating the obvious in your comments, it doesn't help in understanding what the code is doing.

   Good Comment

   ```
   ADD R3, R3, -1          ; counter--
   BRp LOOP                ; if counter == 0 don't loop again
   ```

   Bad Comment

   ```
   ADD R3, R3, -1          ; Decrement R3
   BRp LOOP                ; Branch to LOOP if positive
   ```

4. **DO NOT assume that ANYTHING in the LC-3 is already zero.** Treat the machine as if your program was loaded into a machine with random values stored in the memory and register file.

5. Following from 3. You can randomize the memory and load your program by doing File - Randomize and Load.

6. Use the LC-3 calling convention. This means that all local variables, frame pointer, etc... must be pushed onto the stack. Our autograder will be checking for correct stack setup.

7. Start the stack at xF000. **The stack pointer always points to the last used stack location.** This means you will allocate space **first**, then store onto the stack pointer.

8. Do NOT execute any data as if it were an instruction (meaning you should put .fills after **HALT** or RET).

9. Do not add any comments beginning with @plugin or change any comments of this kind.

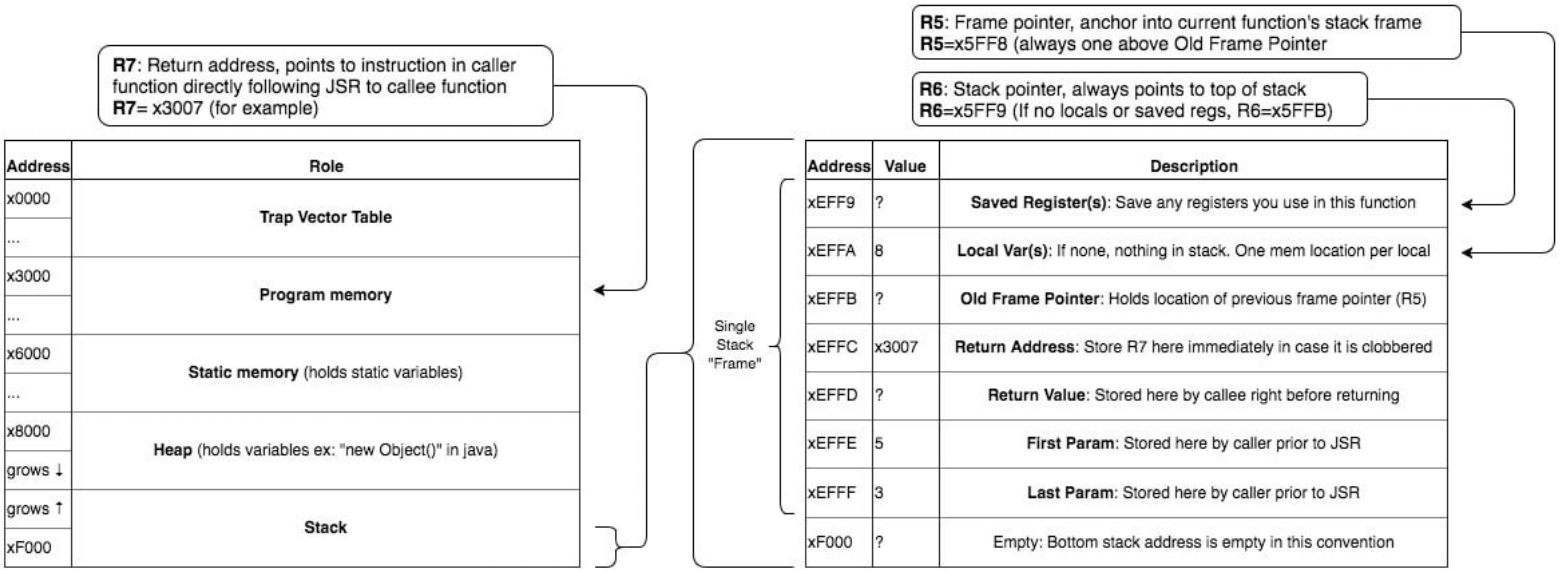
10. **Test your assembly.** Don't just assume it works and turn it in.

5.2 LC-3 Calling Convention Guide

A handy assembly guide written by Kyle Murray (RIP) follows on the next page:

Overview

On the left is a grand-scale layout of memory in a computer. On the right is a zoomed in view of the stack. The state of the stack is a snapshot during the execution of some function (we will call it "foo"). Since the stack begins at xF000, we know that we are currently executing the first function called by main (main itself does not have its own stack frame). At this state in time, the main function is the "caller", and the currently executing function, foo, is the "callee". Appropriate example addresses and values are included throughout this guide, but note that these values are not necessarily the standard.



Walkthrough

In the following walkthrough, function "foo" will call function "bar", and you will be shown snapshots of the stack at different stages of the calling convention. Note that "caller/foo" and "callee/bar" will be used interchangeably. For transitions 2, 3, and 4 (not 1, 2.1, or 3.0), the instructions can be EXACTLY the between different functions, a "cookie cutter" set of instructions. The following pseudocode describes the two functions (they don't do anything important, they are just examples of functions you might have to recreate in assembly):

```
int foo (int a, int b) {
    int local_var = a + b;
    answer = bar(a, local_var);
    return answer;
}
```

```
int bar (int c, int d) {
    int var1 = c + d;
    int var2 = c - d;
    answer = var1 / var2;
    return answer;
}
```

State 0 - During Foo's Execution

```
int foo (int a, int b) {
    int local_var = a + b;
    answer = bar(a, local_var);   <-- PC
    return answer;
}
```

The function has stored local_var but not yet called bar. Foo is about to call bar so we will now refer to foo as the "caller" and bar as the "callee". Remember the PC points to the *planned-to-be-next instruction.*

Address	Value	Description	
xEFF9	?	**Foo's Saved Register(s)**: Foo saved all registers it used before use	R6
xEFFA	8	**Foo's Local Var(s)**: stores value of local_var	R5
xEFFB	?	**Foo's Old Frame Pointer**: Foo's caller is main so this is garbage	
xEFFC	x3007	**Foo's Return Address**: Points to instruction in main to return to	
xEFFD	?	**Foo's Return Value**: currently unknown	
xEFFE	5	**Foo's First Param**: a	
xEFFF	3	**Foo's Last Param**: b	
xF000	?	**Empty** (main's frame)	

Caller Setup (Transition 1)

Before calling JSR BAR, foo must initiate the creation of bar's stack frame by giving bar its params and moving the stack pointer (R6) accordingly. That's it.

Tasks

Tasks directly translate to assembly instructions

Move stack pointer up enough spaces to make room for params.
Store params on stack.
JSR to BAR.

State 1 - Foo to Bar handoff

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}
int bar (int c, int d) {
        int var1 = c + d;
        int var2 = c - d;
        answer = var1 / var2;
        return answer;
}
```

PC before JSR
R7 after JSR

PC after JSR

This is the snapshot of the stack right as JSR is called. Notice the callee's (bar's) frame has been started but is incomplete. Think of this stage as the handoff between caller and callee. Immediately after JSR is called, R7 has the value of the return address and our program is now executing in bar.

	Address	Value	Description	
Callee's (bar's) stack frame	xEFF7	5	**Callee's First param:** c	← R6
	xEFF8	8	**Callee's Last param:** d	
Caller's (foo's) stack frame	xEFF9	?	**Caller's Saved Register(s):** Foo saved any registers before use	
	xEFFA	8	**Caller's Local Var(s)**	← R5
	xEFFB	?	**Caller's Old Frame Pointer**: Foo's caller is main so this is garbage	
	xEFFC	x3007	**Caller's Return Address**: Points to main	
	xEFFD	?	**Caller's Return Value**: currently unknown	
	xEFFE	5	**Caller's First Param**: a	
	xEFFF	3	**Caller's Last Param**: b	
	xF000	?	**Empty** (main's frame)	

Callee Setup (Transition 2)

As soon as our new function (bar) is called, we have to finish creating the stack frame and store our important values: return address (R7) and old frame pointer (R5). You should use R6 as your reference point.

Tasks

Move stack pointer up to make room for everything.
Store current value of R5 (caller's frame pointer) onto the stack.
Store the return address (currently stored in R7) onto the stack.
Update R5 to point to one above the position of the old frame pointer in the current frame.

Callee Setup cont (Transition 2.1)

We must now make room for local variables and store the values of any registers you plan on using on the stack. They will be restored later, immediately prior to returning to the caller (foo).

Tasks

Move stack pointer up enough to make room for locals and saved registers.
Store registers you plan to use onto stack.
Store value of local variables whenever they are calculated (not part of this transition)

State 2 - During Bar's Execution

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}

int bar (int c, int d) {
        int var1 = c + d;
        int var2 = c - d;
        answer = var1 / var2;
        return answer;
}
```

Return address

PC

This is the snapshot of the stack once bar has finished setting up its stack frame. It may now execute all of its logic freely, making sure to only use registers R0-R4 if they have been saved on the stack (in this example just R0 and R1). Notice that this state is similar to State 0, except it is for bar instead of foo. Also note that if bar decided to call a function (eg "baz"), bar would become the caller in that relationship and baz would be the callee.

	Address	Value	Description	
Callee's (bar's) stack frame	xEFF0	ex: 6	**Bar's Saved Register:** R0	R6
	xEFF1	ex: 7	**Bar's Saved Register:** R1	
	xEFF2	13	**Bar's Last Local Var**: stores value of var2	
	xEFF3	-3	**Bar's First Local Var**: stores value of var1	R5
	xEFF4	**x5FFA**	**Bar's Old Frame Pointer**: points to Foo's old FP/R5	
	xEFF5	**See Left**	**Bar's Return Address**: Points to instruction in foo to return to	
	xEFF6	?	**Bar's Return Value**: currently unknown	
	xEFF7	5	**Bar's First Param**: c	
	xEFF8	8	**Bar's Last Param**: d	
Caller's (foo's) stack frame	xEFF9	?	**Caller's Saved Register(s):** Foo saved any registers before use	
	xEFFA	8	**Caller's Local Var(s)**: int local_var	
	xEFFB	?	**Caller's Old Frame Pointer**: Foo's caller is main so this is garbage	
	xEFFC	x3007	**Caller's Return Address**: Points to main	
	...	...	...	

Callee Teardown (Transition 3.0)

Once we have our return value and are ready to return to the caller, we must first store the return value on the stack, then restore any registers we used. You should use R5 as your reference point

Tasks

Store return value onto stack
Restore any registers (R0-R4) you used from stack. In bar's case we restore R0 and R1

Callee Teardown (Transition 3)

Now for the cookie cutter instructions: Restore R5 (caller's frame pointer), restore R7 (our return address into the caller), pop down R6 to point to the return value (making it easy for the caller to find it), and return.

Tasks

Restore R5 (frame pointer) from the stack
Restore R7 (return address) from the stack
Pop down R6 (stack pointer) to point to the return value
Return

State 3 - Bar hands back to Foo

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}
int bar (int c, int d) {
        int var1 = c + d;
        int var2 = c - d;
        answer = var1 / var2;
        return answer;
}
```

R7 / PC after RET

PC during RET

Once bar is finished running and properly torn its stack down to the return value, we are ready to return. Upon calling "RET", the stack looks as it does on the right, the PC is updated with the return value (R7), and the caller, foo, picks it up from there. Bar points R6 at the return value so that foo can load the value immediately using R6.

	Address	Value	Description	
Callee's (bar's) stack frame	xEFF6	-4	**Callee's Return Value**	← R6
	xEFF7	5	**Callee's First param:** c	
	xEFF8	8	**Callee's Last param:** d	
Caller's (foo's) stack frame	xEFF9	?	**Caller's Saved Register(s):** Foo saved any registers before use	
	xEFFA	8	**Caller's Local Var(s)**	← R5
	xEFFB	?	**Caller's Old Frame Pointer**: Foo's caller is main so this is garbage	
	xEFFC	x3007	**Caller's Return Address**: Points to main	
	xEFFD	?	**Caller's Return Value**: currently unknown	
	xEFFE	5	**Caller's First Param**: a	
	xEFFF	3	**Caller's Last Param**: b	
	xF000	?	**Empty** (main's frame)	

Caller Accepts Return (Transition 4)

All we have to do is load the return value into a register so we can work with it, and then pop the stack pointer back down to the top of our stack.

Tasks

Load return value into a register we want it in
Pop down R6 to top of caller's frame

State 4/0 - During Foo's Execution

```
int foo (int a, int b) {
        int local_var = a + b;
        answer = bar(a, local_var);
        return answer;
}
```

PC

We are now back to where we started! Woo, what a lot of work. Just remember that once you figure out the instructions for transitions 2-4, they can be used as cookie cutters for every function call you ever make. Transitions 1, 2.1, and 3.0 will vary depending on the function (number of params, locals, and regs used). Note: If we wanted to continue, returning out of foo and back to main would look just like returning from bar to foo. Just treat foo as the callee of main.

Address	Value	Description	
xEFF9	?	**Foo's Saved Register(s)**: Foo saved all registers it used before use	← R6
xEFFA	8	**Foo's Local Var(s)**: stores value of local_var	← R5
xEFFB	?	**Foo's Old Frame Pointer**: Foo's caller is main so this is garbage	
xEFFC	x3007	**Foo's Return Address**: Points to instruction in main to return to	
xEFFD	?	**Foo's Return Value**: currently unknown	
xEFFE	5	**Foo's First Param**: a	
xEFFF	3	**Foo's Last Param**: b	
xF000	?	**Empty** (main's frame)	

6 Rules and Regulations

6.1 General Rules

1. Starting with the assembly homeworks, Any code you write (if any) must be clearly commented and the comments must be meaningful. You should comment your code in terms of the algorithm you are implementing we all know what the line of code does.

2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.

3. Please read the assignment in its entirety before asking questions.

4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

6.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to Canvas or you may submit an archive (zip or tar.gz only please) of the files (preferred). You can create an archive by right clicking on files and selecting the appropriate compress option on your system.

3. If you choose to submit an archive please don't zip up a folder with the files, only submit an archive of the files we want (see Deliverables).

4. Do not submit compiled files that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

5. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

6.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; neither Canvas, nor your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

6.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

6.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, as well as help each other debug code. What you shouldn't be doing, however, is paired programming where you collaborate with each other on a low level. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, and it is often the case that the recipient will simply modify the code and submit it as their own.

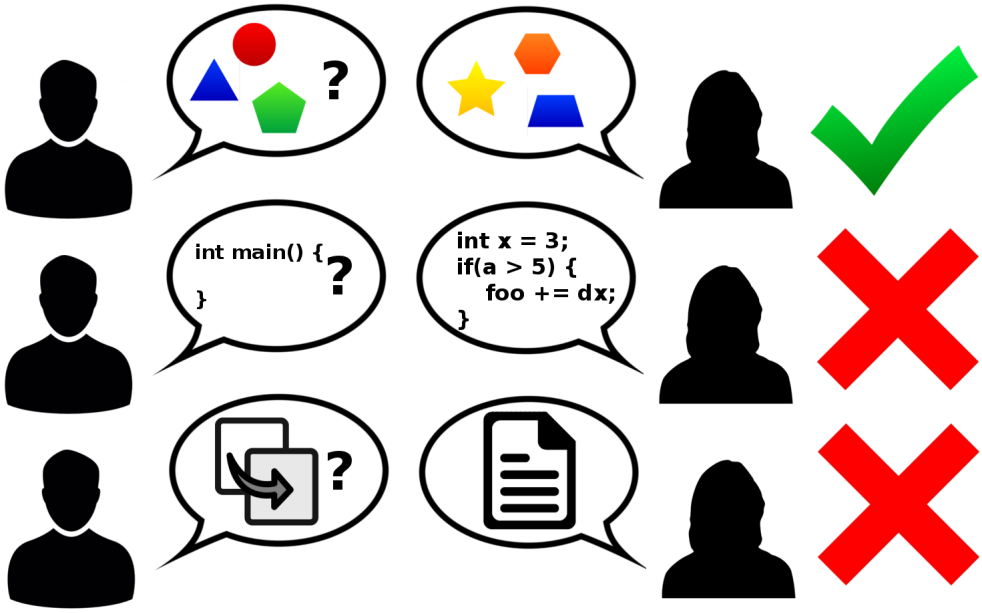


Figure 1: Collaboration rules, explained