ECE 190

Introduction to Computing Systems

Lab Manual

University of Illinois at Urbana-Champaign

Contents

1 LC-3 Convert

The `lc3convert` program converts machine code to an executable that can be run in the simulator.

Note: The first line in the program contains the address of the first instruction. For example, if the first line of the program is `0011 0000 0000 0000`, the first instruction will be located at `x3000` when running the simulator.

Command	**Description**
`lc3convert yourfile.bin`	`lc3convert` will create `yourfile.obj`, which can then be used in the simulator.
`lc3convert -b2 yourfile.bin`	Signifies that "`yourfile.bin`" has machine code in binary (base 2). Same as "`lc3convert yourfile.bin`".
`lc3convert -b16 yourfile.bin`	Signifies that "`yourfile.bin`" has machine code in hex (base 16).

Errors	**Description**
line contains only x digits	Indicates that only *x* digits of a 16-bit LC-3 instruction was specified.
line contains more than 16 digits	Indicates that the line has too many digits.
contains unrecognizable characters	Indicates that invalid characters were used.
constant outside of allowed range	Immediate values such as `imm5`, `PCoffset11`, `PCoffset9`, `offset6`, and `trapvect8` have a valid range of $[-2^{num\_bits-1}, 2^{num\_bits-1} - 1]$. (This error only applies if the file is in hex.)

2 LC-3 Assembler

The LC-3 Assembler (`lc3as`) assembles the file so it can be run in the simulator.

Command	**Description**
`lc3as yourfile.asm`	Creates `yourfile.obj`, which is the object file that can be loaded into the simulator and `yourfile.sym`, which is the symbol table.

Errors	**Description**
file contains only comments	Indicates that there are no assembly instructions in the file, only comments.
no .ORIG or .END directive found	All programs must begin with `.ORIG` followed by the starting address of the program, and must end with `.END`.
no .ORIG directive found	All programs must begin with `.ORIG` followed by the starting address of the program.
multiple .ORIG directives found	Only one `.ORIG` command is allowed per file.
instruction appears before .ORIG	Instructions must start after the `.ORIG` directive.
no .END directive found	All programs must end with `.END`.
label appears before .ORIG	All code must be after the `.ORIG` directive.
label x has already appeared	Indicates that a second label with the same name has been found.
unknown label x	Indicates that label *x* is referenced, but does not exist.
label has already appeared	Each label in a program must only be used once. However, it can be referenced multiple times.
illegal operands for x	Indicates that the instruction has invalid operands.
unterminated string	Indicates that a string specified by the `.STRINGZ` directive was not closed with quotation marks.
contains unrecognizable characters	Indicates the line has characters that the assembler cannot parse.
WARNING: All text after .END ignored	Any instructions after the `.END` directive are not processed by the assembler.
constant outside of allowed range	All constants are limited by their bitwidth. The valid range is $[-2^{num\_bits-1}, 2^{num\_bits-1} - 1]$.

Note: the assembler will tell the total number of errors found in each pass and the line on which these errors occur. However, the assembler only checks for syntax errors, not whether the code is actually doing what it is supposed to.

3 LC-3 Simulator

The Linux/UNIX version of the simulator is used in this class and will be the version that all the programs are tested against. It is STRONGLY recommended that you use this version rather than the Windows version.

Basic Commands	
Command	**Description**
`lc3sim yourfile.obj`	Load your program into the *command-line* simulator. You can also type `lc3sim` and use the command "`file yourfile.obj`" to load the program.
`lc3sim-tk yourfile.obj`	Load your program into the *graphical* simulator. You can also type `lc3sim-tk` and load a file by clicking the "*Browse*" button at the bottom of the screen.
`reset`	Reset the LC-3 and reload the last file.
`quit`	Quit the simulator.
`help`	Print the help message.

Running the program	
Command	**Description**
`continue`	Continues (or starts) execution of the program.
`break`	Sets a breakpoint. In the graphical version, you can double-click on a memory location to set a breakpoint. The line will turn red when a breakpoint is set.
`finish`	Execute until the end of the current subroutine.
`next`	Execute next instruction. If the next instruction is a subroutine (`JSR`) or `TRAP`, it will execute the entire subroutine or `TRAP`.
`step`	Execute one instruction. If the next instruction is a subroutine (`JSR`) or `TRAP`, it will *step into* the subroutine or `TRAP`.

Examining Memory and Registers[1]	
Command	**Description**
`list`	List instructions at the PC, an address, or label.
`dump ...`	Dump memory at the PC, and address, or a label.
`translate <addr>`	Show the value of a label and print the contents.
`printregs`	Print registers and current instruction.
`memory <addr> <val>`	Set the value stored at a memory location.
`register <reg> <val>`	Set a register to a value.
`execute <file name>`	Execute a script file.

[1]The graphical version displays memory and registers, so these commands are specific to the command-line version. *After editing a register or memory value in the graphical version, press the Enter key to apply the change.*

4 GNU C Compiler - gcc

Programs written in C for the class will be compiled using the GNU C compiler (`gcc`). `gcc` is available on all the Linux EWS machines. Although you may work on other machines to write your code, all C programs will be compiled using `gcc` for grading, so it is your responsibility to make sure it works on the Linux EWS machines.

`gcc` has many flags and command-line options, so only those used by the class are described here. For more information about `gcc`, see *http://gcc.gnu.org* or type "`man gcc`" on a Linux machine. For each MP, we will provide the correct command to compile your program with `gcc`.

The typical command to compile C programs for this class is
```
gcc -Wall -ansi -g -o mp_num mp_num.c
```

Flag	**Description**
`-Wall`	Turns on all warnings when compiling.
`-ansi`	Makes `gcc` use the ISO C89 standard.
`-g`	Creates debugging symbols so you can run your program in a debugger.
`-o`	Specifies the name of the compiled program. In the example above, the program would be named "`mp_num`".

After all the flags comes the source file(s). In the example above, the source file is `mp_num.c`.

5 Debugging with GDB and DDD

The GNU Project Debugger, `gdb`, is a command-line program that allows you to debug the C programs that you will write. The DataDisplayDebugger, `ddd`, is a graphical front-end for `gdb`, so all the `gdb` commands work for it as well. It will be useful to know how to use a debugger since some of the C programming assignments at the end of the semester are more challenging. Even if you do not need a debugger for this class, you will need one at some point in the future, so now is a good time to learn. More documentation on `gdb` is available at *http://www.gnu.org/software/gdb/*, and more information on `ddd` can be found at *http://www.gnu.org/software/ddd/*.

Compiling: When compiling with `gcc`, make sure to use the "`-g`" flag, which will compile with debugging symbols for use with `gdb`.

Basic Commands	
Command	**Description**
`gdb my_program_name`	Start GDB.
`ddd my_program_name`	Start DDD.
`quit`	Quit GDB or DDD.

Program Control Flow	
Command	**Description**
`run`	Start the debugged program. If the program takes command-line arguments, you may specify them as well. For example, if you normally start the program with the command "`my_program argument1`", you would use the command "`run argument1`".
`finish`	Finish executing until the selected stack frame returns.
`continue`	Continue to run the program until the next breakpoint. Use the `continue` command after you have hit a breakpoint and want to continue running the program.
`step`	Step through a line of source in the program. `step` will *step into* function (subroutine) calls.
`next`	Step through the program, but *step over* the function (subroutine) calls (the function calls will still execute).

Displaying Information	
Command	**Description**
`list`	List the source code of the program with line numbers. You can also specify a line number to display the source code around that line (e.g., `list 100`.
`print`	Print out an expression (variables, etc.).
`info break`	Print out breakpoint information.
`info locals`	Print out local variables.
`info function`	Print out all function names.
`info variables`	Print out all global and static variables.

Breakpoints	
Command	**Description**
`break <line # or function name>`	Set a breakpoint at the specified line (e.g., `break 100`) or at the specified function (e.g., `break foo`).
`cond <breakpoint #> <C expression>`	Set a conditional breakpoint that will only break when the C expression evaluates to true (e.g., `cond 1 if (x > 0)` will make breakpoint 1 only break if the variable `x` is greater than 0).
`delete <breakpoint #>`	Delete the breakpoint.

Stack Information	
Command	**Description**
`where` or `bt`	Print a backtrace of all the stack frames.
`up`	Move up to the stack that called this one.
`down`	Move down to the stack called by this one.

6 Vim and gVim

Vim is different than most editors because it has several modes, such as normal mode (command mode), visual mode, and insert mode. Typing "`vimtutor`" in an xterm window will start a tutorial that teaches the basics of Vim. For more detailed information on Vim see *http://www.vim.org*.

Basic Commands	
Command	**Description**
`vim` or `gvim`	Start Vim or gVim.
`vim my_file`	Open a file in Vim.
`i`	Switch to insertion mode. Vim starts in command mode, which does not allow you to edit text. Switching to insertion mode allows you to edit text.

Command Mode Operations	
Command	**Description**
`escape`	Switches to command mode. Being in command mode allows you to perform the operations listed below.
`:w`	Save (write) the file you are currently editing.
`:q`	Quit Vim.
`:q!`	Quit Vim without saving changes.
`:line`	Jump to line number *line* (e.g., "`:30`" will jump to line 30.)
`:u`	Undo.
`CTRL-R`	Redo.
`y`	Copy (yank) highlighted text.
`d`	Cut highlighted text.
`p`	Paste text.

Visual mode: Another mode that lets you select text for copying and pasting. To switch to visual mode press `CTRL-V`. You can then use the arrow keys to select text. After selecting text, press `escape` to switch back to command mode.

7 Emacs and XEmacs

Unlike Vim, Emacs does not have modes. Emacs commands are typically written in the form "`C-x C-s`", which means hold `CTRL` and `x`, then press `CTRL` and `s`. Some commands are written such as "`M-x`", where `M` stands for the meta key, which corresponds to `ALT` or `escape` on most machines. For more detailed information on Emacs visit *http://www.gnu.org/software/emacs/*.

Basic Commands	
Command	**Description**
`emacs` or `xemacs`	Start Emacs or XEmacs.
`emacs my_new_file`	Create a new file.
`C-h t`	Start the Emacs tutorial.
`C-x C-s`	Save the file.
`C-x C-c`	Exit Emacs.
`M-x goto-line`	Go to a specific line number.
`M-w`	Copy highlighted text.
`C-w`	Cut highlighted text.
`C-_`	Undo (hold `CTRL`, `shift`, and the dash).
`C-y`	Paste text.

ECE190 Tutorial One

Introduction to Linux and LC-3 Tools

John H. Kelm

Logging In and Account Access

- You can log in **locally** using any of the EWS machines located in Everitt, Engineering Hall, DCL, or Grainger.
- Any of these machines can also be accessed **remotely.**
- Once you login, enter the command **`ece190`** to access your ece190 directory located under: **`'/work2/ece190/<net id>'`**
- From this directory you will have access to all the LC-3 commands and the handin command to submit your completed Machine Problems.
- *Note*: Grading is done on EWS Linux machines—make sure your code works on one of these machines before turning it in.

Basic Command Reference

Command	Description
man <command>	Prints the manual page help information (usage:'man ssh')
ls, ls -al	List contents of directory
cd <directory>	Change the current directory—with no options, takes you to your home directory
rm <filename>	Remove a file (Caution!)
pwd	Print the current directory
cp <src file> <dest file>	Copy a file from one location to another
mv <src file> <dest file>	Move a file from one location to another (Caution!)

A Sample LC-3 ASM File

Run the command **ece190** and follow the steps below.

```
eelnx12> ece190
ece190> pwd
/work2/ece190/jkelm2
ece190> cd /homesta/ece190/linux_tutorial/
ece190> ls
mp0.asm
ece190> cp mp0.asm /work2/ece190/jkelm2/
ece190> cd /work2/ece190/jkelm2
ece190> ls
mp0.asm  -shell
ece190> vim mp0.asm
```

See where you currently are in the filesystem.

Go to TA directory.

Copy the file to your home directory.

Go back to your home directory.

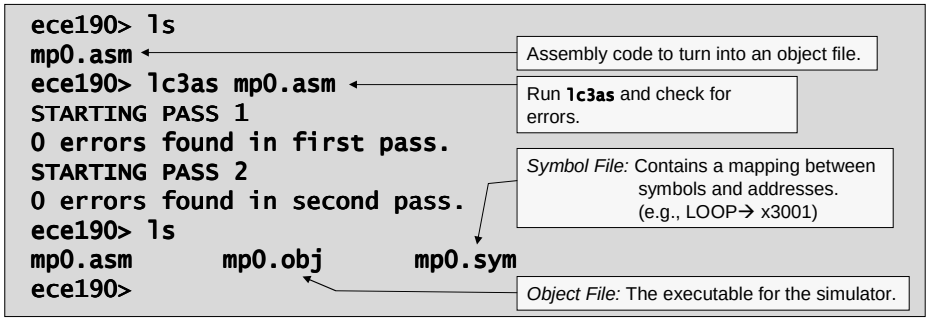
Text Editor Choices

- Editors:
  - Vim (or gVim)
    - Check out http://www.vim.org for more details.
  - Emacs (or XEmacs)
    - Go to http://www.gnu.org/software/emacs/ to learn about this option.
- Make sure you know how to **open**, **edit**, **close**, **save** files—this may sound trivial, but both have large learning curves due to their immense power and utility.

LC-3 Assembler

Purpose: Convert human-readable LC-3 assembly code into machine-readable ***object*** (.obj) file.

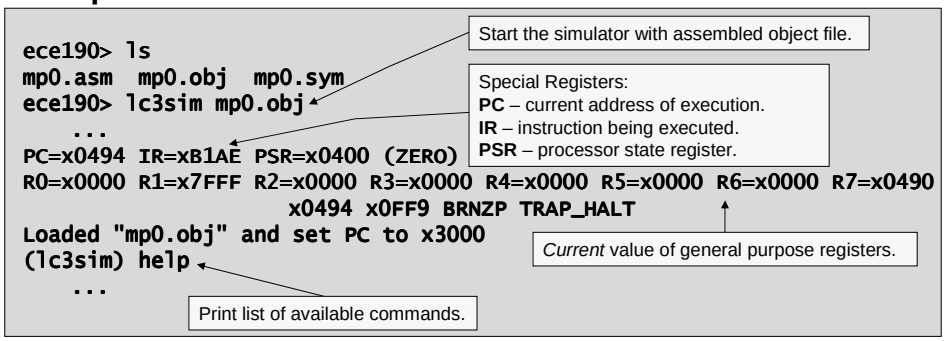
Example:



LC-3 Simulator

Purpose: To step through the output of an LC-3 program cycle by cycle.

Example:



Note: There is a GUI version (`lc3sim-tk`) that can also be used.

LC-3 Command Reference

- **break** (*clear* | *list* | *set* <*label* | *address*>) – Sets a breakpoint where execution will stop.
- **step**/**next** – move from one instruction to the next (i.e., state transition/clock tick)
- **continue**/**finish** – Run the code up until the next breakpoint (or run it until complete).
- **list** <*label* | *address* > – Print the code/data around the address or label given as an argument.
- **dump** <*label* | *address*> – Display values in memory near location taken as argument.
- **memory**/**register** – Allows the user to set value in memory or register, respectively.
- **reset**/**quit** – Restart the simulation at .ORIG address or leave the simulation, respectively.

Submitting Your Finished Work

1) Log in to an EWS machine.
2) Run the `ece190` command and make sure the file you are submitting is located in the current directory
3) Submit your MP by running `handin`.

Example:

```
mycomputer$  ssh -ljkelm2 eelnx12.ews.uiuc.edu
please enter your password: ******
eelnx12> ece190
ece190> ls
mp0.asm
ece190> handin --MP 0 mp0.asm
The file "mp0.asm" has been copied.
          ...
ece190> exit
eelnx12> exit
```

Make sure you are in your ece190 account.

Check to make sure file exists.

Turn in the file (can be done multiple times before deadline).

Check **handin** output to make sure file was copied.

Advice

- Backup often
  - Example: Create backup and disable writing

```
cp mp1.bin mp1_backup0.bin
chmod a-w mp1_backup0.bin
```

- Learn to use command line tools
  - Example:
    - Vim instead of gVim.
    - Commands: **cp**, **rm**, **mv**, etc. instead of a GUI FS manager.

Supplemental: Remote Login

- EWS only allows *secure* logins—i.e., ssh, putty, sftp
- CITES has information on obtaining and installing SSH clients: http://www.cites.uiuc.edu/security/ssh/index.html
- There is a listing of EWS machines at http://www.ews.uiuc.edu/labs/.

ECE190 Tutorial Two

Introduction to C Programming

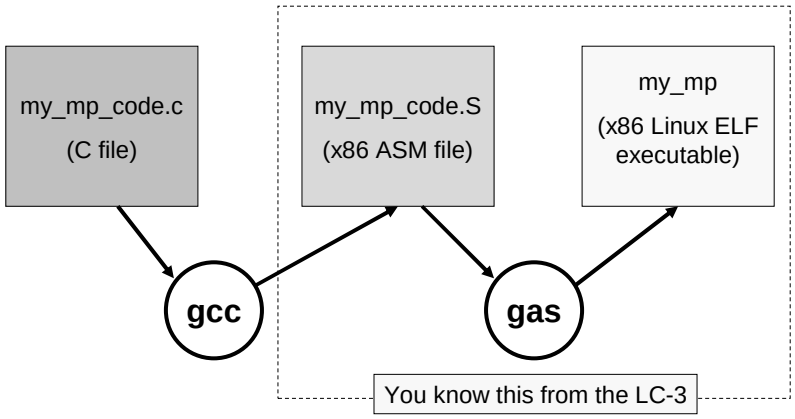
John H. Kelm

Vim/Emacs Setup

- Use syntax highlighting and automatic indentation to make bug finding easier.
- We created setup files for Emacs and Vim.

```
cp /homesta/ece190/for_students/.vimrc ~/.vimrc
…or…
cp /homesta/ece190/for_students/.vimrc ~/.gvimrc
…or…
cp /homesta/ece190/for_students/.emacs ~/.emacs
```

Compilation Process



Note: Linking omitted for clarity

Creating a C Program

- Go to your favorite text editor (which of course is Vim)

```
#include <stdio.h>

int main() {
      printf("I <3 ECE190.\n");
      return 0;
}
```

This is what gives us `printf()`

- Save the file as: *c_tut.c*

Compiling First C Program

- In the same directory as where you just saved the file, compile it.

```
ece190> gcc -ansi -Wall -g -oc_tut c_tut.c
```

  - gcc – The GNU compiler (Important: GNU is Not Unix!)
  - -Wall -ansi – Turn on all warnings (More on this soon)
  - -o<output_file> – Where the executable will go
  - -g – Include a symbol table with the executable (For debugging)
- Run the program:

```
ece190> ./c_tut
I <3 ECE190!
ece190>
```


Warnings and Errors

- The *-Wall -ansi* command line arguments will force gcc to show **ALL** warnings.
- We require that your program create **NO** warnings or errors.
- A **warning** is a friendly reminder that although what you are doing functions, it is incorrect.
- An **error** is a not so friendly reminder that something is incorrect and compilation cannot proceed.

Adding User Input

- We now want to allow the user to input a number and then print it back out to the screen.

```
#include <stdio.h>

int main() {
        int input;
        printf("Input: ");
        scanf("%d", &input);
        printf("You entered: %d\n", input);
        return 0;
}
```

Note: &input is the address

Further Note: input is the value

- Recompile!

Conditionals

- What if we only want to print out numbers greater than 5?

```
int main() {
        …
        scanf("%d", &input);
        if (input > 5)
                printf("You entered: %d\n", input);
        else
                printf("Your number was less than 5!\n");
        return 0;
}
```

Why are there no braces?

Looping Constructs

- What if we want to say we really love the class?

```
#include <stdio.h>

int main() {
        int i;
        for (i = 0; i < 5; i++) {
                printf("I <3 ECE190.\n");
        }
        return 0;
}
```

What value does this have?

- Note: We used braces ('{' and '}'), but here we did not need to. If the block of code inside the loop (or conditional!) is more than one line, they are required.

Switch Statements

- Used when you need to make a decision that may have many outcomes

```
char input;
printf("select a state ('g', 'y', 'r'): ");
scanf("%c", &input);
switch (input) {
        case 'g':
                printf("Green!\n");
                break;
        /* other cases */
        default:
                printf("Bad input!\n");
                break;
}
```

Remember to use a break

Fall-through case

Debugging Tips

- Errors and warnings ***will*** happen, but not in MP's that get turned in for all the credit you deserve ☺.
- Use **`printf()`** calls to diagnose problems.
- Think about whether you are dealing with addresses (i.e., pointers) or values (i.e., locations in memory)
- Know what *****, ******, **&** are doing to alter the meaning of your variables.
- The GNU debugger, **gdb**, is a vital tool--we will introduce it at a later date.

ECE190 Tutorial Three

Debugging with GDB

John H. Kelm

Role of Debugging

- A way to find and correct **runtime** errors.
- **Pointer** analysis is hard to do statically (i.e., at compile time).
- **Semantics** and **Algorithms**: Code *looks* correct, but produces incorrect results. Check by stepping through execution.
- Debugging can answer one important question: Where is this darn **segfault** coming from?

Common Error Types

- Dereferencing null pointers

```
*ptr = 5
```

What if we set **ptr** to NULL by mistake?

(e.g., str_ptr = NULL instead of *str_ptr = NULL)

- Buffer overflow

```
int array[10];
```

Where is array in memory?

```
...
array[i] = 0xBEEF;
```

What if i == -4?

Will this always crash?

- Infinite looping

```
for(i = 0; i < 10; i--) { … }
```


GDB Commands

- **run *arg1 arg2* …** – Start the program in motion.
- **step** – Go one line of code forward (may be multiple instructions).
- **next** – Like step, but steps over function calls (i.e., you do not start stepping through a function call, rather you step to the line proceeding the call).
- **break *label*** – When the application is about to execute this line, stop.
- **delete <bp #>** – Remove a breakpoint.
- **continue** – Restart execution from a breakpoint.
- **Ctrl+C** – Stop gdb. Useful when you are stuck in an infinite loop.

GDB Commands (cont.)

- **print *symbol*** – Print a variable (can use `*ptr` to get value of a pointer).
- **list** – Print C code for where we are in the program.
- **info locals** – Display the local variables.
- **disassemble** – View x86 assembly language of our program.
- **bt** – Get a back trace (i.e., view the call stack)
- **frame** – The current stack frame.
- **kill** – Terminate the program (send it a SIGKILL—more on this in 391 or an OS class).
- **quit** – Exit gdb.

Getting Started

- Log into your ece190 account.
- Get the file:

```
cp /homesta/ece190/for_students/gdb_test.c ~/
```

- Compile the file (Must have -g flag):

```
gcc -g -ogdb_test gdb_test.c
```

- Try executing `gdb_test` and watch it fail :-(.
- Start the application in gdb:

```
gdb ./gdb_test
```

Stepping

- **Concept**: Just like lc3sim, but a whole line of C code (which could be any number of x86 instructions).
- This is the basic form of movement through programs that you will do in ECE190.
- But the program just executes and either crashes or finishes—*how do I 'step'*?

Breakpoints

- Stop program *prior* to execution of a specific *line* of code.

How do I do it?

A basic method for stopping execution immediately and allowing you to step through it from the start is by setting a breakpoint at `main()`. For this to work, the breakpoint must be set prior to typing `run`.

How does GDB do it?

GDB is smart and places the equivalent of a TRAP in LC-3 at the point you want to break (gdb for x86 machines use the `int3` instruction). When the program being traced (for more information look into the Linux ptrace facilities) by gdb hits that point, the OS is summoned and runs a special routine that hands control off to gdb. You can consider it magic.

```
(gdb) break main
Breakpoint 1 at 0x80483bb: file gdb_test.c, line 6.
```

Start Running!

- We can view the breakpoint:

```
(gdb) info breakpoints
```

- We will now let execution begin and go to our first breakpoint:

```
(gdb) run
       ...GDB Information Stuff...
char str[10] = "Howdy";
(gdb)
```

- We can view the where we are in the code too:

```
(gdb) list
```


Move Around a Little

- Now we can proceed a little bit, step a few times:

```
(gdb) step
11     for (i = 0; i > -10; i++) {
```

- Where are we in the stack? A less trivial example:

```
(gdb) kill
(gdb) b bar
(gdb) run
(gdb) bt
#0 bar (d=12472013) at gdb_test.c:35
#1 0x080483fe in foo() at gdb_test.c:31
#2 0x08048378 in main () at gdb_test.c:13
```

Where you are now.

What is this? Where would it be in memory?

Displaying Values

- We can stop the program, but what information can we get about that current process' *state*?

The command 'print' allows us to view the value of a particular variable that is in the current scope of execution. This is the main piece of information you will be interested while debugging in ECE190.

```
(gdb) print str_ptr
$1 = 0xbfffb220 "Howdy"
```

Nifty Hints:

gdb can do auto-completion by hitting tab after print. You may also want to look deeper into the use of `info` commands. (Type 'help info' at the `(gdb)` prompt).

To dump all local variable values try : `info locals`

How Does Any This Help Us?

- It may not be clear at first, but not all problems can be solved by strategically placing `printf()` (even with `fflush(stdout)`!)
- If the application just segfaults, like this one, what can we do?
  - Randomly change things until it works.
  - Scatter `printf` statements everywhere.
  - **Use the debugger, of course!**

What is a Segmentation Fault?

- Requires understanding ***paging*** (note: segmentation is a bit of a misnomer and really a relic from old architectures like x86 that think segmentation is cool)—All of this is way beyond the scope of ECE190.

What you need to know:

- You own a certain area of memory (e.g., *your* application space, *your* stack space, *your* global variables)
- You cannot touch other parts of memory (e.g., whatever really is at address 0x0 (NULL pointer—usually nothing is here), or thinking in terms of LC3—A regular user should not be able to alter the trap space that the OS owns).
- If you try to dereference (i.e., get the value at) a pointer to a location you do not own, Linux causes your program to **die**. (Your program becomes `0xDEADBEEF` or rather, a `SIGSEGV` AKA a segmentation violation).

Let Us Find That Segfault

- Load the program back into gdb and start the application with the 'run' command.

What happened to my program?

```
Program received signal SIGSEGV, Segmentation fault.
0x0804838a in main () at gdb_test.c:18
18 if (*str_ptr != 'o')
(gdb)
```

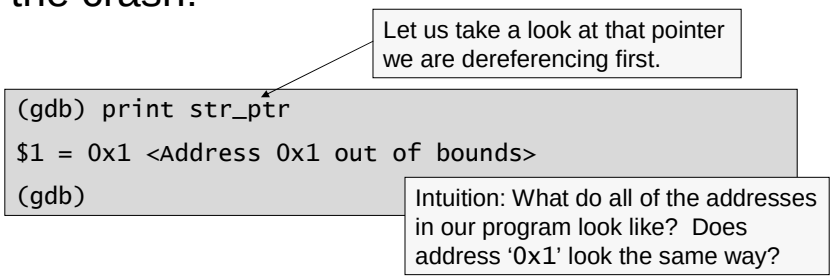
Where did it happen?

Whatcha' gonna do?
Who you gonna call?
Ghostbusters? No. **GDB**!

Who did this to me?
The OS did it because you did something bad.

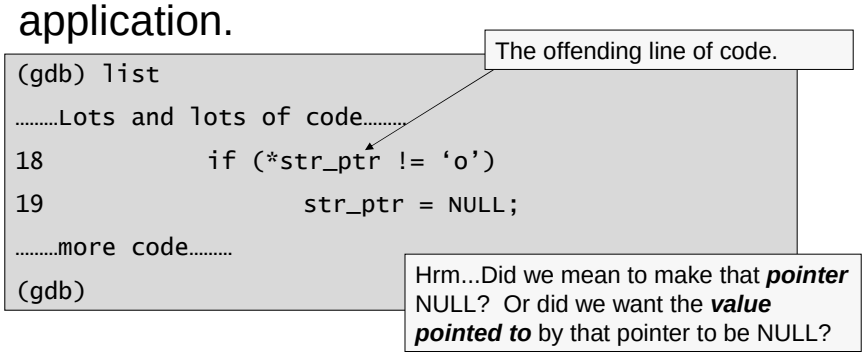
How to be Sure This is Wrong

- Remember, we can view values at runtime? Well we can even see them after the crash:



Determine What We Did Wrong

- Now we may want to look at the code and see why Linux crashed our hapless application.



Extension to Lecture 1

This handout describes the first problem known to be undecidable. This material is beyond the scope of the course but is nonetheless reasonably accessible and important, and you should eventually (in future semesters) be able to recognize it readily. This material is not intended to be part of the core material for the class, and you will not be tested on it; it's just for fun and the future.

The Halting Problem

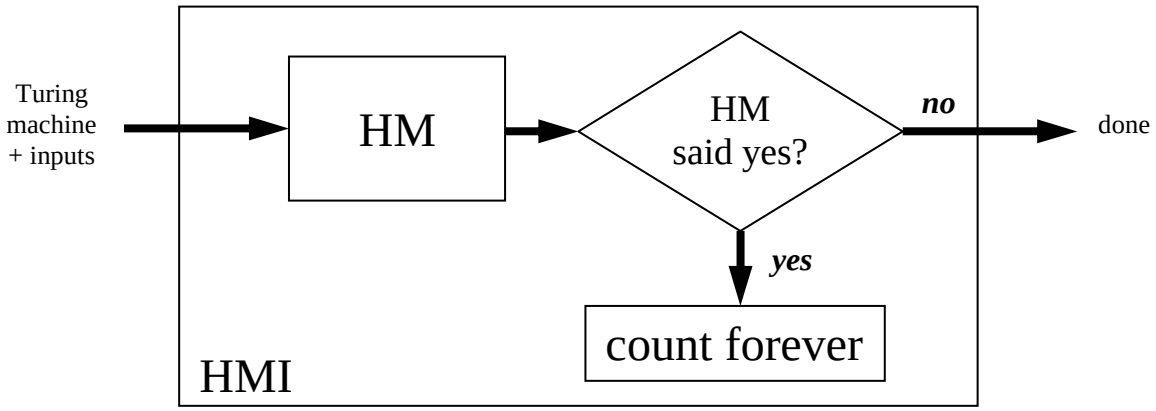
Another thing that Alan Turing did in his paper in 1936 was to introduce (and prove) that there are in fact problems that cannot be computed by a universal computing machine, or *Turing machine*, as we've come to call them today (remember also that everything that we call a computer today is equivalent to a Turing machine). The problem that proved indecidable, using proof techniques almost identical to those developed for similar problems in the 1880s, is now known as the halting problem, and is the subject of this document.

Turing also conjectured that his definition of computable was identical to the "natural" definition. In other words, a problem that cannot be solved by a Turing machine cannot be solved in any systematic manner, with any machine, or by any person. **This thesis remains unproven!** However, neither has anyone been able to disprove the thesis, and it is widely believed to be true. Disproving the thesis requires that one demonstrate a systematic technique (or a machine) capable of solving a problem that cannot be solved by a Turing machine. No one has been able to do so to date.

The halting problem is easy to state and easy to prove undecidable. The problem is this: given a Turing machine and an input to the Turing machine, does the Turing machine finish computing in a finite number of steps (a finite amount of time)? In order to solve the problem, an answer, either yes or no, must be given in a finite amount of time regardless of the machine or input in question. Clearly some machines never finish. For example, we can write a Turing machine that counts upwards starting from one.

To see that no Turing machine can solve the halting problem, we begin by assuming that such a machine exists, and then show that its existence is self-contradictory. We call the machine the "Halting Machine," or HM for short. HM is a machine that operates on another machine and its inputs to produce a yes or no answer in finite time: either the machine in question finishes in finite time (HM returns "yes"), or it does not (HM returns "no"). The figure below shows its operation:

From HM, we construct a second machine that we call the HM Inverter, or HMI. This machine inverts the sense of the answer given by HM. In particular, the inputs are fed directly into a copy of HM, and if HM answers "yes," HMI enters an infinite loop. If HM answers "no," HMI halts. A diagram appears on the next page.



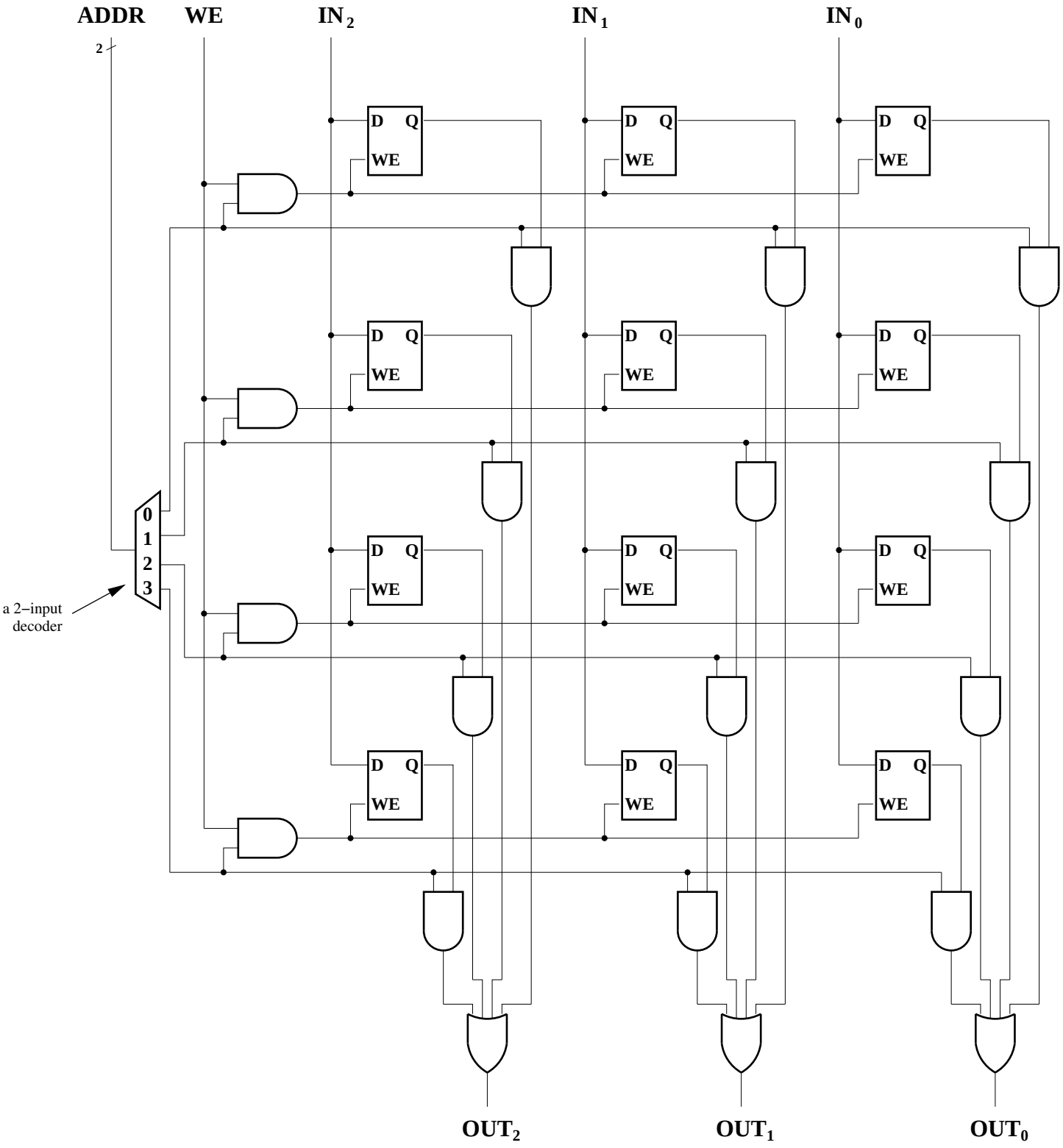
The inconsistency can now be seen by asking HM whether HMI halts when given itself as an input (repeatedly). Two copies of HM are thus being asked the same question. One copy is the one that we are using, and the second is embedded in the HMI machine that we are using as the input to our HM. As the two copies of HM operate on the same input (HMI operating on HMI), they should return the same answer: a Turing machine either halts on an input, or it does not; they are deterministic.

Let's assume that the HM tells us that HMI operating on itself halts. Then the copy of HM in HMI (when HMI executes on itself, with itself as an input) must also say "yes." But this answer implies that HMI doesn't halt (see the figure above), so the answer should have been no!

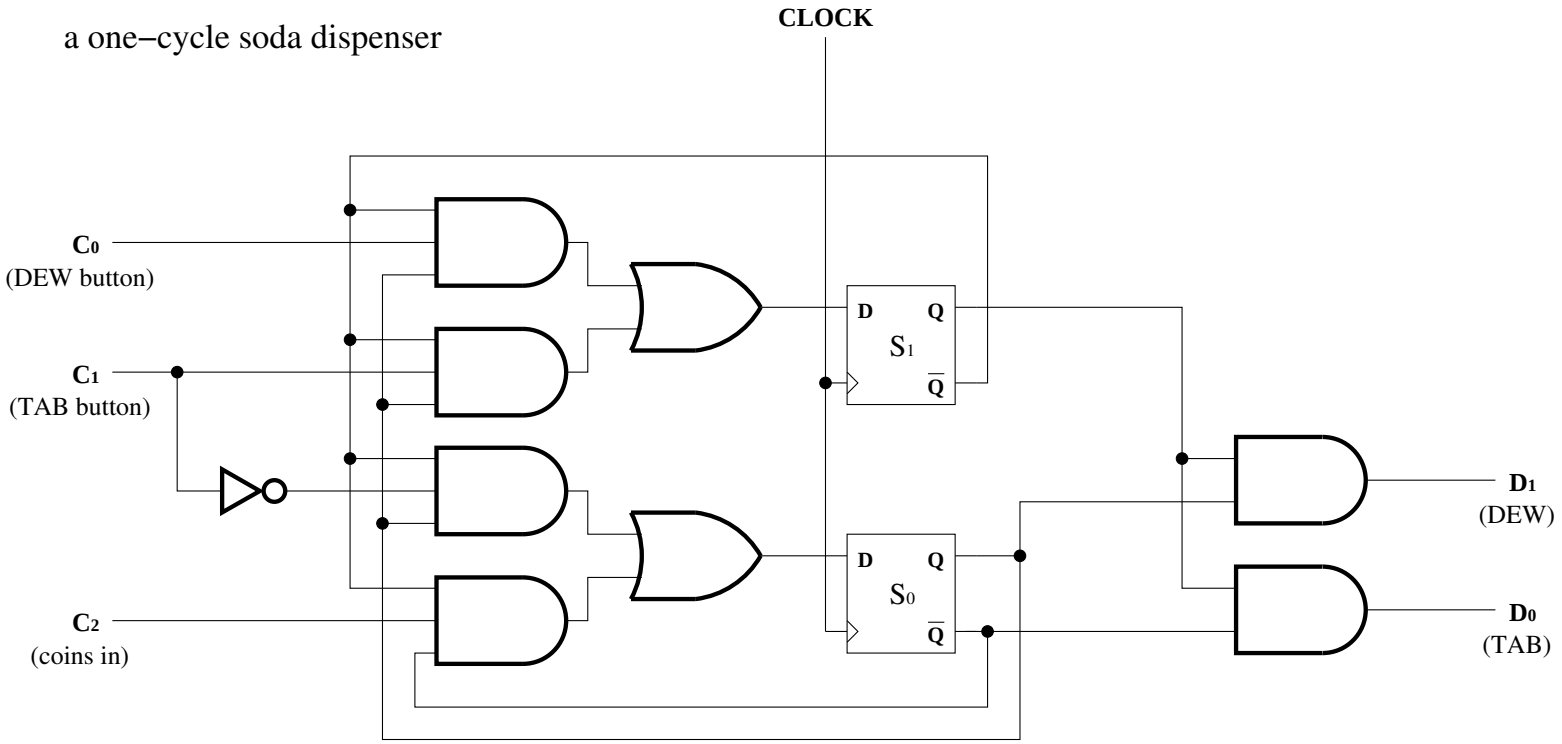
Alternatively, we can assume that HM says that HMI operating on itself does not halt. Again, the copy of HM in HMI must give the same answer. But in this case HMI halts, again contradicting our assumption.

Since neither answer is consistent, no consistent answer can be given, and the original assumption that HM exists is incorrect. Thus no Turing machine can solve the halting problem.

You may be familiar with a related problem known as the Liar's paradox (which is at least 2,300 years old). In its stengthened form, it is the following sentence: "This sentence is not true."


ADDR
WE
IN2
IN1
IN0
2
D Q
WE
0
1
2
3
a 2−input
decoder
OUT2
OUT1
OUT0

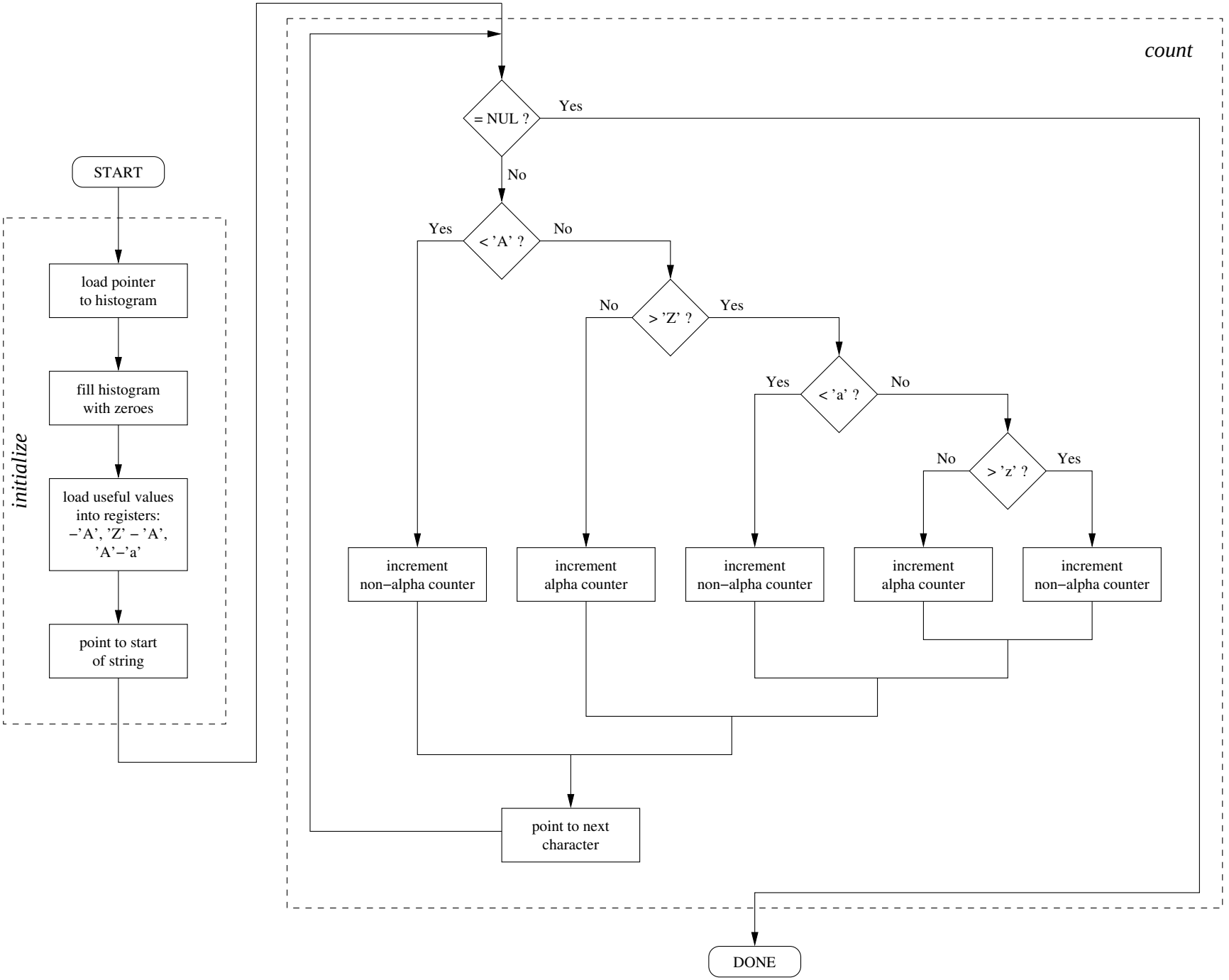
a one−cycle soda dispenser



```
            counting to ten with PC-relative addressing

0x3000  0010 010 010011111     _____________________________________________

0x3001  0001 010 010 1 00001   _____________________________________________

0x3002  0011 010 010011101     _____________________________________________

   .
   .    (something that we want to do ten times)
   .

0x3010  0001 010 010 1 10110   _____________________________________________

0x3011  0000 100 111101110     _____________________________________________

   .
   .
   .

0x30A0  0000000000000000       _____________________________________________

--------------------------------------------------------------------------------

              counting to ten with indirect addressing

0x3000  1010 011 010011111     _____________________________________________

0x3001  0001 100 011 1 00001   _____________________________________________

0x3002  1011 100 010011101     _____________________________________________

   .
   .    (something that we want to do ten times)
   .

0x3010  0001 100 100 1 10110   _____________________________________________

0x3011  0000 100 111101110     _____________________________________________

   .
   .
   .

0x30A0  0100000100100011       _____________________________________________

   .
   .
   .

0x4123  0000000000000000       _____________________________________________
```

```
            counting to ten with base+offset addressing

0x3000  1110 110 010011111     _____________________________________________

0x3001  0110 001 110 000000    _____________________________________________

0x3002  0001 001 001 1 00001   _____________________________________________

0x3003  0111 001 110 000000    _____________________________________________

   .
   .    (some more complex task that we want to do ten times)
   .

0x3018  0001 001 001 1 10110   _____________________________________________

0x3011  0000 100 111100111     _____________________________________________

   .
   .
   .

0x30A0  0000000000000000       _____________________________________________
```

```
; read a decimal number from the keyboard,
; convert it from ASCII to 2's complement, and
; store it in a predefined memory location.  If
; any non-numeric character is pressed, or the
; number overflows, store a 0 and print an error
; message.

; R0 holds the value of the last key pressed
; R1 holds the current value of the number being input
; R2 holds the additive inverse of ASCII '0' (0xFFD0)
; R3 is used as a temporary register

00110000 00000000     ; starting address is x3000

0010 010 000010100    ; LD R2,x14      (put the value -x30 in R2)
0101 001 001 1 00000  ; AND R1,R1,#0   (clear the current value)
1111 0000 00100000    ; TRAP x20       (read a character)
1111 0000 00100001    ; TRAP x21       (echo it back to monitor)
0001 011 000 1 10110  ; ADD R3,R0,#-10 (compare with ENTER)
0000 010 000010001    ; BRz x11        (ENTER pressed, so done)
0001 000 000 0 00 010 ; ADD R0,R0,R2   (subtract x30 from R0)
0000 100 000010001    ; BRn x11        (smaller than '0' means error)
0001 011 000 1 10110  ; ADD R3,R0,#-10 (check if > '9')
0000 011 000001111    ; BRzp xF        (greater than '9' means error)
0001 011 001 0 00 001 ; ADD R3,R1,R1   (sequence of adds multiplies R1 by 10)
0000 100 000010101    ; BRn x15        (overflow, but not really necessary here)
0001 011 011 0 00 011 ; ADD R3,R3,R3
0000 100 000010011    ; BRn x13        (overflow, but not really necessary here)
0001 001 001 0 00 011 ; ADD R1,R1,R3
0000 100 000010001    ; BRn x11        (overflow)
0001 001 001 0 00 001 ; ADD R1,R1,R1
0000 100 000001111    ; BRn xF         (overflow)
0001 001 001 0 00 000 ; ADD R1,R1,R0   (finally, add in new digit)
0000 100 000001101    ; BRn xD         (overflow)
0000 111 111101101    ; BRnzp 0x1ED    (get another digit)

11111111 11010000     ; the additive inverse of ASCII '0'
11111111 11111111     ; storage for the result

; done
0011 001 111111110    ; ST R1,x1FE
1111 0000 00100101    ; TRAP x25

; print error message: "non-digit pressed"
; R4 holds pointer to character
; R0 used to pass output character to trap
; end of string marked with negative value
1110 100 000001001    ; LEA R4,x9      (point R4 to the start of the string)
0110 000 100 000000   ; LDR R0,R4,#0   (read character pointed to by R4)
0000 100 000000011    ; BRn x3         (done printing)
1111 0000 00100001    ; TRAP x21       (print the character)
0001 100 100 1 00001  ; ADD R4,R4,#1   (point to next character)
0000 111 111111011    ; BRnzp x1FB     (loop to read string)
0101 001 001 1 00000  ; AND R1,R1,#0   (clear the current value)
0000 111 111110110    ; BRnzp x1F6     (store the zero and end)

; print error message: "overflow"
1110 100 000010101    ; LEA R4,x15     (point R4 to the start of the string)
0000 111 111110111    ; BRnzp x1F7     (branch back to the string-printing code)

; first error message (all in ASCII)
00000000 00001010     ; LF (line feed)
00000000 01101110     ; 'n'
00000000 01101111     ; 'o'
00000000 01101110     ; 'n'
00000000 00101101     ; '-'
00000000 01100100     ; 'd'
00000000 01101001     ; 'i'
00000000 01100111     ; 'g'
00000000 01101001     ; 'i'
00000000 01110100     ; 't'
00000000 00100000     ; ' '
00000000 01110000     ; 'p'
00000000 01110010     ; 'r'
00000000 01100101     ; 'e'
00000000 01110011     ; 's'
00000000 01110011     ; 's'
00000000 01100101     ; 'e'
00000000 01100100     ; 'd'
00000000 00001010     ; LF (line feed)
11111111 11111111     ; end of string marker

; second error message (all in ASCII)
00000000 00001010     ; LF (line feed)
00000000 01101111     ; 'o'
00000000 01110110     ; 'v'
00000000 01100101     ; 'e'
00000000 01110010     ; 'r'
00000000 01100110     ; 'f'
00000000 01101100     ; 'l'
00000000 01101111     ; 'o'
00000000 01110111     ; 'w'
00000000 00001010     ; LF (line feed)
11111111 11111111     ; end of string marker
```


START
initialize
load pointer to histogram
fill histogram with zeroes
load useful values into registers: –'A', 'Z' – 'A', 'A'–'a'
point to start of string
count
= NUL ?
Yes
No
< 'A' ?
Yes
No
> 'Z' ?
No
Yes
< 'a' ?
Yes
No
> 'z' ?
No
Yes
increment non–alpha counter
increment alpha counter
increment non–alpha counter
increment alpha counter
increment non–alpha counter
point to next character
DONE

```
; Count the occurrences of each letter (A to Z)
; in an ASCII string terminated by a NUL character.
; Lower case and upper case should be counted
; together, and a count also kept of all
; non-alphabetic characters (not counting the
; terminal NUL).

; The string starts at x4000.

; The resulting histogram (which will NOT be
; initialized in advance) should be stored starting
; at x3100, with the non-alphabetic count at x3100,
; and the count for each letter in x3101 (A) through
; x311A (Z).

; R0 holds a pointer to the histogram (x3100)
; R1 holds a pointer to the current position in the string
;    and holds the loop count during histogram initialization
; R2 holds the current character being counted
;    and is also used to point to the histogram entry
; R3 holds the additive inverse of ASCII '@' (0xFFC0)
; R4 holds the difference between ASCII '@' and 'Z' (xFFE6)
; R5 holds the difference between ASCII '@' and ''' (xFFE0)
; R6 is used as a temporary register

00110000 00000000     ; starting address is x3000

1110 000 011111111    ; LEA R0,xFF      (point R0 to the start of the histogram)

; fill the histogram with zeroes
0101 110 110 1 00000  ; AND R6,R6,#0   (put a zero into R6)
0010 001 000100000    ; LD R1,x20      (initialize loop count to 27)
0001 010 000 1 00000  ; ADD R2,R0,#0   (copy start of histogram into R2)
; loop to fill histogram starts here
0111 110 010 000000   ; STR R6,R2,#0   (write a zero into histogram)
0001 010 010 1 00001  ; ADD R2,R2,#1   (point to next histogram entry)
0001 001 001 1 11111  ; ADD R1,R1,#-1  (decrement loop count)
0000 001 111111100    ; BRp x1FC       (continue until loop count reaches zero)

; initialize R1, R3, R4, and R5 from memory
0010 011 000011011    ; LD R3,x1B      (set R3 to additive inverse of ASCII '@')
0010 100 000011011    ; LD R4,x1B      (R4 holds difference between ASCII '@' and 'Z')
0010 101 000011011    ; LD R5,x1B      (R5 holds difference between ASCII '@' and ''')
0010 001 000011011    ; LD R1,x1B      (point R1 to start of string)

; the counting loop starts here
0110 010 001 000000   ; LDR R2,R1,#0   (read the next character from the string)
0000 010 000010100    ; BRz x14        (found the end of the string)

0001 010 010 0 00 011 ; ADD R2,R2,R3   (subtract '@' from the character)
0000 001 000000100    ; BRp x4         (branch if > '@', i.e., >= 'A')

0110 110 000 000000   ; LDR R6,R0,#0   (load the non-alpha count)
0001 110 110 1 00001  ; ADD R6,R6,#1   (add one to it)
0111 110 000 000000   ; STR R6,R0,#0   (store the new non-alpha count)
0000 111 000001100    ; BRnzp xC       (branch to end of conditional structure)

0001 110 010 0 00 100 ; ADD R6,R2,R4   (compare with 'Z')
0000 001 000000101    ; BRp x5         (branch if > 'Z')

; note that we no longer need the current character
; so we can reuse R2 for the pointer to the correct
; histogram entry for incrementing
0001 010 010 0 00 000 ; ADD R2,R2,R0   (point to correct histogram entry)
0110 110 010 000000   ; LDR R6,R2,#0   (load the count)
0001 110 110 1 00001  ; ADD R6,R6,#1   (add one to it)
0111 110 010 000000   ; STR R6,R2,#0   (store the new count)
0000 111 000000101    ; BRnzp x5       (branch to end of conditional structure)

; subtracting as below yields the original character minus '''
0001 010 010 0 00 101 ; ADD R2,R2,R5   (subtract ''' - '@' from the character)
0000 110 111110011    ; BRnz x1F3      (if <= ''', i.e., < 'a', increment non-alpha)

0001 110 010 0 00 100 ; ADD R6,R2,R4   (compare with 'z')
0000 110 111110111    ; BRnz x1F7      (if <= 'z', go increment alpha count)
0000 111 111110000    ; BR x1F0        (otherwise, go increment non-alpha)

0001 001 001 1 00001  ; ADD R1,R1,#1   (point to next character in string)
0000 111 111101010    ; BRnzp x1EA     (go to start of counting loop)

1111 0000 00100101    ; TRAP x25       (done)

; the data needed by the program
00000000 00011011     ; 27 loop iterations
11111111 11000000     ; the additive inverse of ASCII '@'
11111111 11100110     ; the difference between ASCII '@' and 'Z'
11111111 11100000     ; the difference between ASCII '@' and '''
01000000 00000000     ; string starts at x4000
```

```
; (An assembly-language version of the original binary code.)

; Count the occurrences of each letter (A to Z)
; in an ASCII string terminated by a NUL character.
; Lower case and upper case should be counted
; together, and a count also kept of all
; non-alphabetic characters (not counting the
; terminal NUL).

; The string starts at x4000.

; The resulting histogram (which will NOT be
; initialized in advance) should be stored starting
; at x3100, with the non-alphabetic count at x3100,
; and the count for each letter in x3101 (A) through
; x311A (Z).

; R0 holds a pointer to the histogram (x3100)
; R1 holds a pointer to the current position in the string
;    and as the loop count during histogram initialization
; R2 holds the current character being counted
;    and is also used to point to the histogram entry
; R3 holds the additive inverse of ASCII '@' (0xFFC0)
; R4 holds the difference between ASCII '@' and 'Z' (xFFE6)
; R5 holds the difference between ASCII '@' and ''' (xFFE0)
; R6 is used as a temporary register

        .ORIG   x3000           ; starting address is x3000

        LEA R0,HIST             ; point R0 to the start of the histogram

        ; fill the histogram with zeroes
        AND R6,R6,#0            ; put a zero into R6
        LD R1,NUM_BINS          ; initialize loop count to 27
        ADD R2,R0,#0            ; copy start of histogram into R2

        ; loop to fill histogram starts here
HFLOOP  STR R6,R2,#0            ; write a zero into histogram
        ADD R2,R2,#1            ; point to next histogram entry
        ADD R1,R1,#-1           ; decrement loop count
        BRp HFLOOP              ; continue until loop count reaches zero

        ; initialize R1, R3, R4, and R5 from memory
        LD R3,NEG_AT            ; R3 holds additive inverse of ASCII '@'
        LD R4,AT_MIN_Z          ; R4 holds difference between ASCII '@' and 'Z'
        LD R5,AT_MIN_BQ         ; R5 holds difference between ASCII '@' and '''
        LD R1,STR_START         ; point R1 to start of string

        ; the counting loop starts here
COUNTLOOP
        LDR R2,R1,#0            ; read the next character from the string
        BRz DONE                ; found the end of the string

        ADD R2,R2,R3            ; subtract '@' from the character
        BRp AT_LEAST_A          ; branch if > '@', i.e., >= 'A'
NON_ALPHA
        LDR R6,R0,#0            ; load the non-alpha count
        ADD R6,R6,#1            ; add one to it
        STR R6,R0,#0            ; store the new non-alpha count
        BRnzp GET_NEXT          ; branch to end of conditional structure
AT_LEAST_A
        ADD R6,R2,R4            ; compare with 'Z'
        BRp MORE_THAN_Z         ; branch if > 'Z'

; note that we no longer need the current character
; so we can reuse R2 for the pointer to the correct
; histogram entry for incrementing
ALPHA   ADD R2,R2,R0            ; point to correct histogram entry
        LDR R6,R2,#0            ; load the count
        ADD R6,R6,#1            ; add one to it
        STR R6,R2,#0            ; store the new count
        BRnzp GET_NEXT          ; branch to end of conditional structure

; subtracting as below yields the original character minus '''
MORE_THAN_Z
        ADD R2,R2,R5            ; subtract ''' - '@' from the character
        BRnz NON_ALPHA          ; if <= ''', i.e., < 'a', go increment non-alpha
        ADD R6,R2,R4            ; compare with 'z'
        BRnz ALPHA              ; if <= 'z', go increment alpha count
        BRnzp NON_ALPHA         ; otherwise, go increment non-alpha

GET_NEXT
        ADD R1,R1,#1            ; point to next character in string
        BRnzp COUNTLOOP         ; go to start of counting loop

DONE    HALT                    ; done

; the data needed by the program
NUM_BINS        .FILL #27       ; 27 loop iterations
NEG_AT          .FILL xFFC0     ; the additive inverse of ASCII '@'
AT_MIN_Z        .FILL xFFE6     ; the difference between ASCII '@' and 'Z'
AT_MIN_BQ       .FILL xFFE0     ; the difference between ASCII '@' and '''
STR_START       .FILL STRING    ; string stored below for simplicity
HIST            .BLKW #27       ; space to store the histogram

STRING  .STRINGZ "This is a test of the counting frequency code.  AbCd...WxYz."

        .END
```

```
; read two numbers using a subroutine and store them to memory

        .ORIG x3000             ; starting address is x3000

        JSR READNUM             ; read two numbers and store them
        ST R0,NUM1
        JSR READNUM
        ST R0,NUM2
        HALT

; subroutine developed as an extension of the
; earlier binary code

; read a decimal number from the keyboard,
; convert it from ASCII to 2's complement, and
; return it in R0.  If any non-numeric character
; is pressed, or the number overflows, print an
; error message and start over.

; R0 holds the value of the last key pressed
; R1 holds the current value of the number being input
; R2 holds the additive inverse of ASCII '0' (0xFFD0)
; R3 is used as a temporary register

READNUM                         ; the subroutine to read a number

        ST R7,SAVE_R7           ; TRAP overwrites R7, so must save
        ST R3,SAVE_R3           ; callee saves register values
        ST R2,SAVE_R2
        ST R1,SAVE_R1

        LD R2,NEG_0             ; put the value -x30 in R2
        AND R1,R1,#0            ; clear the current value

READ_LOOP
        GETC                    ; read a character
        OUT                     ; echo it back to monitor
        ADD R3,R0,#-10          ; compare with ENTER
        BRz DONE                ; if ENTER pressed, done

        ADD R0,R0,R2            ; subtract x30 from R0
        BRn BAD_KEY             ; smaller than '0' means error
        ADD R3,R0,#-10          ; check if > '9'
        BRzp BAD_KEY            ; greater than '9' means error
        ADD R3,R1,R1            ; sequence of adds multiplies R1 by 10
        BRn OVERFLOW            ; overflow, but not really necessary here
        ADD R3,R3,R3
        BRn OVERFLOW            ; overflow, but not really necessary here
        ADD R1,R1,R3
        BRn OVERFLOW            ; overflow
        ADD R1,R1,R1
        BRn OVERFLOW            ; overflow
        ADD R1,R1,R0            ; finally, add in new digit
        BRn OVERFLOW            ; overflow
        BRnzp READ_LOOP         ; get another digit

DONE
        ADD R0,R1,#0            ; move R1 into R0
        LD R1,SAVE_R1           ; restore register values for caller
        LD R2,SAVE_R2
        LD R3,SAVE_R3
        LD R7,SAVE_R7

        RET                     ; return

; print error message: "non-digit pressed"
BAD_KEY
        LEA R0,BK_MSG           ; point R0 to the start of the string
PRINT_ERR
        PUTS                    ; the trap that you're not allowed to use in MP2
        AND R1,R1,#0            ; reset current value
        BRnzp READ_LOOP         ; try reading again

; print error message: "overflow"
OVERFLOW
        LEA R0,OF_MSG           ; point R0 to the start of the string
        BRnzp PRINT_ERR

SAVE_R1 .BLKW 1         ; storage for saved register values
SAVE_R2 .BLKW 1
SAVE_R3 .BLKW 1
SAVE_R7 .BLKW 1
NEG_0   .FILL xFFD0     ; the additive inverse of ASCII '0'
NUM1    .BLKW 1         ; storage for the results
NUM2    .BLKW 1

; error messages.  The sequence \n means newline and is replaced
; with a single ASCII linefeed character (#10).  Similar sequences
; include \r for #13 (carriage return), \t for #9 (TAB), \\ for
; backslash, etc.
BK_MSG  .STRINGZ "\nnon-digit pressed\n"
OF_MSG  .STRINGZ "\noverflow\n"

        .END
```

```
/*
 * ECE190 Fall 2005
 *
 * Program name: dump_memory.c, a procedure to print the contents of memory
 *
 * Description: This function uses the LC-3 simulator's read_memory function
 *              to print the contents of memory from a starting address to
 *              an ending address (both given as parameters).
 */

#include <stdio.h>        /* Include C's standard I/O header file.           */

#include "lc3sim.h"       /* Include the header file for the LC-3 simulator. */

/*
 * Function: dump_memory
 * Description: print a range of LC-3 memory in hexadecimal
 * Parameters: the starting and ending addresses; the range printed does
 *             not include the ending address, but stops at the previous
 *             location
 * Return Value: nothing
 */

void
dump_memory (int addr_s, int addr_e)
{
    int start;    /* First address of line being printed.                   */
    int addr;     /* Address being printed.                                 */
    int index;    /* Index of address being printed on current line (0-11). */

    /* Simplify code by not using modular arithmetic for address range.
       If given range wraps around 0, replace the ending address with
       one that is larger than the starting address, but equivalent
       modulo the size of the memory space. */
    if (addr_s >= addr_e)
        addr_e += 0x10000;

    /* Loop 12 addresses at a time.  Starting addresses for lines must
       be multiples of 12. */
    for (start = (addr_s / 12) * 12; start < addr_e; start = start + 12) {

        /* Print an address at the start of each line.  Since start
           is not necessarily in the valid range 0 to 0xFFFF, we first
           AND it with 0xFFFF. */
        printf ("%04X: ", start & 0xFFFF);

        /* This loop goes over all 12 addresses in the current line.
           The index variable runs from 0 to 11 (counts to 12), while
           the addr variable tracks the address currently being printed. */
        for (index = 0, addr = start; index < 12; index++, addr++) {

            /* We only print an address if it lies within the requested
               range.  The value in memory is returned by the call to
               read_memory.  If an address is not within the requested
               range, we print blank space for printing alignment purposes. */
            if (addr >= addr_s && addr < addr_e)
                printf ("%04X ", read_memory (addr & 0xFFFF));
            else
                printf ("     ");
        }

        /* End the printed line. */
        puts ("");
    }
}
```

```
/* a sample of the output
 * 01F8:                                           E002 F022 F025 000A
 * 0204: 0057 0065 006C 0063 006F 006D 0065 0020 0074 006F 0020 0074
 * 0210: 0068 0065 0020 004C 0043 002D 0033 0020 0073 0069 006D 0075
 * 021C: 006C 0061 0074 006F 0072
*/
```

```
/*
 * ECE190 Fall 2005
 *
 * Program name: factorial.c, a factorial calculator
 *
 * Description: This program asks for an integer, then calculates and
 *              prints the factorial of the number.
 */

/* The following two lines are preprocessor directives. */
#include <stdio.h>        /* Include C's standard I/O header file.  */
#define STOP    1         /* Stop when we reach one. */

/*
 * Function: main
 * Description: prompt player for name and bet, then play game and announce
 *              the outcome
 * Parameters: none (we're ignoring the standard ones to main for now)
 * Return Value: 0, which by convention indicates success
 */

int
main ()
{
    /* variable declarations */
    int number;      /* number given by user       */
    int factorial;   /* factorial of user's number */

    /* Print a welcome message, followed by a blank line. */
    printf (">--- Welcome to the factorial calculator! ---<\n\n");

    /* Ask for and read the player's bet into a variable. */
    printf ("What factorial shall I calculate for you today? ");
    scanf ("%d", &number);

    /* Calculate and report the answer (no overflow checking!). */
    for (factorial = number; number > STOP; number = number - 1)
        factorial = factorial * (number - 1);
    printf ("\nThe factorial is %d.\n", factorial);

    /* Program finished successfully. */
    return 0;
}
```

```
/*
 * ECE190 Fall 2005
 *
 * Program name: translate.c, a number translator
 *
 * Description: This program asks for a decimal number, then prints out
 *              the absolute value of the number in hexadecimal form.
 */

#include <stdio.h>        /* Include C's standard I/O header file.  */

int the_number;   /* the number -- no good reason to be a global variable
                     other than to serve the purpose of the example */

/*
 * Function: find_abs
 * Description: convert an integer to its absolute value
 * Parameters: the number to convert
 * Return Value: the absolute value of the number passed
 */

int
find_abs (int num)
{
    int abs_value;

    if (num >= 0) {
        /* Don't change positive numbers. */
        abs_value = num;
    } else {
        /* Negative of negative number is the absolute value. */
        abs_value = -num;
    }

    return abs_value;
}

/*
 * Function: main
 * Description: prompt user for a decimal number, then print absolute value
 *              in hexadecimal
 * Parameters: none (we're ignoring the standard ones to main for now)
 * Return Value: 0, which by convention indicates success
 */

int
main ()
{
    /* no local variable declarations */

    /* Ask for and read the player's bet into a variable. */
    printf ("Please enter a decimal number: ");
    scanf ("%d", &the_number);

    /* Find the absolute value. */
    the_number = find_abs (the_number);

    /* Print the answer. */
    printf ("The absolute value in hexadecimal is %x.\n", the_number);

    /* Program finished successfully. */
    return 0;
}
```

```
/*
 * insertion sort -- performs an assertion sort on an array of integers
 *    inputs: values -- a pointer to an array of integers
 *            num_vals -- the number of values in the array
 *    outputs: values -- returned in sorted order
 *    returns: nothing, but changes array in place
 *
 * NOTE: does nothing if num_vals < 2
 *
 */

void
insertion_sort (int values[], int num_vals)
{
    int sorted;         /* outer loop index; number of values sorted       */
    int current;        /* current value being placed into sorted subarray */
    int index;          /* inner loop index for placing current value      */

    /* Checks on input parameters should go here.
       What kinds of things might you check? */

    /* We start with a subarray of length 1 already sorted, so
       we need iterations to sort each larger subarray from length 2
       up to the full length of the array. */
    for (sorted = 2; sorted <= num_vals; sorted++) {

        /* Keep track of the value being moved into position. */
        current = values[sorted - 1];

        /* Move other array entries aside to make room for "current." */
        for (index = sorted - 1; index > 0; index--) {

            /* Check the order of "current" against the value before
               that at index.  If it's still smaller, move the value
               and continue the loop.  Otherwise, we've found the place
               to which we must move "current." */
            if (current < values[index - 1])
                values[index] = values[index - 1];
            else
                break;
        }

        /* Store current in the right place. */
        values[index] = current;
    }

    /* No return value. */
}
```

Input and Output in Unix and C

Basic Abstractions

Unix and C support a unified abstraction for input and output (I/O) known as *file descriptors*. Input and output from everything ranging from devices to files to network connections uses the same abstraction. In particular, the operating system maintains an array of structures with information about I/O channels, with each channel occupying one place in this table. The array index at which a given channel appears can thus be used to locate the corresponding information within the table, and a file descriptor is nothing more than an integer. Most operating systems limit the size of the table to 1,024 entries by default, so descriptors are typically in the range 0 to 1,023. A diagram appears in Figure 1.

Notice that the first three entries in the array of I/O channels are occupied by the "standard" I/O channels for a program. These channels are set up by the operating system before a program starts. If you execute a program by itself from within a shell, input comes from the keyboard, and output (both normal and error output) goes to the monitor. However, these defaults are easily overridden. In fact, you probably use a graphical window manager when working, in which case the output from your programs does not go to the monitor, but instead to the window manager for display in the window in which your program was started. In the original scheme for providing network services on Unix machines, known as `inetd` (for "Internet Daemon"), the operating system started programs in response to incoming network connections, replacing the standard input and output channels for the new program with the incoming connection. Network services could thus be written and tested easily from any standard shell, then simply redirected to accept input and send output across the network when they were ready.

The information in the I/O channel structure allows the operating system to differentiate between the different types of I/O channels as necessary, and this information can be accessed and manipulated by a wide array of generic and special-purpose system calls, but most of these are beyond the scope of our class. We will consider only a certain class of fairly general-purpose calls in this discussion.

In particular, we focus on the calls that use streams. A *stream* is a logical array of bytes that flow from one place to another through an I/O channel. Some types of I/O channels do not fit readily into the stream model; some network protocols, for example, break data into packets; most devices have control/status registers as well as data registers, and the access pattern necessary to control these devices is generally not the simple linear progression that the stream provides. Channels that can fit into the stream model include input from a keyboard, output to a monitor, files on a disk[1], and certain types of network protocols.

The stream abstraction also provides support for buffering part of the stream in order to improve performance. Files on disk are stored in blocks of four or eight kB, but can take tens of milliseconds to retrieve (tens of millions of processor clock cycles). If this delay were incurred for each byte read from a file, a program would run quite slowly. Even interacting with the operating system through a system call is relatively slow, however, often requiring tens or hundreds of thousands of cycles or more. Buffering reduces the number of interactions with the operating system by bringing data into the program in large blocks and using C library functions to handle most of the actual data transfer for a stream.

Buffering also helps to simplify the implementation of certain expected behaviors. For example, reading from the keyboard typically returns nothing until the user presses the RETURN/ENTER key. If this buffering is turned off, every application must process BACKSPACE, since a keystroke delivered to the application cannot otherwise be taken back. With the default buffering strategy, BACKSPACE is handled by keyboard-processing code, and application programs see only lines that have been completed by pressing RETURN.

[1] The disk itself is a block device, but the filesystem serves to translate this abstraction into a stream for any given file.

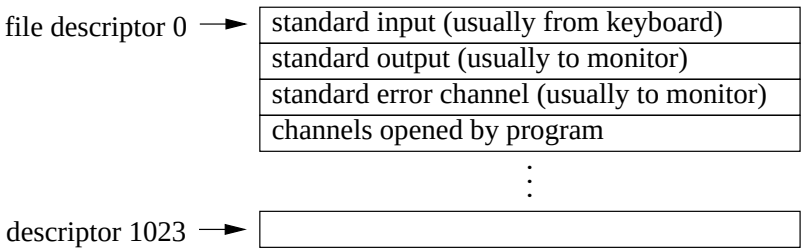


Figure 1: The array of I/O channel structures maintained for a program by the operating system. The array index for a channel's structure is used to identify the channel, and is passed around as a small integer known as a file descriptor.

In C, a stream is represented by a pointer to a structure containing information about the kind of buffering desired as well as the file descriptor to be used by the stream. The structure is written FILE (all capitals), and is allocated by library code, allowing a program to simply declare pointers to these structures in order to manipulate streams. For example:

```
FILE* my_file;
```

declares a variable to refer to a stream.

Default and New Streams

The three default file descriptors created by the operating system are also associated with streams before a program begins to execute. The input stream is named `stdin`, the output stream `stdout`, and the error stream `stderr`. Like the file descriptors, `stdin` can only be read, not written, while `stdout` and `stderr` can only be written, not read.

Other streams can be created with several functions, the simplest of which are those used to access the files stored on disk. As you should already be aware, Unix uses a hierarchical file system in which the names of files can consist of sequence of directory names followed by a name for the file within its local directory.[2] Directories correspond exactly to the folders used by graphical file system browsers. The file names used within a program are the same as those used within a Unix shell (a shell is just a program, after all); also like a shell, each program has a notion of a current directory, so files with no directory names in front of them refer to those files in the directory in which the program was started (assuming that the program has not changed its current directory).

The function below opens a file:

```
FILE* fopen (const char* file_name, const char* mode);
```

The first argument is a string containing the name of the file. The second argument is a string specifying what types of operations are to be performed on the new stream. The function returns NULL if the open fails, and the `perror` function can be called to print a human-readable error message in this case, such as "file not found" or "file not readable." If the file is opened successfully, the function returns a new stream, which should eventually be closed with `fclose`, as described below.

The mode consists of a letter followed by an optional plus sign. If the plus sign is included, the file is opened for both reading and writing. If the plus sign is not included, the file is opened for either reading or writing (depending on the letter, as we discuss shortly), but not both. The letter "r" is used to open an existing file; an error is returned if the file does not exist, and the file is opened for reading if no plus sign is included in the mode. The letter "w" creates a new file for writing, first deleting an existing file of the given name if it exists, and allowing only writing if no plus sign is included in the mode string. Finally, the letter "a" is used to write to the end of an existing file; in this case, if no file exists yet, a new one is created. With the append option, the default mode is writing, and writing begins at the end of the file.

[2] It may interest you to know that Unix "files" can actually be other channels in disguise, including everything from devices to network connections to channels to existing programs.

Finally, the letter "b" can optionally be included after the first letter (or the plus sign) with any mode, but has no effect. On some older systems, the "b" signified that binary data were being stored in the file, and that certain standard translations on ASCII text should not be performed to avoid corrupting the binary data. However, these translations are for the most part obsolete, and are never performed on Unix platforms.

When a program is done using a stream, it should close the stream to free up the I/O channel resource, of which a limited number are available for each program. For this purpose, use the following function:

```
int fclose (FILE* stream);
```

This function takes a stream and attempts to close it. If no errors occurred in accesses to the stream, the function returns 0. If an error occurred, it returns EOF (-1).

All streams are closed when a program finishes execution, but it is a bad habit to rely on program termination rather than using the `fclose` function. While you are unlikely to see any difference in this class, consider the impact of not closing network connections in a web server: after the first 1,021 connections, the server begins to drop all requests, as it has no free I/O channels.

Character by Character I/O

Our description of I/O functions now parallels the textbook, but we will cover a few more functions than are described in the text. We will also differentiate between the functions that operate on any stream and the shortcut functions that operate on `stdin` and `stdout`.

The functions below support reading and writing single characters to streams:

```
int fgetc (FILE* stream);  /* read one character (a function) */
int getc (FILE* stream);   /* read one character (a macro)    */
int fputc (int c, FILE* stream);  /* write one character (a function) */
int putc (int c, FILE* stream);   /* write one character (a macro)    */
```

The first two functions return the ASCII character read from the stream (converted into an integer), or EOF (-1) if the read attempt failed. These calls by default *block* until input is available. That is, a user's not having typed a character yet does not cause the call to fail. Instead, the operating system puts the program to sleep until a character is typed. Recall that the same thing occurs in the LC-3 system calls, which wait for a character to be available rather than returning a failure indicator. Failure thus indicates conditions such as reaching the end of an input file.

The difference between `fgetc` and `getc` is that the first is a function and creates a function call in the assembly code generated by a compiler, while the second is a preprocessor macro-operation that copies the necessary code in place of the call before compilation. In older machines, these functions allowed a tradeoff between superior performance (`getc`) and reduced program size (`fgetc`). In more modern machines, you're probably better off using `fgetc`.

The `fputc` and `putc` functions write a single character (specified by the first argument) to a stream. Although an integer is passed, only a single unsigned character is actually written. These functions both return the value written if successful, or EOF (-1) on failure.

Shortcut functions, both based the macro version of the functions above, are available for reading and writing single characters to `stdin` and `stdout`. These functions are declared as follows:

```
int getchar (void);   /* read one character from stdin (a macro) */
int putchar (int c);  /* write one character to stdout (a macro) */
```

The return values and argument are the same as for the corresponding previous functions, but the values of the stream parameters are implicit in each case.

Reading and Writing Lines

For text files, it is usually most convenient to work with a line at a time, reading each line into an array of characters and treating it as a string, or writing each line into an array of characters before sending it off to the file. The two functions used for these purposes with streams are:

```
char* fgets (char* s, int n, FILE* stream); /* read one line  */
int fputs (const char* s, FILE* stream);    /* write a string */
```

The `fgets` function reads one line of text from a stream into the array of characters given by its first argument. The second argument specifies the size of the array, and `fgets` also stops reading if it runs out of room. The function always appends an ASCII NUL character to terminate the string, so it reads at most `n`-1 characters from the stream. If a full line is read, the linefeed (LF, or "\n" in C) character is also written into the specified array of characters. The function returns its first argument when successful, and NULL when no further data are available from the stream (or some other error occurs).

The `fputs` function writes a string to a stream. No additional characters are sent, so the string must include a linefeed character at the end if it is to appear as a line in a text file. The function returns the number of characters written or EOF on failure.

Shortcut functions are available for both of these functions, but the shortcut for reading a line does not allow the caller to specify the length of the array, and thus poses a security hazard. **You should never use the `gets` function!** The majority of network attacks use strategies based on exactly this type of function, so you should simply never use it. The shortcut function used to write a string to `stdout` is declared as:

```
int puts (char* s);  /* write one line to stdout */
```

This function differs from `fputs` not only in implicitly writing to `stdout`, but also in that it also writes a linefeed character to `stdout` after writing the string passed. A string meant to become a single line of output must therefore NOT include a linefeed character at the end. The return value meanings are the same as for `fputs`.

Formatted I/O

You have already seen the `scanf` and `printf` functions used to translate between the ASCII text representing human-readable text and the binary forms understood by a computer. These two functions are simply the shortcut forms of the more general functions for reading and writing formatted data to streams:

```
int fscanf (FILE* stream, const char* fmt, ...);
int fprintf (FILE* stream, const char* fmt, ...);
```

The only difference between these functions and those already familiar to you is the need to include the stream as the first argument. With `scanf`, the `stdin` stream is used implicitly, while `printf` implicitly writes to `stdout`. Note that any information that should instead be delivered to `stderr` must use the more general form.

A third form of these functions is also useful, particularly in combination with the functions described in the previous section for reading lines and writing strings to streams:

```
int sscanf (const char* s, const char* fmt, ...);
int sprintf (char* s, const char* fmt, ...);
```

These functions read and write formatted data to strings (arrays of characters). Note that the printing function does not allow the caller to specify the size of the array, and can thus be attacked in certain cases. Be careful to allocate enough space for a printed string; you may prefer to use the `snprintf` function instead, but I'm not sure that it is required by the ANSI standard, thus you may need to write this function yourself for fully portable code.

Binary I/O

The last set of functions that we cover allow you to send binary data, such as the contents of an array, directly to and from a stream. While Unix does not perform any translations on bytes, none of the preceding functions allow you to transfer arbitrary sequences of bytes, a fact often overlooked by novice programmers. Pretending that an array of integers is a string does not generally work, for example, as any zero byte in the array ends the "string."

Before describing the functions, we need to explain some possibly new types. To reflect growth in file and memory sizes, ANSI C actually uses a separate type to specify sizes in bytes, allowing this type to grow (to 64 bits, for example) without necessarily growing the size of an integer. This type is called `size_t`, but you can think of it as unsigned 32-bit integer, and on most systems it is just that.

A second type, a pointer to a null type (`void*`), is used to allow automatic conversions to and from any other pointer type. In particular, if the type of a parameter to a function is `void*`, any pointer type can be passed without causing the compiler to repond with warnings or errors

The functions are declared as follows:

```
size_t fread (void* ptr, size_t size, size_t n_items, FILE* stream);
size_t fwrite (const void* ptr, size_t size, size_t n_items, FILE* stream);
```

The list of arguments to both functions is essentially the same. The first argument is a pointer to the memory to be filled with bytes from the stream (in the case of `fread`) or from which bytes should be written to the stream (in the case of `fwrite`). The second argument specifies the size of items to be read or written, typically using the `sizeof()` built-in function, which is evaluated at compile time to the size of a type (or variable's type) in bytes. The third argument specifies how many such items should be read or written, and the last argument gives the stream. Both functions return the number of items (NOT the number of bytes) read or written to the stream. Typically, a return value equal to the third argument indicates that the call was completely successful, but partial success or total failure is also possible, such as when a disk fills up or a user exceeds a disk quota.

No shortcut functions are available, as binary data are not normally delivered by or to human users.

```
/*
 * ECE190 Fall 2005
 *
 * Program name: line_sort.c, a sorting program
 *
 * Description: This program alphabetically sorts lines from stdin.
 *              Lines are stored using dynamically allocated memory.
 */

#include <stdio.h>        /* Include C's standard I/O header file.  */
#include <string.h>       /* Include C's string library.            */

static const int max_num_lines = 5000; /* limit on number of lines */
static const int max_line_len  = 500;  /* limit on line length */

/* My favorite exit condition definitions. */
enum {
    EXIT_SUCCEED  = 0,
    EXIT_FAIL     = 1,
    EXIT_BAD_ARGS = 2,
    EXIT_PANIC    = 3
};


/* function declarations */

/* read lines from stdin into an array; returns number of lines read */
static int read_lines (unsigned char* lines[], int max_lines);

/* sort strings in an array alphabetically using insertion sort */
static void sort_lines (unsigned char* lines[], int n_lines);

/* print an array of strings in order to stdout */
static void print_lines (unsigned char* const lines[], int n_lines);


/*
 * Function: main
 * Description: read stdin one line at a time, copying the lines
 *              into dynamically allocated memory, then sort and
 *              print the lines
 * Parameters: argc -- the number of arguments, including the executable name
 *             argv -- an array of strings containing each argument
 *             argc must equal 1; no other arguments are allowed
 * Return Value: EXIT_SUCCEED for success
 *               EXIT_BAD_ARGS if the wrong number of arguments are given
 */

int
main (int argc, char* argv[])
{
    unsigned char* lines[max_num_lines];        /* array of lines  */
    int num_lines;                              /* number of lines */

    /* Program must receive exactly one argument. */
    if (argc != 1) {
        /* Print an error message.  argv[0] is the executable name. */
        fprintf (stderr, "syntax: %s\n", argv[0]);
        return EXIT_BAD_ARGS;
    }

    /* Read, sort, and print lines from stdin. */
    num_lines = read_lines (lines, max_num_lines);
    sort_lines (lines, num_lines);
    print_lines (lines, num_lines);

    /* Program finished successfully. */
    return EXIT_SUCCEED;
}


/*
 * read_lines -- reads lines from stdin into an array
 *    inputs: lines -- an (empty) array of strings
 *            max_lines -- the size of the array
 *    outputs: nothing
 *    returns: number of lines read
 */

static int
read_lines (unsigned char* lines[], int max_lines)
{
    unsigned char buf[max_line_len + 1];        /* holds current line */
    int num_lines;                              /* number of lines    */

    /* Initialize the line count. */
    num_lines = 0;

    /* Read lines until we find the end of the input. */
    while (fgets (buf, max_line_len + 1, stdin) != NULL) {

        /* Are more lines available than we can read?  If so, print
           a warning message and stop reading. */
        if (num_lines == max_lines) {
            fprintf (stderr, "WARNING: Cannot sort more than %d lines.\n",
                     max_lines);
            break;
        }

        /* Make duplicate copy of line just read in heap memory, then
           store pointer to new copy in lines array. */
        lines[num_lines++] = strdup (buf);
    }

    /* Return number of lines read to caller. */
    return num_lines;
}


/*
 * sort_lines -- performs an insertion sort on an array of integers
 *    inputs: lines -- an array of strings
 *            n_lines -- the number of lines in the array
 *    outputs: lines -- returned in sorted order
 *    returns: nothing, but changes array in place
 *
 * NOTE: does nothing if n_lines < 2
 *
 */

static void
sort_lines (unsigned char* lines[], int n_lines)
{
    int sorted;         /* outer loop index; number of lines sorted       */
    char* current;      /* current line being placed into sorted subarray */
    int index;          /* inner loop index for placing current line      */

    /* We start with a subarray of length 1 already sorted, so
       we need iterations to sort each larger subarray from length 2
       up to the full length of the array. */
```

```
    for (sorted = 2; sorted <= n_lines; sorted++) {

        /* Keep track of the line being moved into position. */
        current = lines[sorted - 1];

        /* Move other array entries aside to make room for "current." */
        for (index = sorted - 1; index > 0; index--) {

            /* Check the order of "current" against the line before
               that at index.  If it's still smaller, move the line
               and continue the loop.  Otherwise, we've found the place
               to which we must move "current." */
            if (strcmp (current, lines[index - 1]) < 0)
                lines[index] = lines[index - 1];
            else
                break;
        }

        /* Store current in the right place. */
        lines[index] = current;
    }

    /* No return value. */
}


/*
 * print_lines -- print an array of strings (lines)
 *    inputs: lines -- an array of strings
 *            n_lines -- the number of lines in the array
 *    outputs: nothing
 *    returns: nothing, but prints all lines in order to stdout
 */

static void
print_lines (unsigned char* const lines[], int n_lines)
{
    int index; /* loop index for printing */

    /* Print all lines in order. */
    for (index = 0; index < n_lines; index++)
        fputs (lines[index], stdout);

    /* No return value. */
}
```

```
/*
 * ECE190 Fall 2005
 *
 * Program name: unique_count.c, a unique line counting program
 *
 * Description: This program reads lines from stdin, merges identical
 *              lines, and prints each line with a number prefix
 *              indicating how many times the same line appeared
 *              consecutively in the input.
 */

#include <stdio.h>        /* Include C's standard I/O header file.  */

static const int max_word_len = 500;  /* limit on word length */

/* My favorite exit condition definitions. */
enum {
    EXIT_SUCCEED  = 0,
    EXIT_FAIL     = 1,
    EXIT_BAD_ARGS = 2,
    EXIT_PANIC    = 3
};

/*
 * Function: main
 * Description: read lines from stdin, merge duplicate consecutive lines,
 *              and print lines prefixed by their multiplicities in the
 *              input (consecutive counts only; appearance elsewhere is
 *              ignored)
 * Parameters: argc -- the number of arguments, including the executable name
 *             argv -- an array of strings containing each argument
 *             argc must equal 1; no additional argument are allowed
 * Return Value: EXIT_SUCCEED for success
 *               EXIT_FAIL if the input contains no lines
 *               EXIT_BAD_ARGS if the wrong number of arguments are given
 */

int
main (int argc, char* argv[])
{
    unsigned char buf1[max_word_len + 1];       /* a line                    */
    unsigned char buf2[max_word_len + 1];       /* a second line             */
    unsigned char* last_line;                   /* points to last line       */
    unsigned char* cur_line;                    /* points to current line    */
    unsigned char* tmp;                         /* a temporary for swapping  */
    int count;                                  /* multiplicity of last_line */

    /* Program must receive exactly one argument. */
    if (argc != 1) {
        /* Print an error message.  argv[0] is the executable name. */
        fprintf (stderr, "syntax: %s\n", argv[0]);
        return EXIT_BAD_ARGS;
    }

    /* Read the first line. */
    if (fgets (buf1, max_word_len + 1, stdin) == NULL) {
        fputs ("Could not read any lines!\n", stderr);
        return EXIT_FAIL;
    }

    /* Initialize the double buffering scheme based on the first line's
       residing in buf1. */
    last_line = buf1;
    count     = 1;
    cur_line  = buf2;

    /* Read lines until we find the end of the input. */
    while (fgets (cur_line, max_word_len + 1, stdin) != NULL) {

        /* Check for duplication. */
        if (strcmp (cur_line, last_line) == 0) {
            count++;
            continue;
        }

        /* Print last line (it already includes a carriage return). */
        printf ("%5d %s", count, last_line);

        /* Switch buffering for lines, and reset count. */
        tmp = cur_line;
        cur_line = last_line;
        last_line = tmp;
        count = 1;
    }

    /* Print final line (it already includes a carriage return). */
    printf ("%5d %s", count, last_line);

    /* Program finished successfully. */
    return EXIT_SUCCEED;
}
```

```
/*
 * ECE190 Fall 2005
 *
 * Program name: word_split.c, an English word splitting program
 *
 * Description: This program splits its input into a list of lower-case
 *              words, with one word per line.  Words are defined as
 *              contiguous sequences of alphabetic characters, hyphens,
 *              and apostrophes.  Words must begin with an alphabetic
 *              character.  All other characters are discarded.
 */

#include <stdio.h>        /* Include C's standard I/O header file.  */

static const int max_word_len = 500;  /* limit on word length */

/* My favorite exit condition definitions. */
enum {
    EXIT_SUCCEED  = 0,
    EXIT_FAIL     = 1,
    EXIT_BAD_ARGS = 2,
    EXIT_PANIC    = 3
};

/*
 * Function: main
 * Description: read a file one character at a time, break input into
 *              lower-case words (alphabetic, hyphens, or apostrophes),
 *              and print words found on separate lines without eliminating
 *              duplicates.  Hyphens and apostrophes are not allowed to
 *              start words.
 * Parameters: argc -- the number of arguments, including the executable name
 *             argv -- an array of strings containing each argument
 *             argc must equal 2, and the second argument is the file name
 *                from which words are read
 * Return Value: EXIT_SUCCEED for success
 *               EXIT_FAIL if file cannot be opened
 *               EXIT_BAD_ARGS if the wrong number of arguments are given
 */

int
main (int argc, char* argv[])
{
    FILE* in_file;                          /* input file pointer     */
    unsigned char buf[max_word_len + 1];    /* holds current word     */
    unsigned char* write;                   /* end of current word    */
    int word_len;                           /* length of current word */
    int a_char;                             /* last character read    */

    /* Program must receive exactly two arguments. */
    if (argc != 2) {
        /* Print an error message.  argv[0] is the executable name. */
        fprintf (stderr, "syntax: %s <file name>\n", argv[0]);
        return EXIT_BAD_ARGS;
    }

    /* Open the file for reading. */
    in_file = fopen (argv[1], "r");
    if (in_file == NULL) {
        /* fopen failed: print an error message to stderr. */
        perror ("open file");
        return EXIT_FAIL;
    }

    /* Initialize the word writing variable to point to the start of
       the word buffer. */
    write = buf;
    word_len = 0;

    /* Read characters until we find the end of the input. */
    while ((a_char = getc (in_file)) != EOF) {

        /* If necessary, change input character to lower case. */
        if (a_char >= 'A' && a_char <= 'Z')
            a_char = a_char - 'A' + 'a';

        /* Can character be part of a word? */
        if ((a_char >= 'a' && a_char <= 'z') ||
            (word_len > 0 && (a_char == '-' || a_char == '\''))) {

            /* Write the character into our word buffer and increment
               the pointer and counter. */
            *write++ = a_char;
            word_len++;

            /* Do we still have room in the buffer?  If so, read
               another character (skip to next loop iteration). */
            if (word_len < max_word_len)
                continue;
        } else {
            /* Invalid character read.  Is there a word that needs
               to be written out?  If not, skip to next character. */
            if (word_len == 0)
                continue;
        }

        /* Write out the current word, then reset the buffer pointer
           and character count. */
        *write = 0;
        puts (buf);
        write = buf;
        word_len = 0;
    }

    /* Any last words? */
    if (word_len > 0) {
        *write = 0;
        puts (buf);
    }

    /* Close the input file, ignoring any errors. */
    fclose (in_file);

    /* Program finished successfully. */
    return EXIT_SUCCEED;
}
```

```
/*                                                                  tab:8
 *
 * mem199.h - header file for ECE199SJP's simple memory management package
 *
 * "Copyright (c) 2003 by Steven S. Lumetta."
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice and the following
 * two paragraphs appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE AUTHOR OR THE UNIVERSITY OF ILLINOIS BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
 * DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION,
 * EVEN IF THE AUTHOR AND/OR THE UNIVERSITY OF ILLINOIS HAS BEEN ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE AUTHOR AND THE UNIVERSITY OF ILLINOIS SPECIFICALLY DISCLAIM ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE
 * PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND NEITHER THE AUTHOR NOR
 * THE UNIVERSITY OF ILLINOIS HAS ANY OBLIGATION TO PROVIDE MAINTENANCE,
 * SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Author:          Steve Lumetta
 * Version:         1
 * Creation Date:   4 December 2003
 * Filename:        mem199.h
 * History:
 *      SL      1       4 December 2003
 *              First written.
 */

#if !defined(_MEM199_H)
#define _MEM199_H

/*
   These constants define the limitations on memory allocation with
   the package.  Nothing larger can be compiled.  Note that the code for
   the package must be recompiled if these numbers are changed.
*/

#define MEM199_MAX_ALLOC_LOG  20
#define MEM199_MAX_ALLOC      (1UL << MEM199_MAX_ALLOC_LOG)

/*
   mem199_allocate

   Allocates n_bytes and returns a pointer to the new memory.  If no memory
   is available, or if 0 bytes are requested, returns NULL.  Note that the
   new memory may contain arbitrary values.
*/
void* mem199_allocate (unsigned n_bytes);

/*
   mem199_allocate_and_zero

   Allocates n_bytes, fills the new memory with zeroes, and returns a
   pointer to the new memory.  If no memory is available, or if 0 bytes
   are requested, returns NULL.
*/
void* mem199_allocate_and_zero (unsigned n_bytes);

/*
   mem199_reallocate

   Attempts to change the size of a previously allocated block of memory.
   The parameters passed are a pointer to the pointer to the old block
   (possibly NULL, if no previous block existed) and the new desired size.
   If possible, a new block of the appropriate size is allocated, any
   data in the old block are copied into the new block, the old block
   is freed, the pointer is changed, and 0 is returned.  If the allocation
   of a new block fails, the pointer to the old block is not changed,
   the old block (if it existed) is not freed, and -1 is returned.
*/
int mem199_reallocate (void** ptr_to_ptr, unsigned n_bytes);

/*
    mem199_free

    Returns control of a block of memory to the memory management system.
    The block should not be accessed after a call to mem199_free.  The
    block may be returned by a successive call to any of the allocation
    functions.
*/
void mem199_free (void* ptr);

#endif /* !defined(_MEM199_H) */
```

```
/*                                                                tab:8
 *
 * mem199.c - a simple memory management package for ECE199SJP
 *
 * "Copyright (c) 2003 by Steven S. Lumetta."
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice and the following
 * two paragraphs appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE AUTHOR OR THE UNIVERSITY OF ILLINOIS BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
 * DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION,
 * EVEN IF THE AUTHOR AND/OR THE UNIVERSITY OF ILLINOIS HAS BEEN ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE AUTHOR AND THE UNIVERSITY OF ILLINOIS SPECIFICALLY DISCLAIM ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE
 * PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND NEITHER THE AUTHOR NOR
 * THE UNIVERSITY OF ILLINOIS HAS ANY OBLIGATION TO PROVIDE MAINTENANCE,
 * SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Author:          Steve Lumetta
 * Version:         1
 * Creation Date:   4 December 2003
 * Filename:        mem199.c
 * History:
 *      SL      1       4 December 2003
 *              First written.
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "mem199.h"


/*
   This memory manager allocates blocks in sizes of powers of two,
   allowing reasonably efficient reuse of freed blocks.  As with almost
   all memory managers, management information is held in a header
   preceding the region allocated to the caller.  For this implementation,
   we need only the block size in the header, which allows free to
   place the block into the correct bin.  We actually store the index
   of the bin in our array of bins, which is equivalent to the log_2 of
   the block size.
*/


/*
   The memory block header structure, stored at the front of each block
   of memory.  It contains the size of the block and a pointer allowing
   us to chain free blocks together into a list.
*/
typedef struct mem_block_t mem_block_t;
struct mem_block_t {
    unsigned size;
    mem_block_t* next;
};

/* static functions (not visible outside of this file) */

/*
   mem199_init

   Initializes the memory management package.  Called before any blocks
   are allocated.
*/
static void mem199_init ();

/*
   log2_ceil

   Calculates the logarithm base 2 of a number, rounded up to the
   nearest integer.  Useful in determining what size block to allocate
   for a given memory request, as allocations are always made in powers
   of two.
*/
static int log2_ceil (unsigned value);


/* file scoped variables */

static char*       free_bytes;                     /* unallocated memory   */
static int         n_free_bytes;                   /* unallocated bytes    */
static mem_block_t* mem_bin[MEM199_MAX_ALLOC_LOG+1]; /* free block lists     */
static int         init_done = 0;                  /* package initialized? */


/*
    mem199_allocate -- allocate a new block of n_bytes
    INPUTS -- minimum number of bytes in block available to caller
    OUTPUTS -- none
    RETURN VALUE -- pointer to new block (past header), or
                    NULL if no more memory available
*/

void*
mem199_allocate (unsigned n_bytes)
{
    unsigned block_size;    /* minimum size of allocated block           */
    int bin;                /* bin that holds blocks of appropriate size */
    mem_block_t* new_block; /* the new block                             */

    /* On first call, initialize static data for the memory manager. */
    if (!init_done)
        mem199_init ();

    /* Add room for a header to find the necessary size. */
    block_size = n_bytes + sizeof (*new_block);

    /* Unsigned, so no need to check for requests < 0. */
    if (n_bytes == 0 || block_size > MEM199_MAX_ALLOC)
        return NULL;

    /* Find the bin number. */
    bin = log2_ceil (block_size);

    /* Do we have a block sitting around? */
    if (mem_bin[bin] != NULL) {

        /*
           If so, remove the first one from the bin
           (a linked list of blocks).
        */
```

```
        new_block = mem_bin[bin];
        mem_bin[bin] = new_block->next;

    } else {
        /* No spare block, so try to allocate a new one. */

        /* Find the total block size. */
        n_bytes = (1UL << bin);

        /* Not enough space left in heap?  Return failure. */
        if (n_free_bytes < n_bytes)
            return NULL;

        /* Allocate the block from the front of the heap. */
        new_block = (mem_block_t*)free_bytes;
        free_bytes += n_bytes;
        n_free_bytes -= n_bytes;

        /* Mark the block's size in the header area. */
        new_block->size = n_bytes;
    }

    /* Return a pointer to the part AFTER the header. */
    return (new_block + 1);
}


/*
    mem199_allocate_and_zero -- allocate a new block of n_bytes and fill
                                it with zeroes
    INPUTS -- minimum number of bytes in block available to caller
    OUTPUTS -- none
    RETURN VALUE -- pointer to new block (past header), or
                    NULL if no more memory available
*/

void*
mem199_allocate_and_zero (unsigned n_bytes)
{
    void* new_block;

    /* First allocate a block.  If the attempt fails, so does this
       function. */
    new_block = mem199_allocate (n_bytes);
    if (new_block == NULL)
        return NULL;

    /* Set the bytes to zero.  Note that the pointer returned to us
       points past the memory header, so we only zero the data to be
       used by the caller, not the memory management information. */
    memset (new_block, 0, n_bytes);

    /* Return the new block. */
    return new_block;
}


/*
    mem199_reallocate -- change the size of a block of memory, allocating
                         a new block if necessary
    INPUTS -- ptr_to_ptr, a pointer to the pointer to the old block
              n_bytes, the minimum number of bytes in reallocated block
                       available to caller
    OUTPUTS -- *ptr_to_ptr, a pointer to the new block (on success only)
    RETURN VALUE -- 0 for success, in which case *ptr_to_ptr may have changed
                    -1 for failure, in which case *ptr_to_ptr does not change
    SIDE EFFECTS -- if a new block is necessary, and is created successfully,
                    data from the old block are copied into it, and the
                    old block is freed
*/

int
mem199_reallocate (void** ptr_to_ptr, unsigned n_bytes)
{
    mem_block_t* old_block;  /* pointer to old block of data */
    mem_block_t* new_block;  /* pointer to reallocated block */

    /*
       Calling with ptr_to_ptr equal to NULL should lead to an
       assertion (deliberate crash), but we'll just return failure.
    */
    if (ptr_to_ptr == NULL)
        return -1;

    /* If the pointer is valid, read the old block pointer. */
    old_block = *ptr_to_ptr;

    /*
       If the new size (including the header) still fits in the
       current block, nothing need be done to succeed.  Note the
       method used to access the header, which sits before the pointer
       returned by the earlier allocation call.
    */
    if (old_block != NULL &&
        n_bytes + sizeof (*old_block) <= old_block[-1].size)
        return 0;

    /*
       Try to create a new block.  Return failure if necessary
       (without changing the old block pointer).
    */
    new_block = mem199_allocate (n_bytes);
    if (new_block == NULL)
        return -1;

    /*
       New block exists, so write it over the old pointer; we still
       have a copy in old_block for the rest of this function.
    */
    *ptr_to_ptr = new_block;

    /*
       The data block must have grown, so copy all old bytes if an old
       block existed, then free the old block.  Note that the header
       bytes are not included, since old_block points past them, and
       the new block has its own header.
    */
    if (old_block != NULL) {
        memcpy (new_block, old_block,
                old_block[-1].size - sizeof (*old_block));
        mem199_free (old_block);
    }

    /* All done.  Return success. */
    return 0;
}


/*
    mem199_free -- free a block of memory
```

```
    INPUTS -- a pointer to the old block
    OUTPUTS -- none
    RETURN VALUE -- none
    SIDE EFFECTS -- the block now belongs to the memory management package,
                    which may reuse it later
*/

void
mem199_free (void* ptr)
{
    mem_block_t* mem_block = ptr;  /* memory block pointer     */
    int bin;                       /* bin number for old block */

    /* Check for free of NULL pointer.  Again, should probably have
       assertion rather than simple return. */
    if (ptr == NULL)
        return;

    /* Put the block into the appropriate bin. */
    bin = log2_ceil (mem_block[-1].size);
    mem_block[-1].next = mem_bin[bin];
    mem_bin[bin] = &mem_block[-1];
}


/*
    mem199_init -- initialize memory management data
    INPUTS -- none
    OUTPUTS -- none
    RETURN VALUE -- none
    SIDE EFFECTS -- initializes static data and sets up pointers to
                    unallocated region of memory (a simulated heap)
*/

static void
mem199_init ()
{
    /* All bins are empty (set pointers to NULL). */
    memset (mem_bin, 0, sizeof (mem_bin));

    /* Allocate a "heap" for us to manage. */
    n_free_bytes = 16 * MEM199_MAX_ALLOC;
    free_bytes = malloc (n_free_bytes);
    if (free_bytes == NULL) {
        perror ("initialize (malloc) mem199 package");
        exit (3);
    }

    /* Init has run; make a note of it. */
    init_done = 1;
}


/*
    log2_ceil -- calculate the logarithm base 2 of the value passed, rounded
                 up to the nearest integer
    INPUTS -- an unsigned value on which to operate
    OUTPUTS -- none
    RETURN VALUE -- ceil (log_2 (value)), as an integer, or
                    -1 if value == 0
*/

static int
log2_ceil (unsigned value)
{
    int ret_val;

    /*
       If value is a power of 2, we start counting at -1, otherwise,
       we start counting at 0 (to round up).
    */
    if ((value & (value - 1)) == 0)
        ret_val = -1;
    else
        ret_val = 0;

    /*
       Shift the value to the right until it disappears.  Counting with
       a loop in this manner is not the fastest possible method, but it
       is the simplest.
    */
    while (value > 0) {
        ret_val++;
        value >>= 1;
    }

    return ret_val;
}
```

14.1 Object-oriented programming

In C, it is only possible to hide code by putting it in a single file (file-scoped) and making the functions static. However, it is often nice to be able to create "code modules," where the only way internal data can be modified is through specific functions, or *methods*.

C++ supports this idea of "code modules" with *classes*. A *class* is a structure plus additional information that includes information hiding and scoping support, function prototypes, and static variables. C++ allows the class to be separated into visible (public) and private portions.

Visible (public)	Private
Existence of structures Interface functions	Data within structures Internal (implementation) functions Initialization (constructor[1]) Teardown code (destructor[1])

Functions that operate on a single *instance* (one copy) of something are common when programming. Let's say we again have a `player_t` structure below that has some information about one of the players in a game we are constructing.

```
struct player_t {
    char* name;         /* player login name */
    char* password;     /* cleartext password */
    int num_played;     /* number of games we have played */
    int win_guesses[13];/* number of guesses it took to win */
    double win_percent; /* percent of times we have won */
};
```

In C, we might declare a function that is called when the player wins the game. It will update `num_played`, `win_percent`, and the `win_guesses` array when called:

```
void player_win(player_t* p, int num_guesses);
```

In C, we would call this function the following way:

```
player_t player1; /* assume player already initialized */

/* some code that plays the game */

/* player1 wins the game after 5 guesses*/
player_win(&player1, 5);
```

[1] Automatically called by the compiler.

Let's recreate the player with a C++ class:

```
class player_t
{
    public:
        void player_win(int num_guesses);  /* player_t* argument
                                              is implicit */
    private:
        char* name;          /* player login name */
        char* password;      /* cleartext password */
        int num_played;      /* number of games we have played */
        int win_guesses[13]; /* number of guesses it took to win */
        double win_percent;  /* percent of times we have won */
};
```

In C++ the `player_t*` argument is implicit when we call the `player_win` function. We have written the following inside the class:

```
void player_win(int num_guesses);  /* player_t* arg is implicit */
```

The function above is called a `method`, which means that the first argument is implicit and its type is a pointer to this class. A method operates on the object that calls it.

Let's see how to use the C++ class:

```
player_t* p;   /* pointer to a player class object[2] */

/* some code to play the game */

p->player_win(5);  /* player wins after 5 guesses */
```

Now let's write the definition for `player_win`:

```
int player_t::player_win(int num_guesses)
{
    this->win_guesses[this->num_played] = num_guesses;
    this->win_percent = ((this->win_percent * this->num_played)+1)
                         /(this->num_played+1);
    this->num_played++;
}
```

The `player_t` followed by two colons above means that this method is a member of the `player_t` class. This notation is called *scoping*, and you must provide the correct scoping when writing C++ definitions. The "`this`" pointer above is an implicit pointer to the object that called the method, in this case a `player_t*`. For example, in the above code we have

[2]This isn't quite right because the memory has not been allocated for it and it hasn't been initialized by the constructor. For now assume that those things have been done.

`p->player_win(5);` in which case "`this`" would be the same as `p`. Since it is somewhat cumbersome to always write "`this`", C++ allows you to leave it out inside the definition, and the compiler will automatically fill it in. The definition for `player_win` then becomes:

```
int player_t::player_win(int num_guesses)
{
    win_guesses[num_played] = num_guesses;
    win_percent = ((win_percent * num_played)+1)/(num_played+1);
    num_played++;
}
```

14.2 Constructor and Destructor

C++ allows one to write code that is run whenever a new object is created or destroyed. This code is called the *constructor* and *destructor*, respectively. For example, we might want to initialize all of the player structure values in our `player_t` class, so we add a constructor and destructor to our class[3]:

```
class player_t
{
    public:
        player_t();   /* constructor */
        ~player_t();  /* destructor  */
        void player_win(int num_guesses);  /* player_t* argument
                                              is implicit */
    private:
        char* name;          /* player login name */
        char* password;      /* cleartext password */
        int num_played;      /* number of games we have played */
        int win_guesses[13]; /* number of guesses it took to win */
        double win_percent;  /* percent of times we have won */
};
```

[3]The C++ compiler will automatically create a constructor and destructor if you do not specify one.

```
/* Constructor
 * We initialize the values and also allocate dynamic
 * memory for the name and password using the new operator (see
 * description of new below)
 */
player_t::player_t()
{
    num_played = 0;
    for(int i=0; 13 > i; i++){
        win_guesses[i] = 0;
    }
    win_percent = 0;

    name = new char[10]; /* dynamic memory for 9 char string + NULL */
    password = new char[10];
}

/* Destructor
 * We need to deallocate the dynamic memory that was created in
 * the constructor so we don't have any memory leaks
 */
player_t::~player_t()
{
    delete[] name;   /* free dynamic memory allocated in constructor */
    delete[] password;
}
```

Now, whenever we create a new `player_t` object, the constructor will get called by the compiler, and the object will be initialized. The initialization will zero out all of the fields and allocate dynamic memory for the `name` and `password` strings. The destructor is then called by the compiler when the object needs to be destroyed. The destructor will then free the dynamic memory that was initially allocated by the constructor. For example, consider the following function:

```
void play_round()
{
    player_t player1;  /* calls player_t constructor implicitly */
    player_t player2;  /* calls player_t constructor implicitly */

    /* some code to play the game */
}
```

The `play_round` function declares two local variables: `player1` and `player2`. The compiler inserts code to call the `player_t` constructor when each of these local variables is

created at the start of the function. After the function has ended, the local variables are no longer needed; before the function returns to the caller, the compiler inserts code to call the `player_t` destructor on the `player1` and `player2` objects. Note that in this case the calls to the constructor and destructor were *implicit* and taken care of by the compiler.

14.3 The New and Delete Operators

Rather than using `malloc` and `free`, we can allocate dynamic memory using the `new` and `delete` operators. The operators also provide support for calling the object's *constructor* and *destructor*. The `new` operator calls `malloc` followed by the constructor; the `delete` operator calls the destructor followed by `free`.

14.4 Data Inheritance

C++ also provides data inheritance. This can be useful when creating objects that share information. We say that the data is inherited. Consider the illustration in Figure 14.1.

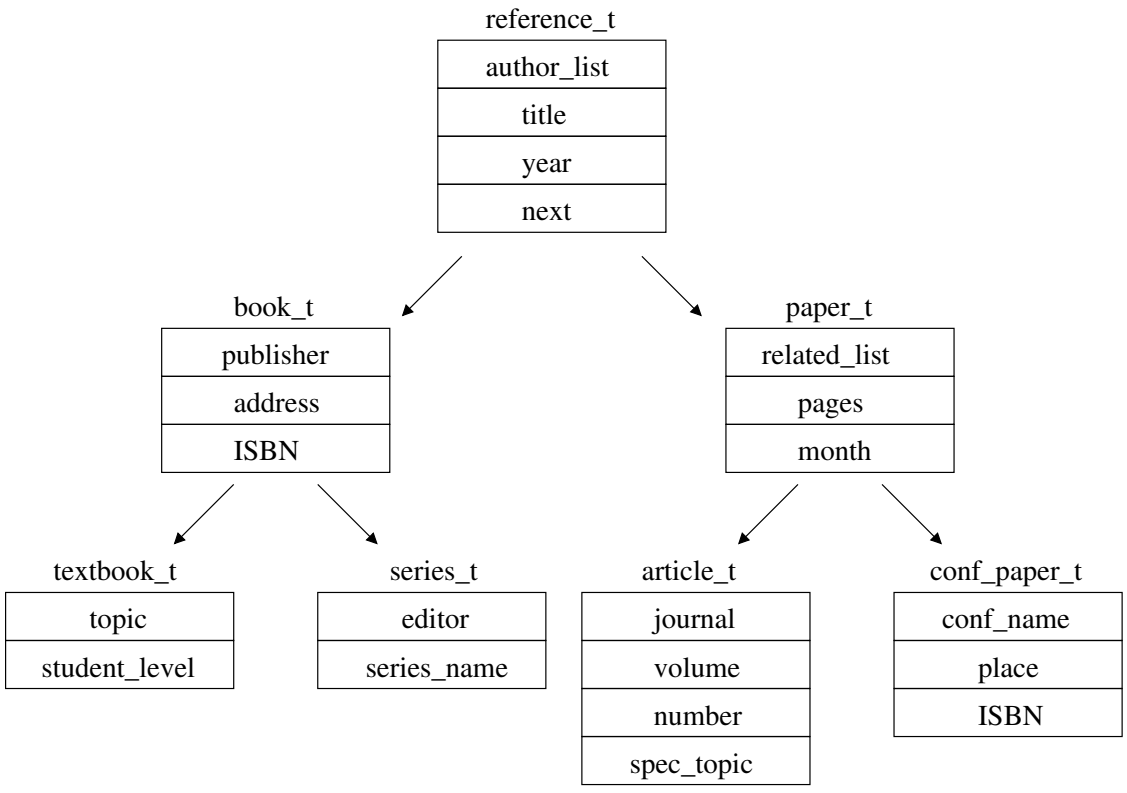


Figure 14.1: Data Inheritance

At the top of 14.1 we have a `reference_t`, which includes an `author_list`, `title`, `year`, and a `next` pointer to the next `reference_t`. However, we also have books and papers, for which we want to store more information. For a book, we want to have the `publisher`, `address`, and `ISBN`. Furthermore, an academic paper should have different information such as a list of related papers, the number of pages, and the month it was written. Likewise, we have textbooks and series, which contain book and reference information in addition to new textbook and series-specific information.

How can we make this abstraction work in C++? The answer is that classes in C++ can inherit other classes. As a result a class hierarchy is created with *superclasses* being "above" a class and *subclasses* being "below" the class. If we just created a `reference_t` class only the data in Figure 14.2 would be included. However, if we create a `series_t`, it will be laid out as shown in Figure 14.3.

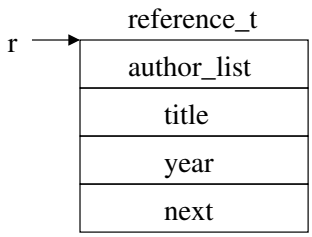


Figure 14.2: reference_t layout

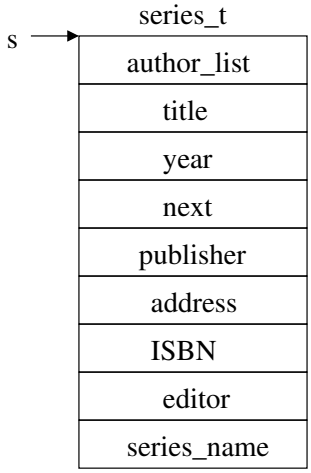


Figure 14.3: series_t layout

Suppose we create a function `void print_cite();` that is defined in the `reference_t` class. What happens if we do the following:

```
series_t* s;   /* assume memory has been allocated and initialized */
s->print_cite();
```

The above will work even though we are calling a `reference_t` method on a `series_t` object. The reason is that conversion to superclass is automatic in C++. Notice how `s` points to the same data as the pointer `r` to the `reference_t` object. When a `series_t` object calls `print_cite`, the `print_cite` function simply ignores all the data after `next`.

14.5 The `static` keyword

The `static` keyword means that the function is not a method and therefore is not associated with an instance of a class. For example:

```
class reference_t
{
    public:
        static reference_t* find_author(const char* find);
    private:
        author_list* author_list;
        char* title;
        int year;
        reference_t* next;
};
```

When calling `find_author`, we do not want to call it on a specific `reference_t` object, but rather want to search the list of `reference_t` objects for a certain author.

On the other hand, the following function should be a method because it needs to operate on a specific reference:

```
int has_author(const char* find);
r->has_author("Lumetta"); /* check object to see if Lumetta is the
                             author */
```

14.6 Virtual functions (Functional Inheritance)

What if we are in the middle of creating our subclasses, but don't know how many subclasses we are going to create or are in the middle of creating them? How can we write a function to print all the information about each of our objects? The solution is *virtual functions*.

```
virtual void print_all_cites();

reference_t* r;
r->print_all_cites();
```

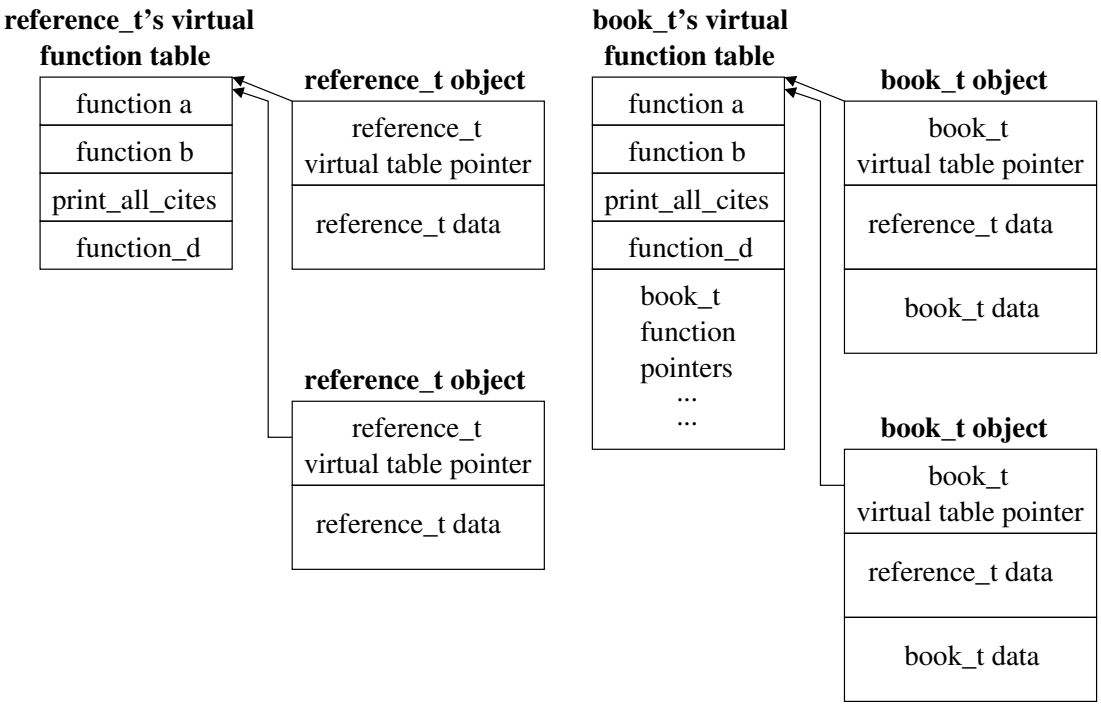


Figure 14.4: Virtual Functions

Note that each class in Figure 14.4 has a virtual function table that contains pointers to functions. One of the functions in the `reference_t` virtual function table is `print_all_cites`. The `book_t` virtual function table inherits the `reference_t` virtual functions. One thing we can do is replace the `print_all_cites` function pointer in `book_t`'s virtual function table with a pointer to another function. For example, we could replace `reference_t::print_all_cites` in the `book_t` virtual function table on the right with a function pointer to a new function `book_t::print_all_cites`. Now if we do

```
book_t* b;
b->print_all_cites();
```

the `book_t::print_all_cites` function will be called instead of the `reference_t::print_all_cites` function. How exactly does this work? As can be seen from Figure 14.4, each object has an implicit field inserted by the compiler that points to its virtual function table. For `reference_t` objects, the pointer refers to the `reference_t` virtual function table, while for `book_t` objects, the pointer refers to the `book_t` virtual function table. When the `book_t` object calls the `print_all_cites` function, the address of the function is looked up by following the pointer from the `book_t` object to the virtual function table. From the virtual function table, we can find the pointer to the function we want to invoke, which is `print_all_cites` in this case.

```
class reference_t
{
    public:
        void print_reference_cites()
        {
            printf("Title: %s\n", title);
            printf("Year: %d\n", year);
        }
        virtual void print_all_cites()   /* (1) */
        {
            print_reference_cites();
        }
    private:
        author_list* author_list;
        char* title;
        int year;
        reference_t* next;
};

class book_t : public reference_t
{
    public:
        virtual void print_all_cites();   /* (2) */
        {
             print_reference_cites();  /* inherited method from
                                          reference_t class */
             printf("Publisher: %s\n", publisher);
             printf("ISBN: %d\n", ISBN);
        }

    private:
        char* publisher;  /* publisher name      */
        char* address;    /* publisher's address */
        double ISBN;      /* ISBN number         */
};
```

The code above defines the `reference_t` class with the virtual function `print_all_cites`. The `book_t` class is a subclass (derived class) of the `reference_t` class, which is indicated by `": public reference_t"`.[4] These class definitions tell the compiler that the virtual function table for the `book_t` class should contain a function pointer to the `book_t::print_all_cites` function rather than the `reference_t::print_all_cites` function. Looking at Figure 14.4, this means that the `reference_t` virtual function table on the left will have a pointer to the function defined by `(1)` for `print_all_cites`, and the `book_t` virtual function table on the right will have a pointer to the function defined by `(2)` for `print_all_cites`.

What will each print if we call it?

```
reference_t* r;
r->print_all_cites();
```

This will call the function defined by `(1)` and print the title and the year.

```
book_t* b;
b->print_all_cites();
```

This will call the function defined by `(2)` and print the title, year, publisher, and ISBN.

You may wonder why the `print_reference_cites` method exists. The reason is that only public members of a class are inherited, which means that the `book_t` subclass does not have direct access to `author_list`, `title`, `year`, and `next`. However, since the `print_reference_cites` method is public, the `book_t` subclass does have access to this method.[5]

[4] The public keyword preceding `reference_t` indicates that all of the public members of the `reference_t` superclass will remain public in the `book_t` subclass. You can also use private to indicate that public members of the `reference_t` superclass should become private in the `book_t` subclass.

[5] C++ also provides the `protected` access specifier, which is the same as `private`, except that subclasses have access to `protected` members.

14.7 C++ References

In C++ a reference is a pointer with implicit dereference. For example:

```
int val;
int& val_ref = val;  /* declare a reference, which
                        implicitly assigns the address of val */
val_ref = 10;        /* same as val = 10 */
```

Why is this used? It is often used in *operator overloading*. Operator overloading lets us change the way a certain operator (+ - / * , *etc.*) behaves for a class. For example, if we create a new class to represent a complex number called `complex_t`, we might want to do the following:

```
complex_t x, y, z;
z = x + y;
```

If we overload the addition operator (+), we can specify exactly how the class should behave when adding two `complex_t` objects.[6] We would declare the overloaded function as follows:

```
complex_t operator+(const complex_t& a, const complex_t& b);
```

The reference operator forces the compiler to pass in a pointer, so that the value passed in can be modified by the overloaded function. However, in this example we do not need to modify the arguments (`a` and `b`) in order to implement the addition operator, so we use the `const` keyword[7], which prevents the function from modifying the arguments passed in. The advantage of passing arguments by `const reference` is that less stack overhead is needed for large objects.[8]

[6] A complex number has both a real and imaginary part.

[7] *Never* use non-`const` references outside of operator overloading because doing so will make the code incredibly hard to read and understand; others that use your code will not be expecting the function arguments they pass in to be able to be modified by the callee function.

[8] The extra time needed to indirectly access the value through a pointer may be greater than just copying the values to the stack if the objects are small. However, the actual size of an object may be difficult to determine, so the best solution is usually passing by `const reference`.

An Incomplete List of Advice for Sophomore System Builders

1. Take on a big project during your next few years.
2. Learn to use a debugger.
3. Don't put off learning about tools (such as make, CVS, perl, etc…).
4. Avoid optimizing prematurely.
5. Build. Burn. Rebuild.
6. The best designers are the best testers and debuggers.
7. Good code is like good prose.
8. Take on a big team project during your next few years.
9. Don't be afraid to break things.
10. Turn drudge work into opportunities for invention.