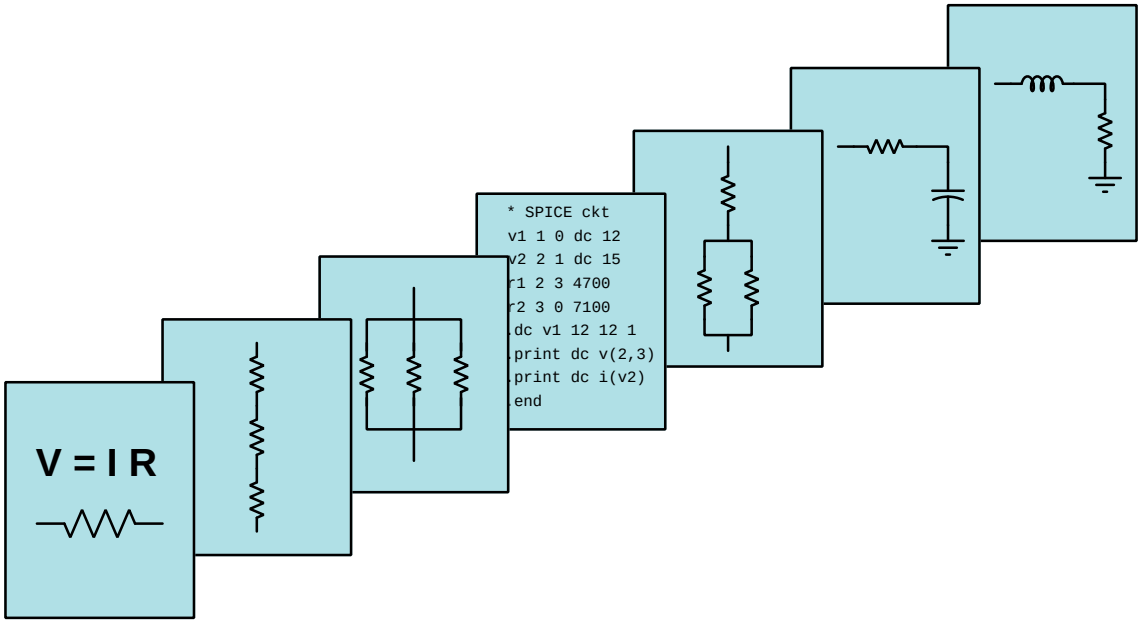
Modular Electronics Learning (ModEL) project



Introduction to C Language Programming

© 2021-2026 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 4 October 2026

This is a copyrighted work, but licensed under the Creative Commons Attribution 4.0 International Public License. A copy of this license is found in the last Appendix of this document. Alternatively, you may visit `http://creativecommons.org/licenses/by/4.0/` or send a letter to Creative Commons: 171 Second Street, Suite 300, San Francisco, California, 94105, USA. The terms and conditions of this license allow for free copying, distribution, and/or modification of all licensed works by the general public.

Contents

1 Introduction **3**
- 1.1 Recommendations for students . . . 3
- 1.2 Challenging concepts related to C programming . . . 6
- 1.3 Recommendations for instructors . . . 7

2 Case Tutorial **9**
- 2.1 Example: whitespace . . . 10
- 2.2 Example: variable scope . . . 12
- 2.3 Example: unique ways of using for loops . . . 15
- 2.4 Example: ncurses demo program . . . 18

3 Tutorial **21**
- 3.1 What is a programming language? . . . 21
- 3.2 Creating and running a simple C program . . . 22
  - 3.2.1 Step 1: write the source code . . . 23
  - 3.2.2 Step 2: compile the source code . . . 24
  - 3.2.3 Step 3: run the executable code . . . 25
- 3.3 C program fundamentals . . . 26
- 3.4 Writing neat code . . . 29
- 3.5 Comments . . . 31
- 3.6 Formatted output using `printf` . . . 35
- 3.7 Data types . . . 39
- 3.8 Basic mathematical functions . . . 44
- 3.9 Using the C math library . . . 50
- 3.10 Complex-number operations . . . 53
- 3.11 Formatted input using `scanf` . . . 55
- 3.12 Conditionals . . . 57
- 3.13 Loops . . . 66
- 3.14 Logical operators . . . 74
- 3.15 Bitwise operators . . . 83
  - 3.15.1 Bitwise-AND . . . 85
  - 3.15.2 Bitwise-OR . . . 86
  - 3.15.3 Bitwise-XOR . . . 87
  - 3.15.4 Bitwise complementation . . . 88

3.15.5 Bit-shifting . . . 89
3.15.6 Bitwise demonstration program . . . 91
3.15.7 Testing bit states . . . 93
3.16 Functions . . . 98
3.17 Pointers . . . 105
3.18 Arrays . . . 114
3.19 Structures . . . 123
3.20 Unions . . . 131
3.21 Operating system calls . . . 135
3.22 Debugging . . . 137
3.23 Simple graphics using C . . . 145
3.23.1 Ncurses . . . 146
3.23.2 Graphical back-end rendering . . . 148

4 Derivations and Technical References **155**
4.1 Introduction to assembly language programming . . . 156
4.1.1 Machine code to blink an LED . . . 157
4.1.2 Assembly code to blink an LED . . . 159
4.1.3 Slowing down the blinking . . . 161
4.1.4 Simplifying with symbols . . . 164
4.1.5 Using the stack . . . 165
4.2 ASCII character codes . . . 169
4.3 GCC quick reference . . . 170

5 Questions **171**
5.1 Conceptual reasoning . . . 175
5.1.1 Reading outline and reflections . . . 176
5.1.2 Foundational concepts . . . 177
5.1.3 Writing your first C program . . . 178
5.1.4 Re-writing a "Hello world!" program . . . 179
5.1.5 Writing a power calculation program . . . 180
5.1.6 Resonant frequency calculator program . . . 181
5.1.7 Decibel calculator program . . . 183
5.1.8 Logical-AND versus bitwise-AND . . . 185
5.1.9 Driving microcontroller output bits . . . 186
5.1.10 Bit-rotate program . . . 187
5.2 Quantitative reasoning . . . 188
5.2.1 Miscellaneous physical constants . . . 189
5.2.2 Introduction to spreadsheets . . . 190
5.2.3 Using C to analyze a series resistor circuit . . . 193
5.2.4 Using C to analyze a parallel resistor circuit . . . 195
5.2.5 Using C to analyze a series-parallel resistor circuit . . . 197
5.2.6 Using C to analyze a multi-source circuit . . . 199
5.2.7 Using C to calculate capacitive reactance . . . 201
5.2.8 Using C to analyze a series AC RLC circuit . . . 202
5.2.9 Using C to analyze a parallel AC RLC circuit . . . 204

5.2.10 Using C to analyze a series AC resistor-capacitor circuit . . . 206
5.2.11 Using complex numbers in C to analyze a series-parallel AC RLC circuit . . . 208
5.2.12 Using C to plot a sine wave . . . 210
5.2.13 Geometric sequence counting program . . . 212
5.2.14 RC time-constant calculator program . . . 213
5.2.15 Using C to analyze an RC charging-discharging circuit . . . 214
5.2.16 Writing an LR time-delay analysis program . . . 216
5.2.17 Writing a cutoff frequency calculator program . . . 217
5.2.18 Writing a low-pass filter simulation program . . . 218
5.2.19 Bitwise operation practice . . . 219
5.2.20 XOR cryptography . . . 220
5.2.21 Byte-shifting algorithm . . . 221
5.2.22 Sine calculator program . . . 222
5.2.23 Using C arrays to analyze a resistor circuit . . . 223
5.2.24 Sine look-up table . . . 225
5.2.25 Array-reversal program . . . 227
5.3 Diagnostic reasoning . . . 228
5.3.1 Find mistakes in a very simple program . . . 229
5.3.2 Find mistakes in a millimeter conversion program . . . 230
5.3.3 Case-sensitivity in variable names . . . 231
5.3.4 Find mistake in a function-calling program . . . 232
5.3.5 Automatic and static function variable testing . . . 233

A Problem-Solving Strategies **235**

B Instructional philosophy **237**
B.1 First principles of learning . . . 238
B.2 Proven strategies for instructors . . . 239
B.3 Proven strategies for students . . . 241
B.4 Design of these learning modules . . . 242

C Tools used **245**

D Creative Commons License **249**

E References **257**

F Version history **259**

Index **262**

Chapter 1

Introduction

1.1 Recommendations for students

The C programming language was invented in the early 1970's by Dennis Ritchie, a seminal figure in the world of digital computing. Despite its age, it remains a relatively powerful and widely-used programming language even at the time of this writing (2021). In fact, one online survey in 2020 listed C as the #5 programming language based partly[1] on the number of posted job descriptions for programming jobs. However, regardless of popularity, C is a good programming language to learn because it is relatively simple compared to others such as C++ and because it tends to produce lean code for memory-scarce applications such as embedded systems programming (e.g. microcontrollers).

Important concepts related to C programming include **machine language**, **source code**, **compiling** versus **interpreting**, **text editor**, **integer** versus **floating-point** number, **formatted text**, **modulus**, **casting**, **order of operations**, code **library**, **header** file, **linking**, **conditional** statement, **loop**, **function**, variable **scope**, **argument**, **pointer**, **string**, **bitwise** operators, and **recursion**.

Computer programming is a skill born of practice, and for this reason it is strongly advised that you start writing simple programs as soon as you can based on what you learn in this module. All code examples shown in this module have been tested on a computer and then copied-and-pasted to the page, which means they should all compile and run with no errors. This, in fact, is a great starting point for beginning programmers: to copy-and-paste working code into a text editor, compile and run it to test that it is error-free, and then begin modifying the source code to make the program do what you wish.

Here are some good questions to ask of yourself while studying this subject:

- What does it mean to *compile* a C program? Specifically, what is/are the input(s) to a compiler, and what is/are the output(s)?
- What it the definition of *source code*?

[1] Based solely on the number of open jobs requesting knowledge of languages, C was #6.

- What is a *text editor* application useful for in programming?
- What is whitespace, and where does it matter in a C program?
- What, at minimum, is necessary in a C program to make the code functional?
- What are comments useful for, and how do we place comments within C programs?
- What does the modulus function do?
- What is the "flow" of program execution for any of the example programs shown? In other words, for any given program with any given user input, can you predict exactly which instructions will be executed, and in which order?
- How do `char`, `int`, and `float` data types differ from one another?
- When might we wish to use *casting* in a C program?
- What does the `scanf` instruction, and how does it contrast against the `printf` instruction?
- What is the difference between the `=` operator and the `==` operator?
- When must an `else` statement accompany an `if` statement?
- How does the function of a `switch` statement differ from that of `if/else` statements?
- What does the `break` instruction do?
- What condition(s) must be satisfied for a `while` loop to continue its execution?
- What condition(s) must be satisfied for a `for` loop to continue its execution?
- What purpose is served by a *function* in the C language?
- What does it mean for a function to have *arguments* and to *return* data?
- What controls the scope of a variable in C?
- How does bit-shifting relate to the operations of multiplication and division?
- How do basic logical functions in C such as AND and OR differ from *bitwise* operations of AND and OR?
- What is the purpose of a *mask* in bitwise operations?
- For what purpose might we use the bitwise-AND operation when modifying the contents of a register in a microcontroller?
- For what purpose might we use the bitwise-OR operation when modifying the contents of a register in a microcontroller?
- For what purpose might we use the bitwise-XOR operation when modifying the contents of a register in a microcontroller?

- How may we use bitwise operations to test for the state of a single bit within a microcontroller register?
- How may bit-shifting be used for certain operations of multiplication or division on a binary value?
- Differentiate between the meanings of the following: `x`, `*x`, and `&x`
- What does it mean that functions in C operate on a *pass-by-value* basis rather than *pass-by-reference*?
- What is *defensive programming* and why is it important for real-world coding applications?

1.2 Challenging concepts related to C programming

The following list cites concepts related to this module's topic that are easily misunderstood, along with suggestions for properly understanding them:

- **Semicolons and where to (not) use them** – one of the most aggravating aspects of C for the new programmer is the use of semicolons to terminate lines of code. While annoying, this convention is one of the ways in which C lends freedom to use whitespace. Instructions that are more or less self-contained in one line require a terminating semicolon, while instructions that always precede other related lines of code (e.g. `if` conditionals, loop headers, function headers) do not.

- **Bitwise logical operations** – applying AND, OR, and XOR logical operations to respective bits of binary words is not a straight-forward concept, but is made more understandable by drawing out the binary word(s) in question and if necessary drawing small logic gates taking inputs from those bits to generate the respective bits of the output word. Another helpful perspective for understanding the purpose of bitwise operations is to view AND functions as *forcing* a 0 output if any input is 0, and OR functions as *forcing* a 1 output of any input is a 1.

- **Pointers** – a powerful yet confusing feature of C is that of *pointers* in which data may be referenced not only by variable name but also by memory location within the computer. Compiling, running, and modifying the example programs in the "Pointers" section of the Tutorial is a great way to learn this concept. The same is true where pointers are used in conjunction with arrays and structures.

The best general suggestion there is for learning how to program a computer in any language is to *write simple programs in that language* and keep a record of them for future reference. Programming is not a spectator sport, so get busy typing and compiling!!

1.3 Recommendations for instructors

This section lists realistic student learning outcomes supported by the content of the module as well as suggested means of assessing (measuring) student learning. The outcomes state what learners should be able to do, and the assessments are specific challenges to prove students have learned.

- **Outcome** – Demonstrate effective technical reading and writing

  Assessment – Students present their outlines of this module's instructional chapters (e.g. Case Tutorial, Tutorial, Historical References, etc.) ideally as an entry to a larger Journal document chronicling their learning. These outlines should exhibit good-faith effort at summarizing major concepts explained in the text.

  Assessment – Students show their own efforts compiling and running the code examples contained in the Tutorial chapter.

  Assessment – Students present their "Conceptual Toolbox" Theory Journal entries listing important concepts applied in the Tutorial text(s). A "Conceptual Toolbox" serves as inspiration when solving problems, each concept suggesting a potential problem-solving action. Challenge students to collaboratively generate a Conceptual Toolbox for the day's topic during the class session and then use it when solving problems together, thus fostering the habit of reasoning from principles rather than recalling procedures!

- **Outcome** – Explore programming concepts experimentally

  Assessment – For each new programming concept (e.g. at least one per Tutorial section), write a simple program in the C language that demonstrates the principle, and then modify that program to explore the effects of different parameter values, different strategies of implementation, etc. Computer programming provides an *excellent* opportunity for experimental learning, with the computer serving as a virtual laboratory in which to test ideas!

  Assessment – For programs accepting user input, challenge students to devise entries to intentionally "break" the program and make it fail. Then explore "defensive programming" techniques to guard against such attacks.

- **Outcome** – Combine mutually-reinforcing concepts such as programming and circuit analysis together

  Assessment – Write or modify a C program to analyze an electric circuit; e.g. pose problems in the form of the "Using C to analyze a series resistor circuit" Quantitative Reasoning question.

- **Outcome** – Independent research

  Assessment – Consult Brian Kernighan's and Dennis Ritchie's classic book *The C Programming Language* to learn how the earliest versions of C functioned, and to learn efficient programming techniques.

  Assessment – Consult *The GNU C Library Reference Manual* to learn how one of the most popular C compilers (`gcc`) interprets C code.

Chapter 2

Case Tutorial

The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as a scientific experimenter. Hopefully you will find these cases illuminating, and a good supplement to text-based tutorials.

These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: whitespace

A C compiler ignores *whitespace*, which means all non-character space between instructions. This apathy toward whitespace may be illustrated by examining the following "Hello World" programs, all of which compile with no errors and produce the exact same output:

C source code:

```
#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");

  return 0;
}
```

Program output:

```
Hello world!
```

Next, we see the exact same program re-written with no indenting or blank lines, compiles with no errors and produces the same output:

```
#include <stdio.h>
int main (void)
{
printf("Hello world!\n");
return 0;
}
```

Next we have eliminated all but one "carriage return" character, placing most of the code on a single line of text. It still compiles without error and produces the same output as before:

```
#include <stdio.h>
int main (void) { printf("Hello world!\n"); return 0; }
```

Finally, we have eliminated all "space" characters except those essential to defining certain special words like `int`. It still compiles without error and produces the same output as before:

```
#include<stdio.h>
int main(void){printf("Hello world!\n");return 0;}
```

2.2 Example: variable scope

```
#include <stdio.h>

void myfunction(float, float);

int main (void)
{
  // x and y are local to main()
  float x = 5.0, y = 8.0;

  printf("Before calling myfunction(): x=%f and y=%f \n", x, y);

  myfunction(x, y);

  printf("After calling myfunction(): x=%f and y=%f \n", x, y);

  return 0;
}

// x and y are local to myfunction()
void myfunction(float x, float y)
{
  x = x * 22.0;
  y = y * -41.0;

  printf("Within myfunction(): x=%f and y=%f \n", x, y);
}
```

When compiled and run, we see how the multiplications within `myfunction()` do not affect the values of `x` and `y` within `main()` even though they affect `x` and `y` within `myfunction()`:

Program output:

```
Before calling myfunction(): x=5.000000 and y=8.000000
Within myfunction(): x=110.000000 and y=-328.000000
After calling myfunction(): x=5.000000 and y=8.000000
```

Even if we make `x` and `y` *global* variables by declaring them outside of the `main()` function, the fact that `myfunction` initializes its own local versions of `x` and `y` means those are still local in scope to `myfunction` and therefore independent of the global `x` and `y`:

```
#include <stdio.h>

// x and y are global variables
float x = 5.0, y = 8.0;

void myfunction(float, float);

int main (void)
{
  printf("Before calling myfunction(): x=%f and y=%f \n", x, y);

  myfunction(x, y);

  printf("After calling myfunction(): x=%f and y=%f \n", x, y);

  return 0;
}

// x and y are local to myfunction()
void myfunction(float x, float y)
{
  x = x * 22.0;
  y = y * -41.0;

  printf("Within myfunction(): x=%f and y=%f \n", x, y);
}
```

Program output:

```
Before calling myfunction(): x=5.000000 and y=8.000000
Within myfunction(): x=110.000000 and y=-328.000000
After calling myfunction(): x=5.000000 and y=8.000000
```

However, if we keep `x` and `y` global in scope but do not pass variables on to `myfunction`, the multiplications taking place within `myfunction()` retain their effect back in `main()`:

```
#include <stdio.h>

// x and y are global variables
float x = 5.0, y = 8.0;

void myfunction(void);

int main (void)
{
  printf("Before calling myfunction(): x=%f and y=%f \n", x, y);

  myfunction();

  printf("After calling myfunction(): x=%f and y=%f \n", x, y);

  return 0;
}

void myfunction(void)
{
  x = x * 22.0;
  y = y * -41.0;

  printf("Within myfunction(): x=%f and y=%f \n", x, y);
}
```

Program output:

```
Before calling myfunction(): x=5.000000 and y=8.000000
Within myfunction(): x=110.000000 and y=-328.000000
After calling myfunction(): x=110.000000 and y=-328.000000
```

2.3 Example: unique ways of using for loops

A `for` loop in the C language takes the following form:

```
for(pre-loop action ; conditional check ; action at end of each pass)
  // instruction(s) to be repeated
```

A trivial example of a `for` loop is the following program and output when run:

```
#include <stdio.h>

int main(void)
{
  int n;

  for (n = 0 ; n < 5 ; ++n)
    printf("The value of n is %i \n", n);

  return 0;
}
```

Program output:

```
The value of n is 0
The value of n is 1
The value of n is 2
The value of n is 3
The value of n is 4
```

Though this may be a common way to use `for` loops – initializing a variable, checking that variable's value, and then incrementing it – other ways exist as well. Note that the first argument within a `for` loop's parentheses is simply the *first action taken* and not necessarily a variable initialization. The last argument is simply an action taken at the end of each pass through the loop, and need not be a changing of that variable's value.

To prove the point, let's completely remove the first and third arguments from the `for` loop, placing the first prior to the loop (declaring and initializing `n` in the same statement) and the second within the loop just prior to the closing right-brace character:

```
#include <stdio.h>

int main(void)
{
  int n = 0;

  for ( ; n < 5 ; )
  {
    printf("The value of n is %i \n", n);
    ++n;
  }

  return 0;
}
```

Program output:

```
The value of n is 0
The value of n is 1
The value of n is 2
The value of n is 3
The value of n is 4
```

We see how this is really just the same thing as a `while` loop:

```
#include <stdio.h>

int main(void)
{
  int n = 0;

  while (n < 5)
  {
    printf("The value of n is %i \n", n);
    ++n;
  }

  return 0;
}
```

Next we see a very unusual way of using a `for` loop:

```
#include <stdio.h>

int main(void)
{
  int n = 1;

  for (printf("Enter zero to quit\n") ; n ; scanf("%i", &n))
   ; // Do nothing

  return 0;
}
```

This program simply keeps looping and taking in user input until finally the entered number is zero at which point it exits. The first `for` argument is a `printf` statement, the middle argument is simply the value of `n` (where any non-zero value is equivalent to "true" and satisfies the `for` loop to do another pass, and the last argument is a `scanf` statement to take in user input. No additional statement(s) are necessary, which is why the line following the `for` loop is simply a semicolon (followed by a comment).

2.4 Example: ncurses demo program

```
#include <stdio.h>
#include <ncurses.h>

int
main (void)
{
  initscr();  // Starts curses mode

  printw("Hello World!");  // Sets up text ready to print

  refresh();  // Places items on the screen
  getch();    // Waits for any user keystroke

  move(10, 20); // Moves cursor to y=10, x=20 position
  printw ("Console lines = %i", LINES);

  move(11, 20); // Moves cursor to y=11, x=20 position
  printw("Console columns = %i", COLS);

  refresh();  // Places items on the screen
  getch();    // Waits for any user keystroke
  clear();  // Clears the screen

  move(0, 30);
  printw("Hello World! (on a blank slate)");

  refresh();
  getch();

  move(5, 30);
  float temp1 = 78.45;  // Declare and initialize two
  float temp2 = 135.01; // floating-point variables
  printw("1st temperature = %5.3f degrees  ", temp1);

  refresh();
  getch();

  move(5, 30);  // Move BACK to y=5,x=30 position
  printw("2nd temperature = %5.3f degrees  ", temp2);

  refresh();
```

```
  getch();

  if (has_colors() == FALSE)  // Test to see if terminal supports colors
    {
      endwin();
      printf("Sorry, but colors aren't supported in your terminal! \n");
      return 1;
    }

  start_color();  // Starts color capability

  // Color pair "0" is default: WHITE text on a BLACK background
  // Defines color pair "1" as BLUE text on a WHITE background
  init_pair(1, COLOR_BLUE, COLOR_WHITE);
  // Defines color pair "2" as RED text on a YELLOW background
  init_pair(2, COLOR_RED, COLOR_YELLOW);

  attron(COLOR_PAIR (1));  // Activate BLUE/WHITE scheme
  move(10, 30);
  printw("2nd temperature = %5.3f degrees  ", temp2);
  attroff(COLOR_PAIR (1)); // De-activate BLUE/WHITE

  attron(COLOR_PAIR (2));  // Activate RED/YELLOW scheme
  move(15, 30);
  printw("1st temperature = %5.3f degrees  ", temp1);
  attroff(COLOR_PAIR (2)); // De-activate RED/YELLOW

  refresh();
  getch();

  endwin();  // Ends ncurses and returns to normal console mode

  return 0;
}
```

This program illustrates some of the elementary features of `ncurses`, including how to display text (using `printw()` rather than `printf()`), how to position and re-position the cursor (using y and x coordinates), using `refresh()` to update the screen with previously-specified items, how to detect and pause for user keystrokes, how to determine the width (columns) and height (rows) of the console window, how to clear the screen, how to detect whether or not the terminal supports colored text, how to define color schemes as text/background combinations, and how to activate and de-activate those color schemes. This program also shows how to start an `ncurses` session (using `initscr()`) and how to end it (using `endwin()`).

Chapter 3

Tutorial

3.1 What is a programming language?

A *general-purpose computer* is a machine designed to operate on instructions you provide to it, these instructions usually relating to the storage and processing of data. Computer instructions are generally written using characters arranged according to specific rules, much like how alphabetical characters are used to visually encode human languages such as English. Thus, a *computer language* is the set of rules and conventions by which written characters may instruct a general-purpose computer to perform tasks.

Microprocessor hardware only "understands" one type of language, and that is called *machine language*. Machine language consists of binary codes representing data and actions to be taken on data. It is cryptic and highly specific to each model of microprocessor. Almost no human programmers write software in machine language, but rather write in some "higher-language" language that is easier to understand. The next higher-level programming language after machine code is *assembly*, consisting of alphanumeric acronyms and symbols representing binary machine codes. Software written in assembly language must be "assembled" into equivalent machine-language instructions before the computer is able to execute it. Like machine language, assembly is also cryptic and highly specific to microprocessor models, and few human programmers use it.

Most programmers opt for much higher-level languages such as C which almost resemble English words. Like assembly language, C and other programming languages cannot be *directly* understood by microprocessor hardware, and must be translated into machine code before running. With C, this translation process is called *compiling* and is done after the C code has been completely written. Another translation process used by some languages (e.g. Python) is called *interpretation*. Unlike compiler software which translates the programmer's entire code into a working machine language program in one step, interpreter software translates the programmer's code immediately as each line is entered into the computer. Whether compiled or interpreted, the final machine code that runs on the microprocessor hardware is called *executable code*, while the set of characters typed by the human programmer is called *source code*.

In this Tutorial you will learn the basic principles of writing source code in the C language, and also how to use compiler software to generate executable code from your source code.

3.2 Creating and running a simple C program

The basic work flow necessary to write a program in the C language and then run it on your computer is as follows:

1. Use a *text editor* application to type the "source code" for your program, and save it as a file
2. Use a *compiler* to translate this source code file to executable code
3. Execute the code you just compiled

On the following pages you will see these three steps performed within a Linux[1] console window.

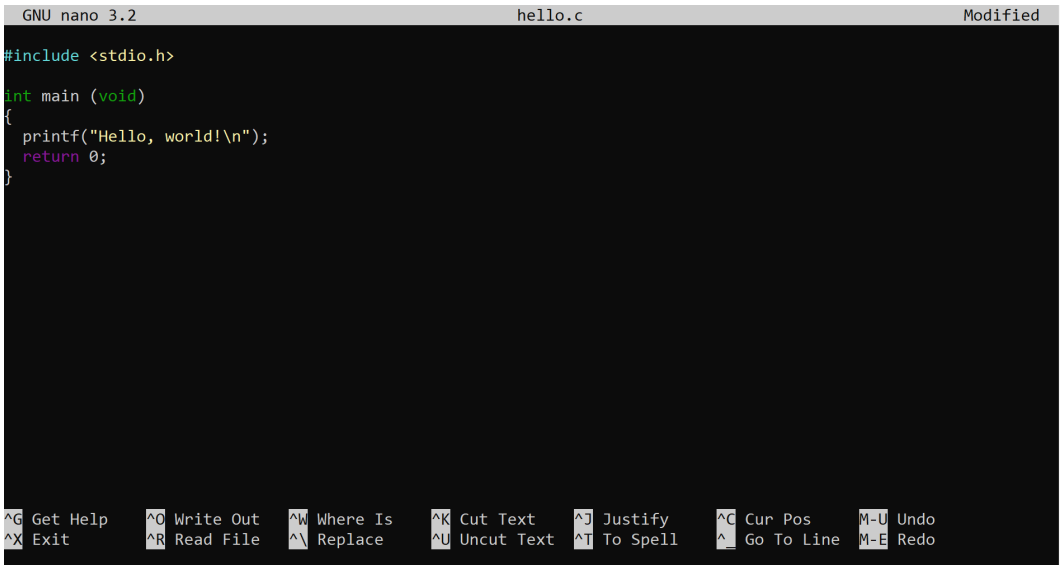
[1] These screenshots happen to be from a personal computer running the Microsoft Windows 10 operating system, but with "WSL" (Windows Subsystem for Linux) installed. This is a light-weight virtual machine (VM) allowing Linux to run concurrently within Windows.

3.2.1 Step 1: write the source code

First, we invoke the text editor `nano` (which is commonly found with most Linux installations) by typing `nano` at the command line followed by a space and then by the name of the plain-text file we wish to edit, and then lastly pressing the Enter key. In this case we will name the file `hello.c`, and so this means typing:

```
nano hello.c
```

Immediately the console view is replaced by the interface of the `nano` text editor where we may type the following code:



In order to save this new code to the file we must press Control-O ("Write Out"), or simply press Control-X ("Exit") to quit the text editor application. Like most editors, `nano` will ask the user if they wish to save any changes made to the file before exiting.

Note how the text we write shows up in different colors. This is because `nano`, like all modern text editors, recognizes the `.c` extension of the filename `hello.c` as a designation that this is a C-language source file, and knows how to color-code key words (e.g. `include`, `int`, `void`, and `return`) according to their purposes within the C language. As you might guess, this color-coding feature is extremely helpful for anyone learning a new programming language, as the color of a key word will usually change if it is not spelled correctly!

3.2.2 Step 2: compile the source code

After exiting `nano`, we are ready to "compile" our source code into a form that the computer will directly understand. We do this by typing the following command at the console's prompt and pressing Enter:

```
gcc hello.c
```

`GCC` is the name of the GNU C Compiler commonly used in Linux operating systems, and of course `hello.c` is the name of our source code file just created in `nano`.

After finishing its work, `GCC` will leave us with a file named `a.out` in the same location as our source file `hello.c`. This `a.out` file is executable, and when run will cause the computer to print "Hello world!" on the console.

3.2.3 Step 3: run the executable code

To run our code, we simply type the name of the executable file, preceded by `./` characters which instruct the Linux operating system to look for this file in the current location:

```
tony@DESKTOP-5MBJQNE:~$ ./a.out
Hello, world!
tony@DESKTOP-5MBJQNE:~$
```

As you can see, the words `Hello world!` appear on the next line following our command to run the program.

If we wish to make changes to our program, we simply re-start the text editor by typing `nano hello.c`, make our edits to the source file, save the changes and close the text editor, re-run the compiler by typing `gcc hello.c`, and then re-trying our executable file by typing `./a.out` again. This cycle of editing followed by compiling followed by a test run of the compiled program is practically universal within software development. For more complex programs we may use another software application called a *debugger* to trace the execution of the compiled code in fine detail, but for now we will focus on writing software in C simply using the edit/compile/run cycle.

3.3 C program fundamentals

A good way to begin learning about the C language is to dissect a simple program written in C. Here we will explain all the lines of code present in a C program intended to print the phrase "Hello world!" to the computer console. Below we see the C source code enclosed in a box, and below that we see the results of running this program on a personal computer. Note that this general format will be used throughout the Tutorial, showing C source code enclosed within a box and the running program's output without an enclosing box:

C source code:

```
#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");

  return 0;
}
```

Program output:

```
Hello world!
```

The first line of code in this simple program (`#include <stdio.h>`) instructs the preprocessor segment of the compiler software to include the contents of a file named `stdio.h` along with our source code. The great benefit of writing a computer program in a language such as C versus a "lower level" language such as machine code or assembly is that the instructions we use within our code tell the computer to do a lot of useful things with just a few characters. Although it may not seem like it, there are *many* steps required for a computer to print the words "Hello world!" to its screen, and a language like C hides a lot of this tedious detail from us so that we do not have to think about it. Some of that tedious detail is contained in files like `stdio.h`.

Any file with a `.h` filename extension is known as a *header* file, written in the same language as our source code (C), typically containing definitions necessary for our program to run. In the specific case of our `stdio.h` header file, this contains many definitions for the compiler to follow related to "standard input and output" which relates to text entered and displayed on a computer console window. If it were not for the `stdio.h` file being included, our "Hello world!" program would have to contain many more lines of code and would not be as easy to understand as it is now.

Programs written in C are organized into different sections called *functions*, which is a section of code that is able to receive data from and send data back to other sections of code. Every C program contains at least one function, called the *main* function. Here we see the "main" function's output data, name, and input data defined by the line `int main (void)`. Obviously, `main` is the name of the function, while less obvious is the fact that it outputs a single integer number value (`int`) and accepts no input data (`void`) when invoked. When a program that was written in C is first executed, the computer seeks the `main()` function and begins execution there. The *contents* of a function are all contained between the two curly-brace symbols ({ and }).

Within the `main()` function we find the one and only instruction doing any visible work, which is the `printf` instruction. Its odd-sounding name means that it prints *formatted* data to the console, which means we have much control over how the characters appear when printed to the computer screen. You will notice that this instruction has a semicolon (`;`) symbol at the very right-hand end, which marks the end of the instruction. This may seem unnecessary until you realize that C ignores most of the "whitespace" in the source code, which means it needs special characters to tell the compiler where the end of each instruction is and where the next instruction begins. To illustrate, consider this next version of the same "Hello world!" program where the `printf` and `return` instructions exist on the same line:

```
#include <stdio.h>

int main (void)
{
  printf("Hello world!\n"); return 0;
}
```

If C recognized whitespace as a delimiter between separate instructions, there would be no need for semicolon characters but we would not be able to place multiple instructions in the same line of code[2]. You will discover that these semicolons are extremely easy to overlook as you write your first C programs, and that at first your most common coding error will be forgetting semicolons at the end of every instruction. Note that the `main()` function required no semicolon because its last curly-brace (}) serves the same role.

The last instruction in this simple "Hello world!" program is the `return`, which outputs a value of zero before the `main()` function terminates. This actually serves no useful purpose in the program, but is included only because certain versions of C expect the `main()` function to always output some kind of data. If your compiler does not require the `main()` function to return any data, the program may be re-written with `void` as the return type, as follows:

```
#include <stdio.h>

void main (void)
{
  printf("Hello world!\n");
}
```

[2]Not all programming languages follow this philosophy. An excellent counter-example is the popular language *Python* where whitespace plays a prominent role, in which carriage returns (new lines) demarcate separate instructions and indentation fulfills the same role as curly-braces in C.

3.4 Writing neat code

The C programming language ignores most whitespace within the source code, granting much freedom as to how we might wish to format our code for appearance. Consider the following "Hello world!" example:

```
#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");

  return 0;
}
```

In this example we see each brace on its own line, a blank line separating the `printf` and `return` instructions, and each instruction contained within the `main()` function indented. These formatting features are completely irrelevant to the compiler, but serve to make the source code more easily read and understood by human eyes just as indentation and spacing enhances the readability of any printed text. For contrast, consider the following version of the same program, which compiles and runs exactly the same as the first, but has been formatted a bit differently so as to occupy fewer lines of text on the page:

```
#include <stdio.h>

int main (void) {
  printf("Hello world!\n");
  return 0;
}
```

Again, the indentation of the `printf` and `return` instructions serves no technical function other than to make it visually apparent that both instructions belong to the body of the `main()` function. We could have omitted this indentation such that both the `printf` and `return` instructions sit flush with the left-hand edge of the page just like `int main (void)` {, but stylistically this would be considered bad form.

Continually adhering to "proper" text formatting is an additional cognitive burden for anyone writing code, and so you will find software applications such as `indent` made to automatically "beautify" C source code after being written. For example, this is what you would type at the command line of your operating system to beautify your `hello.c` source code file:

```
indent hello.c
```

For example, I will show two versions of the same "Hello world!" program, the first hand-written with terrible formatting and the second re-arranged by `indent`:

```
#include <stdio.h>
int main (void) {printf("Hello world!\n"); return 0;}
```

```
#include <stdio.h>
int
main (void)
{
  printf ("Hello world!\n");
  return 0;
}
```

Both of these programs compile without any warnings or errors, and result in identical executable files following compilation. However, it should be clear to see how much more readable the second version is, and how beneficial this automatic formatting might be when writing complicated C programs!

As with most command-line utilities, `indent` has a wide range of options for specifying how it operates. These options allow you to customize the rules it uses when re-arranging C source code, adjusting the visual "style" of that code according to your preferences. It is left as an exercise to the reader to research and experiment with options available within `indent`.

3.5 Comments

All text-based programming languages offer a very useful feature called *comments*, which are nothing more than notes inserted into the source code for the benefit of human readers. Programs written without comments are considered non-professional, and even the author of a program will find their own comments very useful when they must edit their code years after writing it.

The C programming language supports two styles of comments: *single-line* and *multi-line*. This will be illustrated by example, first showing single-line comments which begin with double-slash characters (`//`):

Source code with single-line comments:

```
#include <stdio.h>   // This is a preprocessor directive

// Here is the main function!
int main (void)
{
  // This is where we print "Hello world!" to the console
  printf("Hello world!\n");

  // Returning zero
  return 0;
}
```

Note how single-line comments may occupy their own line in the page, or trail to the right of an actual line of C code. When compiled, all characters to the right of the double-slash (as well as the double-slash characters themselves) are ignored. Comments, no matter where placed, do not influence the compilation process at all and therefore have zero effect on the executable code produced by the compiler.

Some comments are too lengthy to fit on a single line of the page. Consider the following example:

```
// "Hello world!" example program
// Part of the Modular Electronics Learning Project
// August 2021

#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");

  return 0;
}
```

Rather than be forced to write multiple single-line comments, C provides us with a multi-line commenting feature using `/*` to begin a block of comment text and `*/` to end it:

Source code with multi-line comment:

```
/* "Hello world!" example program
   Part of the Modular Electronics Learning Project
   August 2021  */

#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");

  return 0;
}
```

Modern text editor applications recognize comments and shown them in a different color of text from the rest of the C source code, making it easy for you to discern what is a comment and what is not.

Comments are obviously useful for annotating complicated C programs and are strongly encouraged for all programmers, but they have another really useful function as well: *temporary cancellation of code*. It is often useful when developing software to cancel one or more lines of code temporarily so those lines have no effect, without deleting those lines entirely. Coding in any language requires mastery of obscure syntax, and with code being so easy to mis-spell or otherwise corrupt it is generally far better to comment out a line of code you don't want now but may wish to use in the future than to delete that line entirely and have to re-type it in the future when it's needed again.

For example, suppose we wished to test the effect of deleting the `printf` instruction in our simple "Hello world!" program, knowing that it is in fact necessary and will need to be re-inserted into the program to make it complete. Watch here how we may "comment out" that line of code and then test the results:

```
#include <stdio.h>

int main (void)
{
//  printf("Hello world!\n");

  return 0;
}
```

Recall that the compiler ignores all characters to the right of the double-slash symbols, which means now the entire `printf` instruction will be ignored as though it were no longer in the source code file at all. When run, this program does nothing, which of course is what we might expect by removing the `printf` instruction. However, all we need to do to make this program whole again is simply delete the two forward-slash characters. There is no need to re-type that entire instruction, because all we did to "remove" it was make it a comment.

Of course, this technique of "commenting out" code works with multi-line comments as well. Surrounding a block of code with `/*` and `*/` character sequences is a very easy way to cancel all the surrounded code to test the effect(s) of removing that code from the program.

Comments are useful for maintaining multiple versions of code within the same source file, allowing only one of those versions to be compiled at any time. Consider the following program where multiple `printf` instructions exist but only one is seen by the compiler:

```
#include <stdio.h>

int main (void)
{
//  printf("Hello world!\n");
  printf("Hi world!\n");
//  printf("Aloha world!\n");

  return 0;
}
```

All we would need to edit in this program to make it output "Hello world!" instead of "Hi world!" would be to remove the double-slash comment characters from the first `printf` line and insert some in front of the second `printf` line. Of course, the "Hi/Hello world!" program is a trivial example, but you could well imagine a much longer C program with many lines of code, using comments to temporarily "suspend" some lines of code in favor of others with the ability to easily revert back to previous versions of the program simply by moving a few comment slashes (`//`) around.

3.6 Formatted output using printf

As we saw in the "Hello world!" program, the `printf` instruction is capable of printing literal text to the computer's console, that text being enclosed within parentheses:

```
#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");

  return 0;
}
```

A detail not previously discussed is the `\n` character sequence at the end of `Hello world!`. Clearly we did not see either the backslash character nor an "n" character printed to the console when this program ran, and so these are not literal characters to be printed. Instead, the backslash is an *escape character* which instructs the compiler to interpret the next character as a sort of command rather than a letter to be literally printed. There are several such escape-character sequences recognized by `printf`, and they include:

- \b = backspace
- \n = new line
- \t = horizontal tab
- \v = vertical tab

Had it not been for the backslash-n character sequence at the end of "Hello world!" our program would have terminated leaving the cursor at the end of the line rather than on its own (new) line. You are encouraged to try this on your own, so that the code reads `printf("Hello world!");` and see for yourself how the output looks without the benefit of the new line sequence.

Consider the following alteration to our simple program, and the resulting output:

```
#include <stdio.h>

int main (void)
{
  printf("Hello ");
  printf("world!\n");

  return 0;
}
```

Program output:

```
Hello world!
```

While the source code looks different than before, the program outputs the exact same text to the console. The important lesson here is that the `printf` instruction precisely follows what is commanded within its parentheses, no more and no less. The fact that we have two `printf` instructions on two different lines in our source code file is of no matter – there is only one line of text in the output because there is only one "new line" (\n) instructed.

Another capability of `printf` is to print characters specified by their ASCII code values. Simply insert the escape sequence \x followed by the hexadecimal value. Consider the following program and its output:

```
#include <stdio.h>

int main (void)
{
  printf("The ASCII character 0x47 is the letter \x47 \n");

  return 0;
}
```

Program output:

```
The ASCII character 0x47 is the letter G
```

One of the most useful capabilities of the `printf` instruction is its ability to print numerical values that are not literal. Consider the following program, showing one `printf` literally printing a number while the next `printf` displays the result of a mathematical statement:

```
#include <stdio.h>

int main (void)
{
  printf("The sum of 8 and 5 is 13\n");
  printf("The sum of 8 and 5 is %i\n", 8+5);

  return 0;
}
```

Program output:

```
The sum of 8 and 5 is 13
The sum of 8 and 5 is 13
```

The printed results look identical, but the difference is that in the first line the sum was typed into the source code by hand while in the second line the computer actually *computed* the sum of 8 and 5. The `%i` format specifier instructs `printf` to place an integer number at that location in the output, while the source of that number is specified outside of the quotation marks, after a separating comma.

Another capability of `printf` is to specify the precise numerical format of a value, including different numeration systems. Consider the following example:

```
#include <stdio.h>

int main (void)
{
  printf("The sum of 8 and 5 is decimal %i\n", 8+5);
  printf("The sum of 8 and 5 is octal %o\n", 8+5);
  printf("The sum of 8 and 5 is hexadecimal %x\n", 8+5);

  return 0;
}
```

Program output:

```
The sum of 8 and 5 is decimal 13
The sum of 8 and 5 is octal 15
The sum of 8 and 5 is hexadecimal d
```

A listing of different format specifiers for the `printf()` instruction appear here:

- `%i` or `%d` = signed integer number in base-10 (decimal) format
- `%u` = unsigned integer number in base-10 (decimal) format
- `%o` = unsigned integer number in octal (base-8) format
- `%x` = unsigned integer number in hexadecimal (base-16) format
- `%f` = floating-point number in standard format
- `%e` = floating-point number in scientific notation format
- `%g` = floating-point number in terse format
- `%lf` = double-precision floating-point number in standard format
- `%le` = double-precision floating-point number in scientific notation format
- `%lg` = double-precision floating-point number in terse format
- `%s` = a string of ASCII-encoded characters

3.7 Data types

All computers process *data*, digital computers doing so in the form of binary *words* consisting of collections of "high" (1) and "low" (0) states easily represented as electrical voltage signals, magnetization polarities, optical pulses, and other media. Different types of data such as numbers and alphabetical characters require different quantities of binary "bits" to properly represent.

The C programming language is known as a *strongly-typed language*, which means programs written in C must be very explicit about declaring the number of bits reserved in memory for each binary "word" before data may be placed into those reserved memory addresses. Other, more modern, programming languages such as Python are not this way, with the compiler or interpreter making those bit-allocation decisions for the human programmer. More primitive programming languages such as assembly (and machine coding) do not recognize data types at all, leaving it completely up to the programmer to decide how bits are stored and manipulated. With C we must declare the data type of each variable we intend to use in a program, and after that declaration any and all manipulations of those variables proceed on the basis of their respective types. This "strong" typing requirement may seem clumsy for beginning programmers, but there are definite advantages of this approach, namely *total control* over how the computer's memory is used as well as potentially *more efficient* use of that memory. These advantages are especially pronounced in embedded systems programming where the computer in question (often a single integrated circuit "chip") has very limited memory resources, which is why even to this day (2024) C remains the dominant programming language for embedded systems.

C's data types are referenced by pre-defined keyboards, some of the more common keywords being listed below:

- `char` – *intended for a single ASCII character, typically 8 bits*
- `int` – *a signed integer number, the number of bits dependent on the computer*
- `float` – *a floating-point or "real" number, typically 32 bits*
- `double` – *a double-precision floating-point or "real" number, typically 64 bits*
- `long double` – *a floating-point or "real" number, with at least as many bits*[3] *as* `double`

Note how the integer (`int`) data type does not specify a fixed number of binary bits. This means that an `int` number in a C program may be comprised of 32 bits when running in a computer with a 32-bit microprocessor, but when compiled and run on a 64-bit microprocessor that same integer number in the C program will be 64 bits wide. Such non-specificity in C's default integer data type can lead to *portability* problems, where software written for one computer may behave differently if compiled and run on a different computer.

[3] Interestingly, the `long double` data type is not standardized across all compilers and computer systems. It's often 80 bits in width, compared to the 64-bit `double` type, but in some systems it may be as large as 128 bits!

Precise specification of binary word size is possible in C if we include the header file `stdint.h` in our code (in addition to including `stdio.h`) and make use of that header's many fixed-width data types. This is useful for applications where the size and signed/unsigned nature of the integer number cannot be left to circumstance, for example programs that must be compiled to run on multiple models of computer and exhibit the exact same behavior on those differing computer platforms. Some of these fixed-width data types are listed here:

- `int8_t` – *Signed 8-bit integer*
- `int16_t` – *Signed 16-bit integer*
- `int32_t` – *Signed 32-bit integer*
- `int64_t` – *Signed 64-bit integer*
- `uint8_t` – *Unsigned 8-bit integer*
- `uint16_t` – *Unsigned 16-bit integer*
- `uint32_t` – *Unsigned 32-bit integer*
- `uint64_t` – *Unsigned 64-bit integer*

For example, a signed 16-bit integer (`int16_t`) has a range of −32768 to +32767 (using two's-complement notation) while an unsigned 16-bit integer (`uint16_t`) has a range of 0 to 65535. It should be clear from all these examples that choosing the correct data type in our C programs is a very important consideration, as each type limits the range of numerical values we might represent with it.

The following program demonstrates much about C data types and formatted printing of those values. Let us examine the source code, the resulting output, and then analyze what it all means:

```
#include <stdio.h>

int main (void)
{
  int n;
  float x;

  n = 37;
  x = 241.53;

  printf("n = %i decimal = %x hexadecimal \n", n, n);
  printf("x = %f = %e = %g \n", x, x, x);

  return 0;
}
```

Program output:

```
n = 37 decimal = 25 hexadecimal
x = 241.529999 = 2.415300e+02 = 241.53
```

The `int n;` and `float x;` instructions *declare* two variables in the computer's memory, an integer `n` and a floating-point `x`. The next two lines of code *initialize* them with numerical values. After that, the `printf` instructions display these values in different formats.

We've already seen how integer values may be formatted in either decimal or hexadecimal form by `printf`, but now we get to see some formatting options for floating-point values too. `%f` shows the value in decimal notation while `%e` shows it in power-of-ten exponential (i.e. scientific) notation. The `%g` option instructs `printf` to print the value in the most terse way possible, whether as a plain decimal number or in scientific notation. Note also how this simple program shows us how multiple values (or at least formats) may be printed by a single `printf` instruction: by placing the identifier codes where we want them in the text and also by including as many instances of the variable (separated by commas) as necessary after the final quotation mark.

Looking at this program's output, it should be clear that there is an error of sorts in the printed result. Our floating-point number value of 241.53 actually appears as 241.529999. There is no fault in our programming, per se, but rather what we are seeing here is an intrinsic limitation of floating-point binary representation: certain decimal values cannot be exactly represented as floating-point binary without "repeating bits" which means we get the equivalent of a rounding error in some of our numerical representations.

This next sample program shows three new concepts. The first is how we may use whole words and not just single letters as variable names, which is very useful to know because it means we may write our C source code in ways that make more sense to any human reading it. Second, we see a way to declare and initialize variables in single lines of code rather than using multiple lines. Third, we see how we may specify the number of digits printed to the console using `printf`:

```
#include <stdio.h>

int main (void)
{
  int count = 37;
  float temp = 241.53;

  printf("Count = %5i units \n", count);
  printf("Count = %07i units \n", count);
  printf("Temperature = %.8f degrees C \n", temp);

  return 0;
}
```

Program output:

```
Count =    37 units
Count = 0000037 units
Temperature = 241.52999878 degrees C
```

In the first `printf` instruction we see how to print an integer value using exactly five characters, padding this two-digit value with three blank spaces. In the next `printf` instruction we see how to pad an integer number with leading zeroes while specifying a total number of seven digits. Lastly, the final `printf` instruction specifies that our floating-point value should have exactly eight digits to the right of the decimal point rather than the default six.

An interesting feature offered by the C programming language is a way to declare variables as *read-only*, which means their values once initialized to some value cannot be changed. To do this, we simply precede the variable's data-type identifier with the modifier `const` to declare that value to be a *constant*. Consider the following inch-to-centimeter conversion program where the conversion factor between inches and centimeters (2.54) is a fixed constant and should never be altered:

```
#include <stdio.h>

int main (void)
{
  float length = 100.1;       // Defined in unit of inches
  const float convert = 2.54;

  printf("Length is equal to %f inches \n", length);
  printf("Length is equal to %f centimeters \n", length * convert);

  return 0;
}
```

Program output:

```
Length is equal to 100.1 inches
Length is equal to 254.254 centimeters
```

If any instruction is added to this program attempting to modify the value of `convert`, the compiler will refuse to compile that source code and will instead flag an error for the offending instruction.

3.8 Basic mathematical functions

C offers all the basic arithmetic functions you would expect a computer to be able to perform, including addition, subtraction, multiplication, and division. The following program and its output illustrate these basic arithmetic "operators" at work:

```
#include <stdio.h>

int main (void)
{
  int a, b;

  a = 7;
  b = 3;

  printf("Addition = %i\n", a + b);
  printf("Subtraction = %i\n", a - b);
  printf("Multiplication = %i\n", a * b);
  printf("Division = %i\n", a / b);

  return 0;
}
```

Program output:

```
Addition = 10
Subtraction = 4
Multiplication = 21
Division = 2
```

Note how the "Division" line shows a result of two for the operation $7 \div 3$. This is because the output is expressed as an *integer* value and is therefore rounded to the next lowest whole number.

One way to have the computer represent a non-integer quotient using nothing but integer values is to apply the *modulus* operator (%) which returns the remainder following division. Observe how this modified version of the program performs:

```
#include <stdio.h>

int main (void)
{
  int a, b;

  a = 7;
  b = 3;

  printf("Addition = %i\n", a + b);
  printf("Subtraction = %i\n", a - b);
  printf("Multiplication = %i\n", a * b);
  printf("Division = %i remainder %i\n", a / b, a % b);

  return 0;
}
```

Program output:

```
Addition = 10
Subtraction = 4
Multiplication = 21
Division = 2 remainder 1
```

The "2 remainder 1" output tells us that three fits into seven *twice* with *one* left over.

With some clever formatting in the last `printf` instruction we may even get the computer to express this quotient as a mixed fraction:

```
#include <stdio.h>

int main (void)
{
  int a, b;

  a = 7;
  b = 3;

  printf("Addition = %i\n", a + b);
  printf("Subtraction = %i\n", a - b);
  printf("Multiplication = %i\n", a * b);
  printf("Division = %i and %i/%i\n", a / b, a % b, b);

  return 0;
}
```

Program output:

```
Addition = 10
Subtraction = 4
Multiplication = 21
Division = 2 and 1/3
```

That last `printf` instruction displays the integer quotient (rounded down to 2), then the modulus of 7 and 3 (i.e. the remainder value of 1), and lastly the divisor (3).

Yet another option for displaying an accurate quotient for 7 and 3 is to make use of *floating-point* representation. A feature of the C programming language useful for forcing one type of numerical value into a different variable type is called *casting*, and it consists of placing the desired type in parentheses immediately to the left of the variable. In order to successfully divide the integer values of 7 by 3, we must cast each of these variables as floating-point (instead of integer) and then perform the division. Note how this is done in the next version of the program:

```
#include <stdio.h>

int main (void)
{
  int a, b;

  a = 7;
  b = 3;

  printf("Addition = %i\n", a + b);
  printf("Subtraction = %i\n", a - b);
  printf("Multiplication = %i\n", a * b);
  printf("Division = %f\n", (float)a / (float)b);

  return 0;
}
```

Program output:

```
Addition = 10
Subtraction = 4
Multiplication = 21
Division = 2.333333
```

The order of operations followed by the computer when executing an arithmetic expression coded in C is left-to-right, with multiplication and division taking precedence over addition and subtraction. As with standard mathematical notation, parentheses take precedence over all other arithmetic operations which means we may insert parentheses around portions of a mathematical expression to force the computer to evaluate those portions first.

Consider the following program, illustrating the order of operations by example:

```
#include <stdio.h>

int main (void)
{
  printf("%i\n", 3 + 5 * 6 - 2);
  printf("%i\n", (3 + 5) * 6 - 2);
  printf("%i\n", 3 + 5 * (6 - 2));
  printf("%i\n", (3 + 5) * (6 - 2));

  return 0;
}
```

Program output:

```
31
46
23
32
```

Each `printf` instruction evaluates and then displays three plus five times six minus two, but in four different orders as described below.

The first instruction follows the standard order of operations where multiplication comes before addition or subtraction, such that 5×6 is evaluated first, then 3 added and 2 subtracted to yield a result of 31.

The second instruction uses parentheses to force $3 + 5$ as the first step, followed by multiplication (by 6) and lastly subtraction of 2. Thus, the computer first calculates a sum of 8 and then multiplies by 6 to get 48, then subtracts 2 to yield a result of 46.

The third instruction shifts the parentheses to perform subtraction before anything else. Here it first calculates a difference of 4 before multiplying by 5 to get 20, then lastly adding 3 to get 23.

The fourth instruction uses two sets of parentheses to force the addition and subtraction to both occur prior to multiplication. In this case it computes a sum of 8 and a difference of 4, then multiplies those together to yield a result of 32.

C also supports *nested* parentheses where one or more sets of parentheses are enclosed within other sets of parentheses. Consider this example:

```
#include <stdio.h>

int main (void)
{
  printf("%i\n", ((4 + 8) / (8 - 2)) * 3);

  return 0;
}
```

Program output:

```
6
```

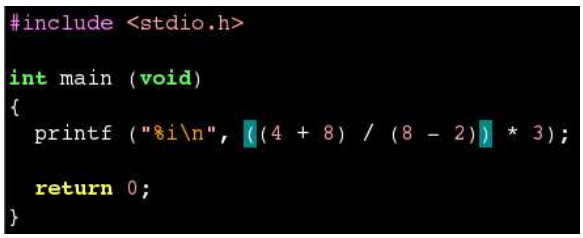
The computer will always evaluate all arithmetic functions from the inner-most parentheses outward. In this case it calculates the sum of $4 + 8$ as well as the difference of $8 - 2$ and then divides those to get a quotient of 2. Only after that does it multiply by 3 to achieve a final result of 6.

In conventional mathematics notation nested parentheses are often formatted on the page such that the outer parentheses are taller than the inner parentheses, as seen here:

$$\left(\frac{(4+8)}{(8-2)}\right) \times 3$$

However, such formatting does not exist in a plain-text editing environment, and so the programmer must keep careful track of all the parentheses to ensure they always exist in *pairs* and in the correct locations. One helpful tip when managing mathematical operations with nested parentheses is to count all the left-hand parentheses and make sure that number is equal to the number of right-hand parentheses.

Some text editor applications will helpfully highlight the matching parenthesis whenever the cursor rests over another one, to make it easier to see what is encompassed by that pair of parentheses. Below we see a screenshot of my editor highlighting a pair of parentheses in blue as a result of hovering the editor's cursor over one of them:



3.9 Using the C math library

In addition to basic arithmetic functions, the C programming language also offers a range of advanced mathematical functions in the *C math library*. A programming "library" is a collection of files accessible to the compiler which may be *linked* to the code you write. Libraries are extremely useful[4] as they allow you to write compact code using special functions pre-defined and pre-coded within the library(ies).

Consider the following example using the `pow()` function available in the C math library, which computes the power of its two arguments (e.g. `pow(3,4)` means 3^4) as a floating-point result:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  printf("%f\n", pow(3,4));

  return 0;
}
```

Program output:

```
81.000000
```

Two very important details must be present in order to access any mathematical functions contained in the C math library: first, the `math.h` header file must be included within your source code, as this file contains all the declarations necessary to use the library's math functions; second, you must instruct the compiler to include the math library itself (containing the actual instructional code for these functions). For example, if your C source code is saved to a file named `myprogram.c`, you would need to type this at your computer's command line:

```
gcc myprogram.c -lm
```

The `-lm` option specified for the `GCC` compiler is shorthand for "math library" or more precisely, "library, math".

[4] The fact that the C programming language has been in active use for many decades means a huge repository of functional libraries exist for the language, many of which are free and open-source which means you may incorporate them into your own software projects and thereby leverage the labors of many other programmers. The C math library is one of the first to be created for the language, and is included by default with every C compiler. Another important fact about programming libraries is that they *may* be machine-specific and/or compiler-specific rather than generic, so be sure to carefully read the documentation for any C library you intend to use.

Some useful C math library functions appear in the following example program:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  printf("The common logarithm of 1000 is %f\n", log10(1000));
  printf("The natural logarithm of 55 is %f\n", log(55));
  printf("e raised to the 2nd power is %f\n", exp(2));
  printf("The square root of 75 is %f\n", sqrt(75));
  printf("Five divided by infinity is %f\n", 5 / INFINITY);
  printf("45 degrees is %f radians\n", 45 * M_PI / 180);
  printf("The sine of 30 degrees is %f\n", sin(30 * M_PI / 180));
  printf("The arctangent of 0.5 is %f radians or %f degrees\n",
           atan(0.5), atan(0.5) * 180 / M_PI);

  return 0;
}
```

Program output:

```
The common logarithm of 1000 is 3.000000
The natural logarithm of 55 is 4.007333
e raised to the 2nd power is 7.389056
The square root of 75 is 8.660254
Five divided by infinity is 0.000000
45 degrees is 0.785398 radians
The sine of 30 degrees is 0.500000
The arctangent of 0.5 is 0.463648 radians or 26.565051 degrees
```

Note how every one of the math library functions consists of a word (e.g. `log10`, `exp`) followed by a set of parentheses in which one or more input value (called *arguments*) are placed. For functions such as `pow()` having multiple arguments, a comma separates the arguments from each other. Trigonometric functions such as sine (`sin()`) and arc-tangent (`atan()`) assume angular units of radians rather than degrees, and so we see the conversion factor $\frac{\pi}{180}$ used to convert degrees into radians and $\frac{180}{\pi}$ used to convert radians into degrees. Although there are no functions in the C math library built specifically to convert between radians and degrees, the `math.h` header file does contain common mathematical constants such as π (`M_PI`) which we see being used in this program.

Note also the `INFINITY` constant used within this example program, a constant provided by the `math.h` header file. Floating-point data types support "infinity" as a valid value, and therefore the `math.h` header file's pre-defined `INFINITY` constant is usable in the context of floating-point numbers. A practical example where `INFINITY` may be useful in a C program is if you are simulating an electric circuit and need to specify the electrical resistance of an *open* component.

A minor detail worthy of note in this program is how that last `printf` instruction is spread over two lines of code on the page. This was done for no other reason than human-readability and formatting on a printed page, as the compiler can understand a long single line of code quite well. Recall that C compilers ignore most whitespace, and so the line break between `degrees\n",` and `atan(0.5)` is of no consequence to the program's function, but breaking up the `printf` instruction at one of its comma characters makes a cleaner and easier-to-read presentation on the screen than letting the text editor word-wrap the line at some random location.

3.10 Complex-number operations

A deficiency in the original C language standard was a lack of support for complex-number data types and functions. This was remedied with the "C99" version of the C programming language standard when complex data types and functions were included in the standard math library. A new header file, `complex.h`, was also introduced with C99 to define certain keywords such as the imaginary operator (i or j, symbolized as `I` in C code).

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main (void)
{
  float complex z = 3 + (4 * I);

  printf("Rectangular = %f + %fi \n", creal(z), cimag(z));
  printf("Polar = %f @ %f radians \n", cabs(z), carg(z));
  printf("Polar = %f @ %f degrees \n", cabs(z), carg(z) * 180 / M_PI);

  return 0;
}
```

Note how the variable `z` is declared as a `float complex` data type rather than a typical `float`. Also note the four special complex-type functions used in this program: `creal()` returns the real value of the complex argument, `cimag()` returns the imaginary value, `cabs()` returns the absolute value (i.e. polar magnitude), and `carg()` returns the polar angle in *radians*[5].

As with all programs utilizing the math library, the math library must be "linked" at compilation, as shown in the following command-line invocation of the GCC compiler:

```
gcc -Wall myprogram.c -lm
```

The `-Wall` option turns on all warning messages, which is a good programming practice but not essential for compiling this program. The `-lm` option instructs `GCC` to "link" the math library to this code, which is essential for the complex functions as well as the built-in definition for π (`M_PI`). `GCC` by default outputs an executable file named `a.out` which is what you must invoke at the command line to run this program:

```
./a.out
```

[5] A full rotation is equivalent to 360 degrees or 2π radians. Thus, the conversion factor from radians to degrees is $\frac{360}{2\pi}$ or simply $\frac{180}{\pi}$.

This next example shows elementary arithmetic operations performed on two complex numbers a and b, assigned with values $3+4j$ and $-5+2j$ respectively. Additionally, it also shows how to use a `#define`[6] statement to make `j` the complex operator instead of `I`, the former being more customary in electrical engineering in order to avoid confusion with the variable i or I representing electrical *current*:

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

#define j I

int main (void)
{
  float complex a, b;

  a = 3 + (4 * j);
  b = -5 + (2 * j);

  printf("a = %f + %fj \n", creal(a), cimag(a));
  printf("b = %f + %fj \n\n", creal(b), cimag(b));
  printf("a + b = %f + %fj \n", creal(a + b), cimag(a + b));
  printf("a - b = %f + %fj \n", creal(a - b), cimag(a - b));
  printf("a * b = %f + %fj \n", creal(a * b), cimag(a * b));
  printf("a / b = %f + %fj \n", creal(a / b), cimag(a / b));
  printf("1 / a = %f + %fj \n", creal(1 / a), cimag(1 / a));

  return 0;
}
```

Program output:

```
a = 3.000000 + 4.000000j
b = -5.000000 + 2.000000j

a + b = -2.000000 + 6.000000j
a - b = 8.000000 + 2.000000j
a * b = -23.000000 + -14.000000j
a / b = -0.241379 + -0.896552j
1 / a = 0.120000 + -0.160000j
```

[6]The `#define` statement is a form of *preprocessor directive* instructing the compiler software to perform a global search-and-replace substitution prior to actually compiling the code. That is, the compiler first replaces every instance of `j` with `I` so that the visible source code may still use `j` as the imaginary operator but the compiler sees `I` instead.

3.11 Formatted input using scanf

We have already seen the `printf` instruction used many times to print formatted text to the computer's console display, and now we will explore the use of a complementary instruction called `scanf` to receive typed input from the human user at the same computer console.

Consider the following program which prompts the user to input a number and then displays the square root of that number:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float x;

  printf("Enter a positive number: ");
  scanf("%f", &x);

  printf("The square root of %f is %f \n", x, sqrt(x));

  return 0;
}
```

Now, we will run this program and manually enter the number 45 when prompted by it:

Program input and output:

```
Enter a positive number: 45
The square root of 45.000000 is 6.708204
```

The syntax of the `scanf` instruction is remarkably similar to that of `printf`: a code specifying the type of data to be input by the user (in this case, a floating-point number, `%f`) appears between two sets of quotation marks, followed by a comma and the location where that value will be stored. In this example, a floating-point variable named `x` has been declared at the beginning of the `main()` function, and within the `scanf` instruction we specify `x` as the variable which will receive the user's typed numerical input.

A very important difference between variables specified in `printf` versus variables specified in `scanf` is that with `scanf` we need to specify the *memory location* of that variable rather than just the name of the variable. In this case, `x` is the name of the declared floating-point variable, and `&x` is the location in the computer's memory[7] where that variable resides. Forgetting to include the `&` symbol in `scanf` is one of the most common errors new C programmers commit when first learning to use this instruction.

[7] The & character preceding a variable name is the "address of" operator in the C programming language. More on this topic is found in the "Pointers" section of the Tutorial.

Like `printf`, it is possible to reference multiple variables in `scanf`. Examine the following program to see how this works:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float x,y;

  printf ("Enter two numbers, pressing [Enter] after each: ");
  scanf ("%f %f", &x, &y);

  printf ("%f raised to the %f power is %f\n", x, y, pow(x,y));

  return 0;
}
```

Program input and output:

```
Enter two numbers, pressing [Enter] after each: 5
3
5.000000 raised to the 3.000000 power is 125.000000
```

Prior to learning about `scanf`, any numerical values required by our simple C programs had to be "hard-coded" into the program itself. This meant editing the C source code and re-compiling the program if we wished it to perform a different calculation. Now, using `scanf`, we may have the user enter numerical values at run-time which makes possible more practical and useful programs written in C.

3.12 Conditionals

A useful capability of all computers is to respond to changing *conditions* rather than performing the exact same action(s) unconditionally. In programming parlance, any code testing for a specific condition and redirecting the flow of instruction execution accordingly is known as a *conditional*. Two types of conditionals are supported in C programming: `if/else` and `switch`.

An example of an `if/else` conditional is shown in the following code:

```
#include <stdio.h>

int main (void)
{
  int n;

  printf("Enter an integer number: ");
  scanf("%i", &n);

  printf("The number %i is ", n);

  if (n % 2 == 0)
    printf("even\n");

  else
    printf("odd\n");

  return 0;
}
```

Program input and output: (with 4 as the entered value)

```
Enter an integer number: 4
The number 4 is even
```

Program input and output: (with 5 as the entered value)

```
Enter an integer number: 5
The number 5 is odd
```

This program uses the *modulus* operator (`%`) to test whether or not the entered value is evenly divisible by two. If so, then the number is even; otherwise, it is odd. Note the use of the *double-equals* symbol (`==`) which in the C language means *checking for equality*. By contrast, a single-equals symbol (`=`) of the type used in mathematical calculations means *assigning a value*.

C provides a means for both `if` and `else` statements to trigger the execution of multiple instructions rather than just single instructions. Consider the following modification of the previous program, where each condition results in two `printf` instructions being executed:

```
#include <stdio.h>

int main (void)
{
  int n;

  printf("Enter an integer number: ");
  scanf("%i", &n);

  printf("The number %i is ", n);

  if (n % 2 == 0)
  {
    printf("even\n");
    printf("because there is no remainder\n");
  }

  else
  {
    printf("odd\n");
    printf("because a remainder exists\n");
  }

  printf("following division by two.\n");

  return 0;
}
```

Note how curly-braces are used to enclose the set of instructions associated with the `if` and `else` conditional statements.

Here we see the results of running this program:

Program input and output: (with 4 as the entered value)

```
Enter an integer number: 4
The number 4 is even
because there is no remainder
following division by two.
```

Program input and output: (with 5 as the entered value)

```
Enter an integer number: 5
The number 5 is odd
because a remainder exists
following division by two.
```

It is worth noting that an `if` statement may be used without a corresponding `else`. Take for example this simplified version of the program, which responds with text if the entered value is even but halts without announcement if the value is odd:

```
#include <stdio.h>

int main (void)
{
  int n;

  printf("Enter an integer number: ");
  scanf("%i", &n);

  if (n % 2 == 0)
    printf("The number %i is even\n", n);

  return 0;
}
```

On occasions where there needs to be several different courses of action depending on the condition, we may use multiple `if` statements as shown in the following program where a numerical entry is converted into its English word-equivalent:

```
#include <stdio.h>

int main (void)
{
  int n;

  printf("Enter an integer number: ");
  scanf("%i", &n);

  if (n == 0)
    printf("The number is zero\n");

  if (n == 1)
    printf("The number is one\n");

  if (n == 2)
    printf("The number is two\n");

  if (n == 3)
    printf("The number is three\n");

  return 0;
}
```

This program will function as intended for numerical entries of 0, 1, 2, or 3, but halts without announcement for any other entered value. A practical addition to the code would be an `else` statement to cover any entries other than 0, 1, 2, or 3, but we must be careful in its use as we shall soon see.

```
#include <stdio.h>

int main (void)
{
  int n;

  printf("Enter an integer number: ");
  scanf("%i", &n);

  if (n == 0)
    printf("The number is zero\n");

  if (n == 1)
    printf("The number is one\n");

  if (n == 2)
    printf("The number is two\n");

  if (n == 3)
    printf("The number is three\n");

  else
    printf("The number is something other than zero, one, two, or three\n");

  return 0;
}
```

The `else` statement functions as intended for a value outside the range of 0 to 3, in this case 10:

```
Enter an integer number: 10
The number is something other than zero, one, two, or three
```

The program also works like it should for an entered value of 3:

```
Enter an integer number: 3
The number is three
```

However, it fails for entered values of 0, 1, or 2 (in this case, 1):

```
Enter an integer number: 1
The number is one
The number is something other than zero, one, two, or three
```

Why does this program print contradicting messages only for values of 0, 1, or 2? The answer is related to the `else` statement, which associates *only* with the preceding `if` statement. That is to say, the `else` statement as placed in this program works perfectly in conjunction with the `if` statement testing for a value of 3, but it does not "know" to refrain from execution if any of the preceding `if` statements test true.

The solution to this problem is a third form of conditional statement called the `else if`. Examine this modified version of the program which works correctly for all entered values:

```
#include <stdio.h>

int main (void)
{
  int n;

  printf("Enter an integer number: ");
  scanf("%i", &n);

  if (n == 0)
    printf("The number is zero\n");

  else if (n == 1)
    printf("The number is one\n");

  else if (n == 2)
    printf("The number is two\n");

  else if (n == 3)
    printf("The number is three\n");

  else
    printf("The number is something other than zero, one, two, or three\n");

  return 0;
}
```

This is the proper way to test for a set of three or more mutually-exclusive conditions: the first conditional test is performed by an `if` statement, the last by an `else` statement, and all others in between by `else if` statements.

A common application of `if` conditionals is *limiting* a variable's value, as in this program:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float entry;

  printf("Enter a number: ");
  scanf("%f", &entry);

  if (entry < 0)
    entry = 0;

  printf("The square root of %f is %f\n", entry, sqrt(entry));

  return 0;
}
```

Any negative entry becomes "capped" at zero before computing the square root. Alternatively:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float entry;

  printf("Enter a number: ");
  scanf("%f", &entry);

  if (entry < 0)
    printf("That is an illegal entry!\n");   // Admonishes the user

  else                                       // Only computes valid numbers
    printf("The square root of %f is %f\n", entry, sqrt(entry));

  return 0;
}
```

A different type of conditional instruction offered by the C language is the `switch` statement, which is useful when sets of specific conditions lead to the same consequences. Consider the following example program, which tests a single keyboard character for its identity as either a vowel, a numeral, or something else:

```
#include <stdio.h>

int main (void)
{
  char key;

  printf("Enter a single character: ");
  scanf("%c", &key);

  switch(key)
  {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
      printf("%c is a vowel\n", key);
      break;
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      printf("%c is a numeral\n", key);
      break;
    default:
      printf("%c is a consonant or a symbol\n", key);
  }

  return 0;
}
```

The `break` instructions are necessary to exit the `switch` conditional as soon as a specified

condition is met. If you were to eliminate one of the `break` statements from this program, its execution would "fall through" to the next set of `case` conditions and actions.

Just for comparison, let's examine a program written using `if/else` statements rather than `switch/case` statements to perform the same task of identifying characters:

```
#include <stdio.h>

int main (void)
{
  char key;

  printf("Enter a single character: ");
  scanf("%c", &key);

  if (key == 'a' || key == 'e' || key == 'i' || key == 'o' || key == 'u')
    printf("%c is a vowel\n", key);

  else if (key == '0' || key == '1' || key == '2' || key == '3' || key == '4' ||
           key == '5' || key == '6' || key == '7' || key == '8' || key == '9')
    printf("%c is a numeral\n", key);

  else
    printf("%c is a consonant or a symbol\n", key);

  return 0;
}
```

In order to have multiple conditions result in common actions using `if` statements, we must string those conditions together using the logical "OR" operator (`||`). One could argue that this source code is messier for a human being to read than the previous version using `switch`, but it's really a matter of personal preference on the part of the person writing the code. Suffice it to say, `switch` lends itself more easily to multiple conditions having the same action, while `if` lends itself more easily to one condition per action.

3.13 Loops

Another useful capability of all computers is performing tasks in a *repeated* fashion. In programming parlance, any code specifying a repeated task or tasks is known as a *loop*. Two types of loops are most popular in C programming: the `while` loop and the `for` loop.

A *while* loop is demonstrated in this next program:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float x;

  while(1)
  {
    printf("Enter a positive number: ");
    scanf("%f", &x);

    printf("The square root of %f is %f \n\n", x, sqrt(x));
  }

  return 0;
}
```

When run, the program repeatedly asks us to input a positive number and then tells us the square root of that number. Here, we see it operating with the user-input values of 4, 8, and 2. Pressing the Control-C keys halts the program:

Program input and output:

```
Enter a positive number: 4
The square root of 4.000000 is 2.000000

Enter a positive number: 8
The square root of 8.000000 is 2.828427

Enter a positive number: 2
The square root of 2.000000 is 1.414214

Enter a positive number: ^C
```

The `while()` instruction in this program contains the value of 1 within its parentheses, and the lines of code to be "looped" are enclosed within another set of curly-brace characters ({ and }). Note how the floating-point variable declaration lies outside of this loop, as does the `return` instruction, because both of these need only be executed once. It is important that the variable `x` be declared prior to the `while` loop which uses that variable, and it is just as important for the `return` instruction to follow the `while` loop because the program's `main()` function exits whenever its `return` instruction executes.

All instructions contained within the `while` loop will be repeatedly executed so long as the numerical value within the `while` instruction's parentheses is non-zero. If we were to substitute `while(0)` for `while(1)` we would find that the program does nothing: its execution skips past all lines of code contained within the `while` loop's braces and goes immediately to the `return` instruction where it terminates. In other words, the instructions contained within the `while` loop will be repeated *while the argument is "true" in the Boolean sense of that word.*

It is worth noting that a `while` loop may contain just a single instruction, in which case no curly-braces are necessary to enclose it. Consider this trivial example:

```
#include <stdio.h>

int main (void)
{
  while(1)
    printf("Hello world!\n");

  return 0;
}
```

When run, the program repeatedly prints "Hello world!" to the console until we press Control-C to halt the program:

Program output:

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!^C
```

The conditional nature of the `while` loop suggests a way we could modify the square-root program to be self-terminating. Instead of having the `while` loop repeat forever with its constant argument of 1, we could include a conditional statement inside the parentheses that only lets the loop continue based on some condition controlled by the user. Consider the following example:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float x = 0.0;

  while(x < 100.0)
  {
    printf("Enter a positive number: ");
    scanf("%f", &x);

    printf("The square root of %f is %f \n\n", x, sqrt(x));
  }

  return 0;
}
```

Now the program only runs if the last number entered by the user is less than one hundred:

Program input and output:

```
Enter a positive number: 9
The square root of 9.000000 is 3.000000

Enter a positive number: 56
The square root of 56.000000 is 7.483315

Enter a positive number: 100
The square root of 100.000000 is 10.000000
```

Note how the program actually calculates the square root of 100 even though that value creates a "false" condition for the `while` loop, because the "truth" of that condition does not get checked until after the user enters the value *and* the square root gets calculated. In other words, any entered value greater than or equal to 100 gets processed just fine, and then only when the loop returns to the top for another pass does the `while` instruction recognize the condition as no longer being met.

The next form of loop we will study in the C programming language is the `for` loop. Whereas the `while` loop repeatedly executes so long as a single condition is met, the `for` loop is a more specific type of looping instruction designed to repeat one or more instructions a specific number of times, using a "counting" integer variable to track the number of executions. Again, the loop's syntax and general structure is easiest to grasp by viewing an example:

```
#include <stdio.h>

int main (void)
{
  int n;

  for(n = 0 ; n < 5 ; ++n)
    printf("Hello world!\n");

  return 0;
}
```

When run, the program prints "Hello world!" exactly five times and then halts on its own:

Program output:

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Just prior to the `for` instruction we see the declaration of an integer variable named `n`, and then within the `for` instruction's parentheses we see the first action taken before entering the loop which is initializing `n` to a value of zero, as well as incrementing the value of `n` at the end of each pass thorugh the loop (`++n`), and the conditional check at the start of each pass to see if the value of `n` is less than five (`n < 5`). The terms within the `for` instruction's parentheses always follow the same order:

```
for(pre-loop action ; conditional check ; action at end of each pass)
  // instruction(s) to be repeated
```

Like the `while` loop, a `for` loop may be used to repeat a set of instructions, in which case we must surround that set of instructions with curly-brace characters to denote their *all* being part of the same loop. Here we will modify our square-root program to use a `for` loop rather than a `while` loop:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  int n;
  float x;

  for(n = 3 ; n > 0 ; --n)
  {
    printf("Enter a positive number: ");
    scanf("%f", &x);

    printf("The square root of %f is %f \n\n", x, sqrt(x));
  }

  return 0;
}
```

Now the program only runs three times before halting on its own:

Program input and output:

```
Enter a positive number: 4
The square root of 4.000000 is 2.000000

Enter a positive number: 7
The square root of 7.000000 is 2.645751

Enter a positive number: 12
The square root of 12.000000 is 3.464102
```

Note the arbitrary decision to use a *decrementing* `n` variable within the `for` loop rather than an incrementing variable. This has no relation to the rest of the program's function, but is shown here merely to illustrate a different configuration of a `for` loop.

A popular use of `for` loops is to repeat a series of instructions where the counting variable is not only used to limit the number of executions but is also an independent variable in one or more mathematical functions being looped. Consider this example, where the `for` loop prints a table of numbers representing powers of ten:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  int n;

  for(n = -4 ; n < 5 ; ++n)
    printf("n = %i    10^n = %f \n", n, pow(10,n));

  return 0;
}
```

Program output:

```
n = -4    10^n = 0.000100
n = -3    10^n = 0.001000
n = -2    10^n = 0.010000
n = -1    10^n = 0.100000
n = 0    10^n = 1.000000
n = 1    10^n = 10.000000
n = 2    10^n = 100.000000
n = 3    10^n = 1000.000000
n = 4    10^n = 10000.000000
```

In these prior examples we have only explored the use of `for` loops employing integer test variables (`n`), but we are not limited to just using integers. We can in fact use other variable types such as `float` within `for` loops. The same three-part structure within the `for` parentheses holds true: (1) initialize the test variable, (2) check for a true/false condition to determine whether or not to continue looping, and (3) modify the value of that test variable somehow.

An important note is that the true/false condition checked within a `for` loop may actually be *any* condition whatsoever, even conditions completely unrelated to the test variable! It is quite commonplace to use `for` loops with the conditional check based on the value of the test variable, but this is not strictly necessary.

Consider the following program using a `for` loop that tests for the value of a calculated result rather than the value of the test variable:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  int n;
  float result = 0.0;

  for(n = -4 ; result < 500000 ; ++n)
  {
    result = pow(10,n);
    printf("n = %i    10^n = %f \n", n, result);
  }

  return 0;
}
```

Program output:

```
n = -4    10^n = 0.000100
n = -3    10^n = 0.001000
n = -2    10^n = 0.010000
n = -1    10^n = 0.100000
n = 0    10^n = 1.000000
n = 1    10^n = 10.000000
n = 2    10^n = 100.000000
n = 3    10^n = 1000.000000
n = 4    10^n = 10000.000000
n = 5    10^n = 100000.000000
n = 6    10^n = 1000000.000000
```

As before, the `for` loop's test variable begins at a value of -4 and increments by one with each pass through the loop. However, instead of continuing or halting the loop based directly on the value of that test variable `n` we instead decide whether or not to continue looping based on the calculated value `result`. If `result` is less than five hundred thousand (500000), then the loop continues; if otherwise, it halts.

Two important details are worth noting about this program and its output:

- First is that the program actually continues past a calculated value of 500000 – instead calculating all the way to one *million* – because the value of `result` doesn't get checked by the `for` loop until *after* it has been calculated and printed by the `printf` function. The conditional statement of any `for()` loop is checked prior to the next pass through the loop, not during a pass itself.
- Second is that you can see in the source code when we declare `result` as a floating-point variable we also initialize it to a starting value of zero. This is important because when any new variable gets declared in C its initial value can be anything at all! Newly-declared numerical variables *usually* start out having a value of zero, but this is not guaranteed because all the program does is reserve some random place in the computer's memory to store each declared variable, and if one of those memory locations just happens to have left-over data in it from some prior program's execution then the variable assigned to that memory location will begin its life holding that data!

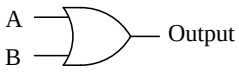
Note how there is no need to initialize `n` with some safe starting value because it's going to get initialized to -4 on the first execution of the `for` loop before it will have any influence on the value of `result`. By contrast, it is important to initialize `result` to a safe value prior to the `for` loop because on the first pass through the `for` loop the value of `result` will be checked (and therefore the loop may immediately terminate if `result` happens to hold a value greater than or equal to 500000) before `result` gets calculated by `result = pow(10,n)`.

Experienced software developers apply such *defensive programming* as a matter of professional habit. That is, you write code with a defensive attitude, anticipating conditions that might otherwise cause your program to fail. In this case, the anticipated condition is some value 500000 or greater randomly assigned to the variable `result` on the program's start-up. Another example of defensive programming seen in the section on Conditional instructions is anticipating improper user-entered values such as negative entries in a square-root calculator program. A well-written program should handle such events gracefully. Defensive programming is especially important for systems controlling critical processes, such as code running in digital automotive braking systems, code running in aircraft autopilot systems, etc.

3.14 Logical operators

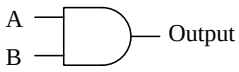
Computer programming languages in general are useful for implementing mathematical functions of all kinds, and a subset of that functionality includes *logical* functions such as AND, OR, and NOT. Students of electronics usually encounter these concepts when beginning their studies into digital circuits, committing certain single- and two-input logic functions to memory (both their symbols and their corresponding truth tables):

OR



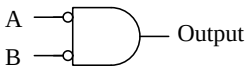
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

AND



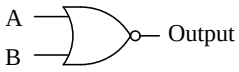
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Neg-AND



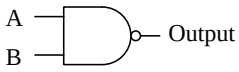
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

NOR



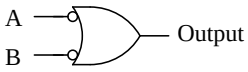
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

NAND



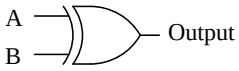
A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Neg-OR



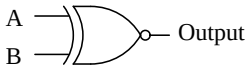
A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

XOR



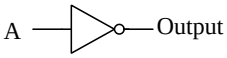
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

XNOR



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

NOT

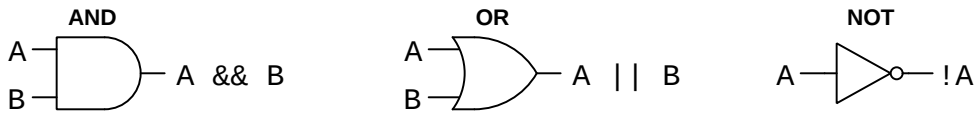


A	Output
0	1
1	0

Logical functions are just special mathematical functions operating on variables capable only of two values (or *states*) known as *true* (e.g. "high" or 1) and *false* (e.g. "low" or 0), and all computer programming languages support them. These are known as *Boolean* variables, after Boolean algebra where all values are either 1 or 0 with no other possibilities. Interestingly, the original C language standard did not offer a special Boolean data type even though logical functions were supported in C from the very beginning. Instead, logical functions in C simply use integer values as arguments,

interpreting a 0 value as "false" and anything other than 0 as "true". The C99 standard[8] introduced a `bool` variable type for the first time, but the logical functions themselves still treat all inputs as either zero (false) or non-zero (true) integer values.

Logical functions have their own special *operators* in the C language analogous to arithmetic operators such as addition (`+`), subtraction (`-`), etc. These include the AND operator (`&&`), the OR operator (`||`), and the NOT operator (`!`), as illustrated below using standard logic function symbols:



The following example showcases the use of logical operators and the `bool` data type (that data type defined in the `stdbool.h` header file):

```
#include <stdio.h>
#include <stdbool.h>

int main (void)
{
  bool a, b;

  a = false;
  b = true;

  printf("a AND b = %i \n", a && b);
  printf("a OR b = %i \n", a || b);
  printf("NOT a = %i \n", !a);

  return 0;
}
```

This example program produces the following output when compiled and executed:

```
a AND b = 0
a OR b = 1
NOT a = 1
```

[8] This is more officially known as ISO/IEC 9899:1999 (released in the year 1999), and was superseded by the C11 version which was released in 2011.

Note how we may initialize our Boolean variables `a` and `b` using "true" and "false" values, but the results of the AND, OR, and NOT functions are shown as numbers (either 1 or 0). The `%i` format specifier we use in the `printf()` statements prints the Boolean values as integer numbers, which in fact is how C treats all Boolean quantities. The following test program[9] and its output proves this fact, that the AND, OR, and NOT operators work just as well on standard integer values, treating zero as "false" and any non-zero value as "true":

```
#include <stdio.h>

int main (void)
{
  int a, b, c;

  a = 0;
  b = 5;
  c = -8;

  printf("a AND b = %i \n", a && b);
  printf("a AND c = %i \n", a && c);
  printf("b AND c = %i \n", b && c);
  printf("a OR b = %i \n", a || b);
  printf("b OR c = %i \n", b || c);
  printf("NOT c = %i \n", !c);

  return 0;
}
```

```
a AND b = 0
a AND c = 0
b AND c = 1
a OR b = 1
b OR c = 1
NOT c = 0
```

As far as the logical operators in this program are concerned, the equivalent Boolean value of `a` is "false" because it is equal to zero, while `b` and `c` are both considered "true" because they are non-zero. The integer values of 5 and -8 are effectively "rounded down" in translation to Boolean, and all that remains after evaluating the logical function is a 1 or 0 value.

[9] Note how this test program does not include the header file `stdbool.h` because nowhere do we need to declare or initialize any proper Boolean variables.

Looping and conditional statements assume Boolean arguments. A `while` statement, for example, continues to repeat all code within its braces ({ and }) so long as the value within its parentheses is logically true. Again, this translates to any non-zero integer value. In a previous section we saw how a `while` loop will repeat indefinitely if its argument is equal to 1, but we could just as well have given the `while` statement any value that wasn't equal to 0, such as:

```
#include <stdio.h>

int main (void)
{
  while (-10)
  {
    printf("Help me, I'm stuck in a loop! \n");
  }

  return 0;
}
```

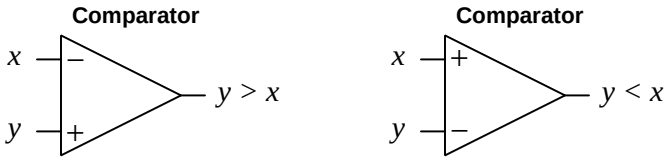
As far as `while` is concerned, −10 is just as "true" as 1.

For the same reason, an `if` conditional may act on any integer argument because it is really just testing for the logical "truth" of that proposition. For example:

```
#include <stdio.h>

int main (void)
{
  int a;

  printf("Enter a value for A: ");
  scanf("%i", &a);

  if (a)
    printf("A is true \n");

  else
    printf("A is false \n");

  return 0;
}
```

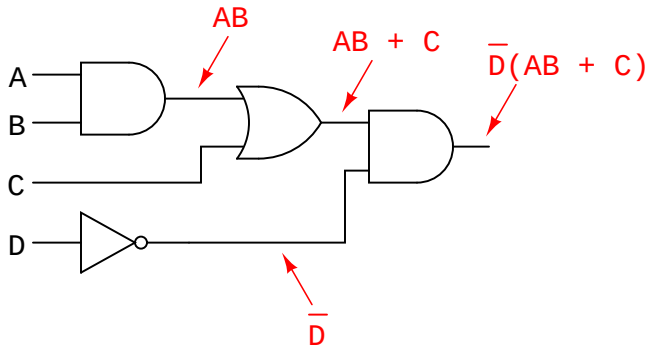
Running this program proves that any non-zero value entered for `a` is accepted as "true".

Relational operators such as "greater than" (>) and "less than" (<) may be modeled as electronic *comparators*, outputting a Boolean "true" or "false" value based on the comparison of two non-Boolean quantities just as comparator circuits generate a discrete (on/off) output signal based on the comparison of two analog signals:

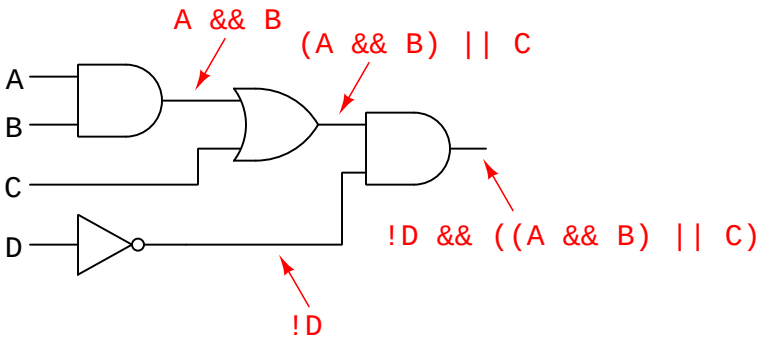



```
#include <stdio.h>

int main (void)
{
  float x, y;

  printf("Enter a value for X: ");
  scanf("%f", &x);

  printf("Enter a value for Y: ");
  scanf("%f", &y);

  if (y > x)
    printf("The statement y > x is true \n");

  else
    printf("The statement y > x is false \n");

  return 0;
}
```

Combinational logic is when multiple logic functions are ganged together to form more complex logical relationships. This topic is usually applied to physical logic gate circuits, but here we will explore how to translate a combinational function into C code. First we will examine a four-function combinational network complete with Boolean expressions[10] written at the output of each function:



Next, we will re-write those Boolean functions using standard C-language logical operators:



We use parentheses to force the desired order-of-operations much the same as we would in any regular algebraic expression where we wish certain operations to be evaluated before others.

[10] Recall that in Boolean algebra the OR function is equivalent to addition and the AND function is equivalent to multiplication. Just as it is mathematical convention that multiplication is performed before addition (standard order-of-operations), AND operations are executed before OR operations in C. The technical term for this prioritization of certain operations over others is *precedence*. Parentheses, however, take precedence over everything else: expressions within inner parentheses must always execute before expressions within higher-level parentheses.

The following program implements this combinational function with four user-entered inputs:

```
#include <stdio.h>

int main (void)
{
  int a, b, c, d;

  printf("Enter a value for A (1=true and 0=false): ");
  scanf("%i", &a);

  printf("Enter a value for B (1=true and 0=false): ");
  scanf("%i", &b);

  printf("Enter a value for C (1=true and 0=false): ");
  scanf("%i", &c);

  printf("Enter a value for D (1=true and 0=false): ");
  scanf("%i", &d);

  if (!d && ((a && b) || c))
    printf("Output is true \n");

  else
    printf("Output is false \n");

  return 0;
}
```

The ability to implement combinational logic networks in C code is perhaps most useful when we need the computer to follow a given truth table. Truth tables are convenient ways for humans to describe how they wish a certain logic system will behave, and these are fairly simple to translate into Boolean expressions which in turn (as we have seen) are fairly simple to translate into C code.

Let's see how this works on the following truth table:

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

When "true" states (1, high) represent the minority of output conditions for a combinational function, a good strategy for translating that table into a Boolean expression is to write a *product* expression for each of those "true" states and then combine them together to form a *sum-of-products* expression for the entire table. This is shown below in an expanded version of that same truth table:

A	B	C	Output	Boolean expression
0	0	0	0	
0	0	1	1	$\overline{A}\,\overline{B}C$
0	1	0	1	$\overline{A}B\overline{C}$
0	1	1	0	
1	0	0	0	
1	0	1	1	$A\overline{B}C$
1	1	0	0	
1	1	1	0	

$$\text{SOP expression} = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}C$$

Written using standard C logical operators:

```
(!A && !B && C) || (!A && B && !C) || (A && !B && C)
```

For truth tables where "true" outcomes are the majority instead of the minority, we obtain a simpler Boolean description for that table's function by writing either a *product-of-sums* expression or an inverted sum-of-products:

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$$\text{POS expression} = (A + \overline{B} + C)(\overline{A} + \overline{B} + \overline{C})$$

$$\text{Inverted SOP expression} = \overline{\overline{A}B\overline{C} + ABC}$$

Both of these Boolean expressions fully encapsulate the truth table, and indeed are mathematically equivalent to each other[11]. It is mostly a matter of personal preference which type of Boolean expression you use to represent the truth table. In my experience most students prefer the inverted SOP because it is so similar to SOP rather than feeling "backwards" like POS[12]. Next we see how each of these expressions appear in C:

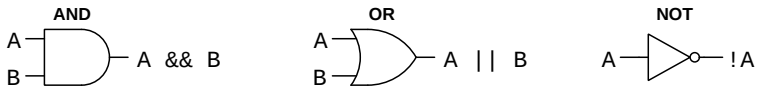
```
(A || !B || C) && (!A || !B || !C)

!((!A && B && !C) || (A && B && C))
```

[11] DeMorgan's Theorem may be used to "break" the inversion bars and show the equivalence, but this is left here as an exercise for the reader because it is not germane to the topic of coding logic functions in C.

[12] SOP expressions are based on "1" states where we write one product term for each "1" output (row) and the variables within each term follow "1" input states (i.e. we write an uncomplemented variable for a "1" state and a complemented variable for a "0" state). POS expressions are based on "0" states where we write a sum for each "0" output (row) and the variables within each sum follow "0" input states (i.e. each "0" input gets an uncomplemented variable and each "1" state gets a complemented variable).

3.15 Bitwise operators

As described in the previous section of this Tutorial, the C programming language supports all the basic logical functions as "operators" acting on Boolean variables:

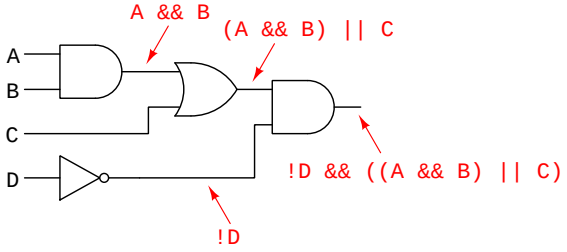


The following source code and run-time output illustrates these three basic logic functions as well as the `bool` data type (defined in the `stdbool.h` header file):

```
#include <stdio.h>
#include <stdbool.h>

int main (void)
{
  bool a, b;

  a = true;
  b = false;

  printf("a AND b = %i \n", a && b);
  printf("a OR b = %i \n", a || b);
  printf("NOT a = %i \n", !a);

  return 0;
}
```

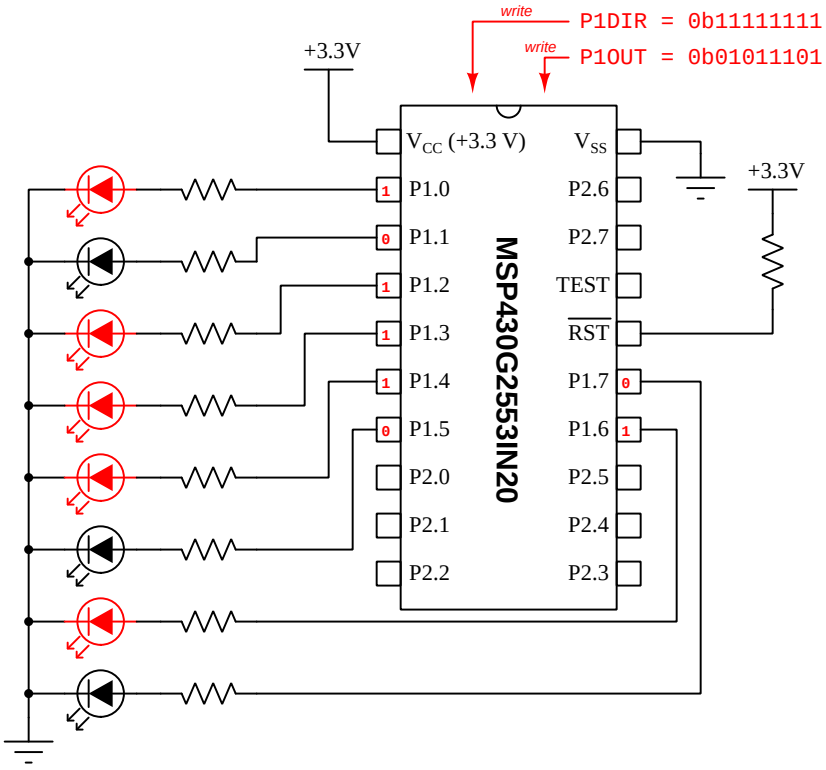
```
a AND b = 0
a OR b = 1
NOT a = 0
```

Combining simple logic functions to make complex *combinational* functions is as simple as combining basic arithmetic operators to make complex algebraic statements, using parentheses as grouping symbols to force any desired order-of-operations:



However, C is not limited to applying logical functions on discrete (Boolean) variables, but is also able to apply logical functions to individual bits in any binary word. This is called *bitwise* logic, and is usually applied to integer variables. There are many practical applications of bitwise logic in computer programming, and it finds common usage in microcontroller programming where the logical states of input and output lines are typically represented as individual bits in 8-bit or 16-bit registers.

For example, the Texas Instruments MSP430G2553 microcontroller uses a set of 8-bit registers to control the eight lines of each input/output (I/O) port. If we wish to use the eight lines of port 1 as outputs to drive LEDs, we must write the binary value `0b11111111` to the `P1DIR` register to set the "direction" of these eight lines to "output" (1), then we simply write whatever bit states we wish for those eight LEDs into a register named `P1OUT` as shown below:



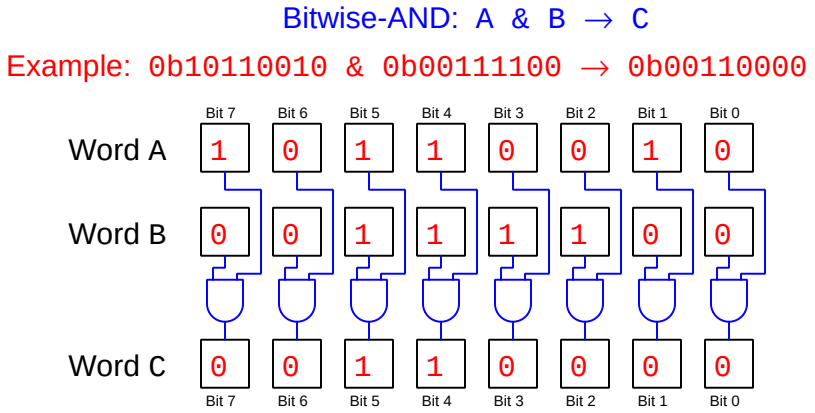
It is easy in C to set all eight bits of these registers with a single instructions (each), like this:

```
P1DIR = 0b11111111;
P1OUT = 0b01011101;
```

However, if we need our program to modify *just one or some of the bits* in either of these 8-bit words without altering or re-writing the others, we must use bitwise functions.

3.15.1 Bitwise-AND

An ampersand symbol (&) designates the *bitwise-AND* operation in the C language, applying AND logic to respective bits between two binary words of the same length:



Word B is commonly referred to as a *mask*, and serves a role analogous to masking tape used to mask off areas we do not wish to paint. Every bit-position occupied by a "1" in the mask will preserve the respective bit in Word A, while every bit-position occupied by a "0" in the mask will be "painted over" with a "0" state in the output. The key to understanding bitwise-AND masking is to realize that any "0" state input to an AND gate guarantees a "0" output regardless of the other input state(s). This makes the bitwise-AND function useful when we need to force individual bits to "0" states.

Referencing our MSP430 microcontroller example, suppose we wished to force bit 6 of the `P1OUT` register to a zero (logical "low") state to turn off that one LED, but not influence any of the other bits in that register. We could do so using the following C instruction, using a mask value with its bit 6 cleared to a zero state (forcing it to zero) and all other bits set to one (preserving their existing states):

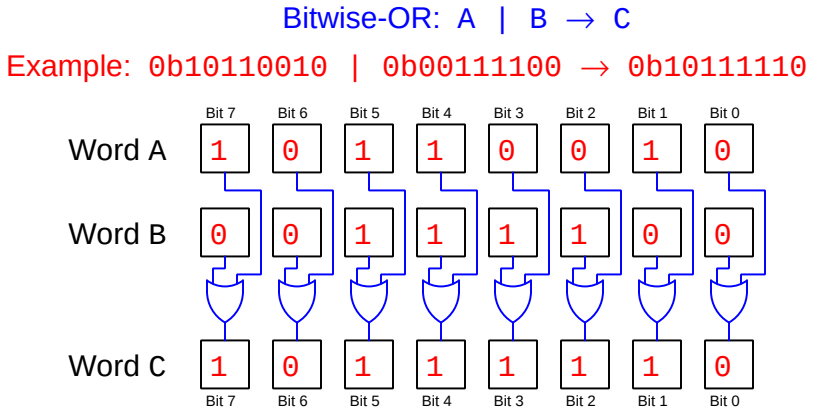
```
P1OUT = P1OUT & 0b10111111;
```

A condensed version of this instruction supported in C combines the bitwise-AND operator (&) and the assignment operator (=) into a single equivalent instruction:

```
P1OUT &= 0b10111111;
```

3.15.2 Bitwise-OR

A "vertical bar" symbol (|) designates the *bitwise-OR* operation in the C language, applying OR logic to respective bits between two binary words of the same length:



As with the bitwise-AND operator, Word B is commonly referred to as a *mask* and serves a similar role of forcing ("painting over") some bits while preserving ("masking off") others. For bitwise-OR, every bit-position occupied by a "0" in the mask will preserve the respective bit in Word A, while every bit-position occupied by a "1" in the mask will be "painted over" by a "1" state at the output. The key to understanding bitwise-OR masking is to realize that any "1" state input to an OR gate guarantees a "1" output regardless of the other input state(s). This makes the bitwise-OR function useful when we need to force individual bits to "1" states.

Referencing our MSP430 microcontroller example again, suppose we wished to force bits 0 and 6 of the `P1OUT` register to a one (logical "high") state to turn on those two LEDs, but not influence any of the other bits in that register. We could do so using the following C instruction, using a mask value with its bit 0 and bit 6 set to one states (forcing them to one) and all other bits set to zero (preserving their existing states):

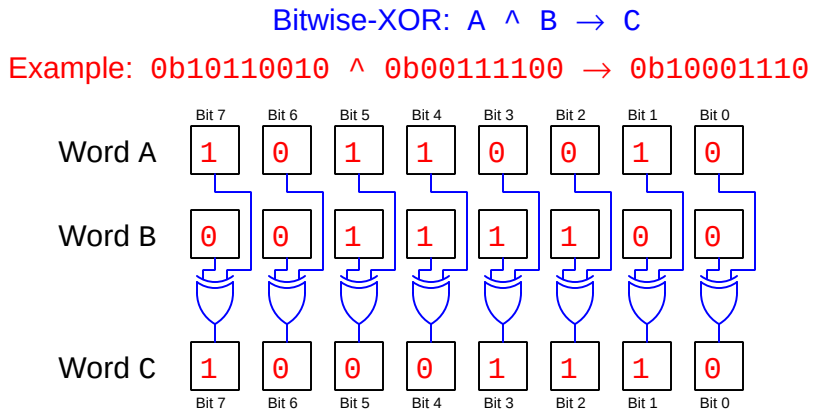
```
P1OUT = P1OUT | 0b01000001;
```

A condensed version of this instruction supported in C combines the bitwise-OR operator (|) and the assignment operator (=) into a single equivalent instruction:

```
P1OUT |= 0b01000001;
```

3.15.3 Bitwise-XOR

A "caret" symbol (^) designates the *bitwise-XOR* operation in the C language, applying Exclusive-OR logic to respective bits between two binary words of the same length:



As with the bitwise-AND and bitwise-OR operators, Word B is commonly referred to as a *mask*. In the case of bitwise-XOR, a "0" bit in the mask preserves that respective bit in Word A while a "1" bit in the mask *complements* that respective bit in Word A. This makes the bitwise-XOR function useful when we need to toggle individual bits from their current states to their opposite states.

Referencing our MSP430 microcontroller example again, suppose we wished to toggle the states of bits 3 and 4 and 7 in the `P1OUT` register but not influence any of the other bits. We could do so using the following C instruction, using a mask value with its bit 3 and bit 4 and bit 7 all set to a one state (enabling toggle action) and all other bits set to zero (preserving their existing states):

```
P1OUT = P1OUT ^ 0b10011000;
```

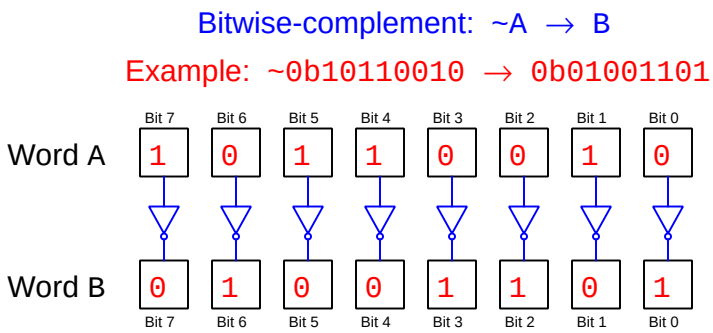
A condensed version of this instruction supported in C combines the bitwise-XOR operator (^) and the assignment operator (=) into a single equivalent instruction:

```
P1OUT ^= 0b10011000;
```

An interesting and widespread application of bitwise-XOR operations is found in *cryptography*, which is the science of encrypting messages to make them indecipherable to anyone but the intended recipient, who alone is able to decrypt the message to make it readable again. In such applications the XOR mask is referred to as a *key*, the bits within that key determining which bits within the message get inverted and which do not. If you imagine a binary-encoded message n bits in length, and a key of the same (n bits) length, that key is able to encrypt the "plaintext" message into "cyphertext" as well as decrypt the "cyphertext" message back into "plaintext" form. The security of this message depends on the key being securely shared between sender and recipient, just like a padlock shared between two people is secure only if those two people are alone in possessing keys fitting that padlock. Other bitwise-XOR-based schemes exist, such as repeating the use of a shorter key (e.g. an eight-bit key repeatedly used for each byte of data in the message), but they are less secure than one where the key is the same size as the message.

3.15.4 Bitwise complementation

Another useful bitwise operation provided by the C language is *complementation*, where we invert (toggle) the logical state of every bit in a word. This is also known as generating the *one's complement* of the binary number value represented by that word. The "tilde" symbol (~) designates this operation in the C language:



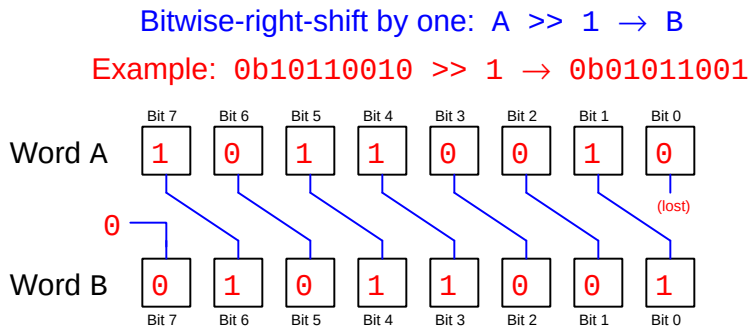
There is no "mask" word when using bitwise complementation because it very simply and direct inverts every bit state in the given word. In other words, this operation is equivalent to bitwise-XOR with a mask word comprised of all "1" bits.

Referencing our MSP430 microcontroller example again, suppose we wished to toggle the states of all bits in the `P1OUT` register. We could do so using the following C instruction:

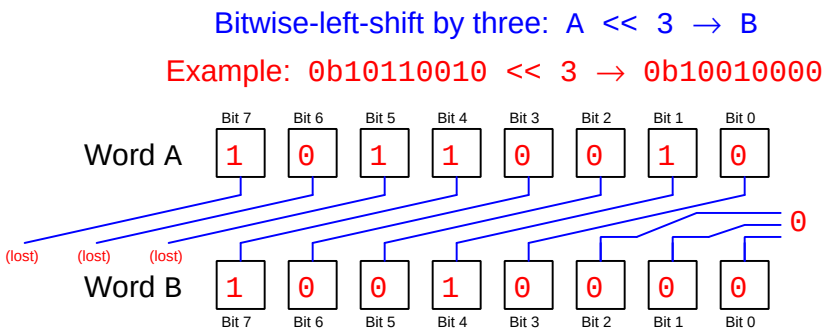
```
P1OUT = ~P1OUT;
```

3.15.5 Bit-shifting

Another useful bitwise instruction *shifts* the positions of bits within a binary word, using either the << (left-shift) operator or the >> (right-shift) operator symbol as seen in these examples:



Here we see bit 0 "lost" because there is no place for it in Word B once it's shifted off the right-end of Word A. On bit 7 of Word B we see that "new" bit state defaulting to "0".



As with the right-shift operation, we see here with left-shift that bits shifted off the end of the data field are lost while new bits are filled with "0" states.

One of the more interesting applications of bit-shifting is to perform integer multiplication and division by powers of two. Shifting a binary number's bits to the right is tantamount to dividing it by two, while shifting a binary number's bits to the left is the same as multiplying it by two[13] assuming there are enough bit-places in the resulting word that we do not "lose" any of the left-shifted "1" bits. For example, if we shift `0b00001100` (twelve) to the left one place, we get `0b00011000` (twenty-four) which is twice as large.

If we shift bits by more than one place, we effectively multiply or divide by that *power* of two. For instance, `0b00001100` (twelve) bit-shifted three places to the left yields `0b01100000` which is ninety-six, 2^3 times larger than twelve.

[13] This is because binary is a base-two numeration system, with each place-weight having a value two times less (right) or two times greater (left) than its adjacent place-weights. Analogously, shifting decimal digits one place to the right divides by ten while shifting decimal digits one place to the left multiplies by ten because decimal is a base-ten numeration system.

Since all microprocessors offer bit-shifting instructions at the "machine" level (i.e. a dedicated machine-language instruction just for that purpose), there is generally less translational work from C to machine language – i.e. fewer machine-language instructions resulting from the compilation and linking of the C source code – when bit-shifting as opposed to general multiplication and division. This makes bit-shifting a *computationally efficient* method of multiplying and dividing when the coefficient is a power of two, an especially useful property for microcontrollers with limited processing power.

3.15.6 Bitwise demonstration program

The following program illustrates all of these bitwise operations, taking in operands from the user[14] and displaying the results in hexadecimal notation:

```
#include <stdio.h>

int main (void)
{
  int n, mask;

  printf("Enter value of first operand in hexadecimal: ");
  scanf("%i", &n);

  printf("Enter value of second operand (i.e. the ‘mask’) in hexadecimal: ");
  scanf("%i", &mask);

  printf("\n");
  printf("Bitwise-AND between 0x%X and 0x%X = %X \n", n, mask, n & mask);
  printf("Bitwise-OR between 0x%X and 0x%X = %X \n", n, mask, n | mask);
  printf("Bitwise-XOR between 0x%X and 0x%X = %X \n", n, mask, n ^ mask);
  printf("Bitwise-complement of 0x%X = 0x%X \n", n, ~n);
  printf("Left-shift 0x%X by four bits = 0x%X \n", n, n << 4);
  printf("Right-shift 0x%X by three bits = 0x%X \n", n, n >> 3);

  return 0;
}
```

Compiling and executing this simple program displays the following results:

```
Enter value of first operand in hexadecimal: 0x57
Enter value of second operand (i.e. the ‘mask’) in hexadecimal: 0xF0

Bitwise-AND between 0x57 and 0xF0 = 50
Bitwise-OR between 0x57 and 0xF0 = F7
Bitwise-XOR between 0x57 and 0xF0 = A7
Bitwise-complement of 0x57 = 0xFFFFFFA8
Left-shift 0x57 by four bits = 0x570
Right-shift 0x57 by three bits = 0xA
```

To understand each of these results, write out the operands in binary format and apply the respective bitwise operations bit-by-bit, then convert that binary result into hexadecimal. Note that

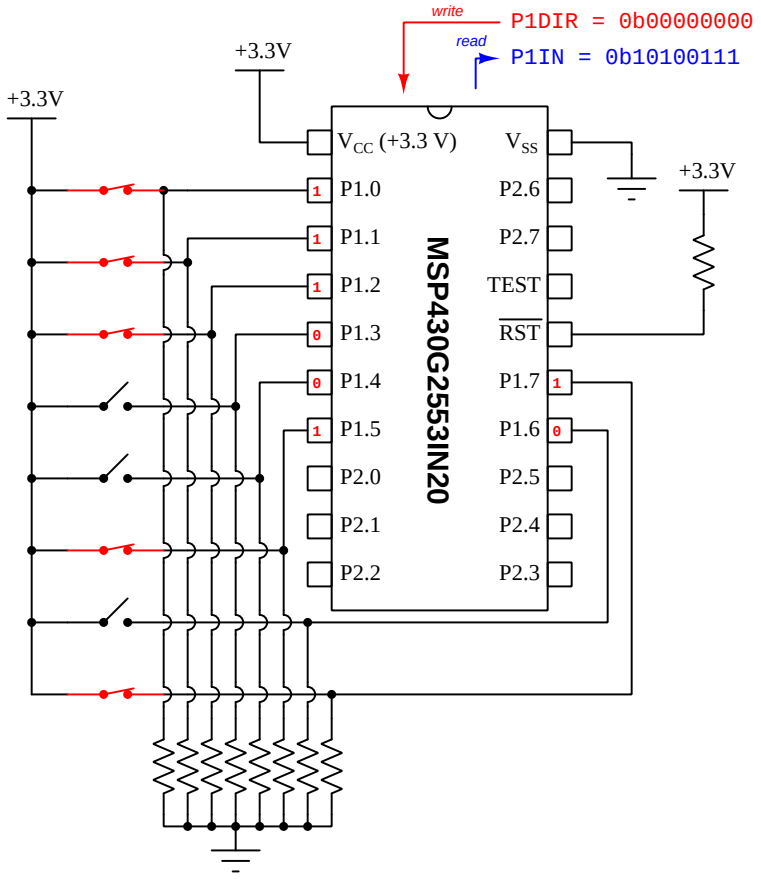
[14] Prefacing every numerical entry with `0x` instructs the program to interpret them as hexadecimal numbers. Decimal entries would have worked just as well, but bitwise relations are easier to comprehend in hexadecimal because each hex character directly expands to four binary bits.

the strange-looking result for bitwise complementation arises out of the fact that this example was created on a computer where the default integer word size is 32 bits, and so performing a one's complement inversion on `0x57` (binary value `0b01010111`) actually means doing so on `0b 0000 0000 0000 0000 0000 0000 0101 0111` which results in `0b 1111 1111 1111 1111 1111 1111 1010 1000` which of course is `0xFFFFFFA8`.

An excellent "active learning" exercise is to compile and run this program for yourself, entering your own operand values, predicting the bitwise results, and then using the program to test your predictions.

3.15.7 Testing bit states

Not only are bitwise functions useful for manipulating individual bit-states in binary words, but they also work well to *test* for the presence of specific bit-states. For example, consider another MSP430 microcontroller example where we use port 1 as inputs rather than outputs. In order to prepare the port 1's pins to function as inputs suitable for connection to switches and pulldown resistors, we would need the microcontroller's program to write `0b00000000` to the `P1DIR` register (recall that in the MSP430 microcontroller series a "1" state in a `P1DIR` register bit designates that pin to function as an output, and so we might logically infer that a "0" designates that pin to be an input):



With the eight switches in the positions shown, the data readable in the `P1IN` register would be `0b10100111`. It is easy enough to test for a particular *combination* of port 1 bits using a standard equality-check within an `if` statement like this:

```
if (P1IN == 0b10100111)
  // (do something here)
```

However, testing one or more selected bits within an integer word is not possible using the equality (==) relation because that result is only "true" if *every* bit of the word matches the test pattern. Suppose, though, we need to test only bit 5 (corresponding to input P1.5) for a "1" state while ignoring all the other inputs (bits) of P1IN? A practical application for such a test might be if the switch connected to input P1.5 needs to initiate some event or sequence of events in the microcontroller independent of the other input states, for example to serve as a "reset" for an internal counter or timer function within the microcontroller.

Selecting just one bit out of a multi-bit word sounds like the kind of task suited for a bitwise *masking* operation, and indeed it is. We may use the bitwise-AND function to "mask off" just the bit(s) we're interested in while setting all others to zero, the result being "true" if any of the masked bit(s) are "1". For example:

```
#include <stdio.h>

int main (void)
{
  int n;

  while(1)
  {
    printf("Enter value of the word to test: ");
    scanf("%i", &n);

    if (n & 0b00100000)
      printf("Bit 5 is HIGH!\n");

    else
      printf("Bit 5 is low\n");
  }

  return 0;
}
```

The test statement `n & 0b00100000` will be "true" for any values of `n` where bit 5 is a "1". The bitwise-AND operation with the mask value of `0b00100000` forces all the other bits to be zero during the test so that only bit 5 is left to be evaluated.

Entering hexadecimal values designed to set just one of eight bits "high" at a time:

```
Enter value of the word to test: 0x80
Bit 5 is low
Enter value of the word to test: 0x40
Bit 5 is low
Enter value of the word to test: 0x20
Bit 5 is HIGH!
Enter value of the word to test: 0x10
Bit 5 is low
Enter value of the word to test: 0x08
Bit 5 is low
Enter value of the word to test: 0x04
Bit 5 is low
Enter value of the word to test: 0x02
Bit 5 is low
Enter value of the word to test: 0x01
Bit 5 is low
```

Only when we enter `0x20` (which is equivalent to `0b00100000` in binary – with only bit 5 "high") is the statement `n & 0b00100000` considered logically "true". By using the bitwise-AND function with `0b00100000` as the mask value, we force to zero all bits except for bit 5 and then the `if` statement simply looks for a non-zero result as confirmation that the test is logically "true". Note that other input values would satisfy this "bit 5 high" condition as well, for example `0xFF`, `0xA0`, `0x37`, etc. (i.e. *any* number where the #5 bit is a "1" regardless of the other bits' states).

A popular strategy to make these bit-tests convenient is to use the `#define` preprocessor directive to define easy-to-understand constants reflecting the bit-positions. A `#define` directive instructs the compiler to perform a global search-and-replace operation, substituting one string of text for another. In the following example, the compiler replaces every instance of `BIT0` with `0b00000001`, every instance of `BIT1` with `0b00000010`, etc. prior to actually compiling the source code, allowing us to write code that is easier to understand:

```
#define BIT0 0b00000001
#define BIT1 0b00000010
#define BIT2 0b00000100
#define BIT3 0b00001000
#define BIT4 0b00010000
#define BIT5 0b00100000
#define BIT6 0b01000000
#define BIT7 0b10000000
```

If we compile and run this demonstration program, it works exactly the same as the previous version, but with an `if` condition that is easier to read in source form:

```
#include <stdio.h>

#define BIT0 0b00000001
#define BIT1 0b00000010
#define BIT2 0b00000100
#define BIT3 0b00001000
#define BIT4 0b00010000
#define BIT5 0b00100000
#define BIT6 0b01000000
#define BIT7 0b10000000

int main (void)
{
  int n;

  while(1)
  {
    printf("Enter value of the word to test: ");
    scanf("%i", &n);

    if (n & BIT5)
      printf("Bit 5 is HIGH!\n");

    else
      printf("Bit 5 is low\n");
  }

  return 0;
}
```

Pre-defined constants such as these are such an aid to building self-documenting programs (i.e. source code that other human programmers are able to read and understand easily due to how variables and constants are named) that we often find them defined within header files specific to the application. For example, the header files used for Texas Instruments MSP430 microcontrollers (e.g. `msp430g2553.h`) contain many such constants, some generically named for bit positions as these, and others named for unique flag bits found in special-purpose registers[15].

[15] For example, within the MSP430G2553's status register (`SR`) we find that bit 0 is the Carry bit, bit 1 is the Zero bit, bit 2 is the Negative bit, etc. So, in that microcontroller's header file we find the constant `C` defined as `0x0001`, `Z` defined as `0x0002`, `N` defined as `0x0004`, etc. That way, if we need to test the status of the Zero bit, for example, we may use the expression `SR & Z` to tell if just that bit is "true" while disregarding the rest. This is simpler than `SR & 0x0002` and even simpler than `SR & BIT1`.

If we need the program to check whether two or more particular bits are "high", we may do so by adding those `BIT` constants together. For example, `BIT1 + BIT5` results in the mask value `00100010` (i.e. bits 5 and 1 are both "high"). See how this works in the following program:

```
#include <stdio.h>

#define BIT0 0b00000001
#define BIT1 0b00000010
#define BIT2 0b00000100
#define BIT3 0b00001000
#define BIT4 0b00010000
#define BIT5 0b00100000
#define BIT6 0b01000000
#define BIT7 0b10000000

int main (void)
{
  int n;

  while(1)
  {
    printf("Enter value of the word to test: ");
    scanf("%i", &n);

    if (n & (BIT5 + BIT1))
      printf("Bit 5 and/or Bit 1 are HIGH!\n");

    else
      printf("Neither bit is high\n");
  }

  return 0;
}
```

```
Enter value of the word to test: 0x20
Bit 5 and/or Bit 1 are HIGH!
Enter value of the word to test: 0x02
Bit 5 and/or Bit 1 are HIGH!
Enter value of the word to test: 0x08
Neither bit is high
Enter value of the word to test: 0xFF
Bit 5 and/or Bit 1 are HIGH!
```

3.16 Functions

Large and complicated computer programs benefit from partitioning the code into separate "modules" which may be invoked by the main portion of the program as needed. In some programming languages this is known as a *subroutine*[16] but in C it is known as a *function*. We will explore this concept using pseudo-code examples to illustrate a certain sequence of operations:

Pseudo-code example without functions:

```
main()
{
  // Instruction A
  // Instruction B
  // Instruction C
  // Instruction B
  // Instruction C
  // Instruction B
  // Instruction C
}
```

Program execution: A → B → C → B → C → B → C

Now, we will examine another pseudo-code program that does the exact same sequence of operations only using a function named `B_and_C`:

Pseudo-code example using a function:

```
main()
{
  // Instruction A
  // Call B_and_C()
  // Call B_and_C()
  // Call B_and_C()
}

B_and_C()
{
  // Instruction B
  // Instruction C
}
```

[16] Literally, a routine within another routine.

Every time a function is "called" from within `main()`, the computer "jumps" to that called function and executes the code there, then resumes where it left off in the main routine.

Let's now explore this concept using real C code. What follows is a program determining whether or not resistors with a ±5% tolerance are within specification (without using any functions):

```
#include <stdio.h>

int main (void)
{
  float ideal, actual, tolerance=5.0, error;

  printf("Enter the resistor's labeled value (Ohms): ");
  scanf("%f", &ideal);

  while(1)
  {
    printf("\nEnter the resistor's measured value (Ohms): ");
    scanf("%f", &actual);

    error = (actual - ideal) / ideal * 100.0;

    if (error > tolerance)
      printf("Resistor value too high, error = %f percent\n", error);

    else if (error < -tolerance)
      printf("Resistor value too low, error = %f percent\n", error);

    else
      printf("Resistor value is within tolerance\n");
  }

  return 0;
}
```

Now consider this re-write of the program using a function named `errorcheck()`:

```
#include <stdio.h>

void errorcheck(float, float);

int main (void)
{
  float ideal, actual, tolerance=5.0, error;

  printf("Enter the resistor's labeled value (Ohms): ");
  scanf("%f", &ideal);

  while(1)
  {
    printf("\nEnter the resistor's measured value (Ohms): ");
    scanf("%f", &actual);

    error = (actual - ideal) / ideal * 100.0;

    errorcheck(error, tolerance);
  }

  return 0;
}

void errorcheck(float e, float t)
{
  if (e > t)
    printf("Resistor value too high, error = %f percent\n", e);

  else if (e < -t)
    printf("Resistor value too low, error = %f percent\n", e);

  else
    printf("Resistor value is within tolerance\n");
}
```

All the `printf` instructions are now located within the `errorcheck()` function rather than within the `main()` function, and are called as needed.

Let's analyze how the `errorcheck()` function works, dividing our analysis into three sections: one for each time we see `errorcheck` in the source code.

The first time we encounter this new function is near the top of the source code where it is *prototyped*. "Prototyping" is to a function what "declaring" is to a variable: a way of telling the compiler to reserve an area in the computer's memory for that entity. Later versions of the C language permit omission of function prototypes, but it is nevertheless a good programming practice to always prototype your custom functions.

In this prototype we can see that the function will take in two floating-point values as inputs (called *arguments*) and will output (*return*) no value at all as indicated by `void`.

The second time we see `errorcheck` in this program is when the function is "called" at the end of the `while` loop. Here, we "pass" the current values of variables `error` and `tolerance` to the `errorcheck()` function as its arguments. The computer's sequence of execution will now "jump" to the code contained within `errorcheck`'s curly-braces.

Here at the end of our source code listing we find the actual function named `errorcheck`. Notice how it is written much like `main` with the name and value types on one line and a pair of curly-braces enclosing all lines of code comprising the function. Note how the two arguments to the `errorcheck()` function are named `e` and `t`, not `error` and `tolerance` as called from within the `main()` function. These variables `e` and `t` are *local*[17] in scope to the `errorcheck()` function, which means they are readable and writable only within that function, and not anywhere else in the program. Essentially, the `void errorcheck(float e, float t)` line *declares* these two local variables and *assigns* them values from `error` and `tolerance`, respectively.

We may also use functions to perform mathematical operations and *return* values back to the calling function. On the next page we will see another version of this same program relegating the calculation of error to its own function (named `errorcalc()`).

[17] We could have named these two variables identically to the calling variables `error` and `tolerance`, and they would still be *local* to the `errorcheck()` function. This would mean, for example, that we could assign them new values from within the `errorcheck()` function, and these assignments would not affect the values of `error` or `tolerance` back within the `main()` function!

```
#include <stdio.h>

void errorcheck(float, float);
float errorcalc(float, float);

int main (void)
{
  float ideal, actual, tolerance=5.0, error;

  printf("Enter the resistor's labeled value (Ohms): ");
  scanf("%f", &ideal);

  while(1)
  {
    printf("\nEnter the resistor's measured value (Ohms): ");
    scanf("%f", &actual);

    errorcheck(errorcalc(actual, ideal), tolerance);
  }

  return 0;
}

void errorcheck(float e, float t)
{
  if (e > t)
    printf("Resistor value too high, error = %f percent\n", e);

  else if (e < -t)
    printf("Resistor value too low, error = %f percent\n", e);

  else
    printf("Resistor value is within tolerance\n");
}

float errorcalc(float a, float i)
{
  return (a - i) / i * 100.0;
}
```

Each time we remove code from the `main()` function and place that code into its own dedicated function, we simplify the code remaining inside of `main()`. This is the major purpose of using functions: to relegate portions of code to their own domains and let the (simpler) main routine of the program "call" those functions as needed. For this resistor tolerance-checking program the use of functions is rather trivial, but in large programs functions make the program source code much easier to understand and to manage as the project evolves over time.

The use of functions to partition code into modules has several advantages when writing large and complicated programs, some of those advantages listed here:

- Writing programs consisting of functions naturally supports the problem-solving strategy of breaking a complex problem down into manageable pieces, then solving each of those pieces one at a time.
- When a team of programmers work together to write code for a large programming project, it becomes possible to assign each team one or more functions which eventually will be linked together to form a whole program. Each team need only know what values will be passed to their function(s) and what value(s) will be returned after execution.
- Instead of one source-code file, we may write multiple source-code files – each one containing one or more functions – which makes file management easier when developing large projects. An example of this is shown on the next page.

Source code file `main.c`

```
#include <stdio.h>

int primecheck(int);

int main (void)
{
  int input;

  printf("Enter a positive integer to check if it is prime: ");
  scanf("%i", &input);

  if (primecheck(input) == 0)
    printf("The number %i is prime\n\n", input);

  else
    printf("The number %i is not prime\n\n", input);

  return 0;
}
```

Source code file `myfunction.c`

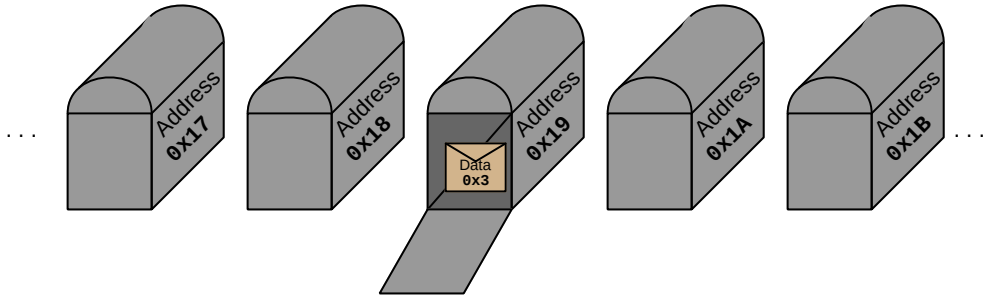
```
#include <stdio.h>

int primecheck (int n)
{
  int count, divided=0;

  for (count = n - 1 ; count > 1 ; --count)
    if (n % count == 0)
      ++divided;

  return divided;
}
```

In order to compile these *two* source-code files, we must invoke `GCC` as follows:

```
gcc main.c myfunction.c
```

3.17 Pointers

A powerful feature of the C programming language is that it allows us to access the addresses where data is stored in the computer's memory. Recall that digital memory relies on an *address* number to locate the stored data within the memory array, analogous to the street address number used to locate a postal mailbox:



In this illustration, the mailbox labels are the addresses, and the letter(s) contained in each mailbox are the data. All variables in C are stored in the computer's memory, and as such have addresses. The one opened mailbox represents data `0x3` stored at address `0x19`.

When a C program we have written declares a certain type of variable, that variable gets assigned a memory location at run-time (i.e. when the compiled program is executed). As programmers we generally do not get to choose which memory location this is, as the assignment process is automatic. However, within the C program it is possible to read that assigned memory address and act upon it in different ways.

In order to identify the address for any declared variable in a C program, we preface the variable's name with the "address of" operator (`&`). For example, if a variable happens to be named `x`, the address of that variable may be found with the label `&x`. This use of the ampersand symbol should look familiar to anyone who has ever used the `scanf()` function, as the following code example shows (accepting a floating-point numerical value from the user and placing that value in variable `x`):

```
scanf("%f", &x);
```

Now that we know what the "address-of" operator does, the `scanf()` instruction's proper syntax becomes a little less mysterious. Unlike `printf()` where we need to pass arguments to the function to be printed to the screen, with `scanf()` we are asking this function to insert data *into* a variable. It would not do any good to pass the existing value of the variable to `scanf()`, but it makes perfect sense to tell `scanf()` *where* that variable resides in memory so that `scanf()` knows where to write the collected data. This is analogous to telephoning a friend to send a package to your new home address: the information you send to your friend is simply a street address, and in response your friend sends the package to that address.

The following program demonstrates C's ability to read memory addresses by declaring and initializing two integer variables with identical data (5), then printing both the variables' values and their respective memory addresses[18]:

```
#include <stdio.h>

int main (void)
{
  int m, n;

  m = n = 5;

  printf("The value %i is stored at address %p\n", m, &m);
  printf("The value %i is stored at address %p\n", n, &n);

  return 0;
}
```

Program output:

```
The value 5 is stored at address 0xbfc7f39c
The value 5 is stored at address 0xbfc7f398
```

If we were to run this compiled program several times, we would find the addresses different every time. This proves how variable addresses are allocated at run-time and are not specified by the source code in any way. Likewise, the allocated addresses are completely independent of the values used to initialize the variables. We could have set `m` and `n` equal to any valid integer values and the addresses would be arbitrarily chosen at run-time all the same.

[18] Note the use of the `%p` formatting identifier for the address. This identifier instructs `printf()` to display a memory address in hexadecimal format rather than a regular integer value. Also note how the addresses displayed by this program at run-time consist of eight hexadecimal characters. Since each hex character is shorthand for four binary bits, this means our program is running in a computing environment using 32-bit memory addressing.

Not only does C allow us to probe the memory location (i.e. address) where any variable is stored, but it also provides a special type of variable which may be used to store those address values. This special variable type is called a *pointer*, because the value it contains "points" to the address of another variable.

Pointer variables are declared the same as any other variable type, but with an asterisk symbol (*) prefacing the variable's name so that the compiler is aware of what type of variable the pointer will be pointing to[19]. For example, if we wished to declare a pointer named `addr` for an integer-type variable, the declaration line would read:

```
int *addr;
```

Let's examine another short program to explore how pointers work:

```
#include <stdio.h>

int main (void)
{
  int n;
  int *pn;

  pn = &n;
  n = -329;

  printf("n = %i \n", n);
  printf("Pointer address = %p \n", pn);
  printf("Value stored at address %p is %i \n", pn, *pn);

  return 0;
}
```

Program output:

```
n = -329
Pointer address = 0xbfc17538
Value stored at address 0xbfc17538 is -329
```

[19]This knowledge is very important for pointer operations, because different types of variables require different amounts of memory (e.g. `char` variables are 8-bit, while `int` variables are typically 16- or 32-bit.).

The first two lines within this program's `main()` function declare the variables used, which is to say it instructs the computer to reserve specific locations in its memory to store these values. `int n` declares an integer variable named `n`, and `int *pn` declares a pointer variable named[20] `pn` that will point to an integer variable's memory address.

Next, the program initializes these two variables. First we set the pointer `pn` equal to the address of `n` using the "address of" operator (`&`), and then we set the value of `n` equal to an arbitrary value of `-329`. The order of these two initializations is irrelevant, because the address of `n` is fixed with that variable's declaration. Thus, we may set the pointer equal to `n`'s address at any time regardless of `n`'s content.

The three `printf()` statements following these initializations perform the following tasks:

- Print the integer value of `n`
- Print the pointer's (`pn`) value which is the address where `n` is stored in memory
- Print the value of `n` again, this time by *de-referencing* the pointer `pn`

In the C language, a pointer is not just useful for storing the address of another variable, but by prepending the `*` operator to the pointer variable's name we may summon the value stored at that "pointed" address. This means practically anything possible in C by using the variable `n` is also possible by using the de-referenced pointer `*pn`.

If you examine the pointer's value (`0xbfc17538`) you will see it consists of eight hexadecimal characters, which is equivalent to 32 bits. This is because this program was compiled and run on a computer having a 32-bit memory address space. If the system were 64-bit we would expect to see a 16-character hexadecimal value (address) for the pointer, which brings us to an interesting feature of pointers: they are a special type of variable, the size of which depends on the memory space in which the executing program operates. Also, since pointers contain memory addresses, the pointer for a character variable (`char`) will be the same size as the pointer for an integer number (`int`) or for a floating-point number (`float`), even though those different types of variables are not the same number of bits each.

Similarly, the value of a pointer is based solely on the memory address of the variable it was initialized with. If the value of `n` in our program happens to change, `pn` will still contain the same address. Conversely, if we manipulate the value of `pn`, it will simply point to a different location in memory without `n` changing value.

[20] There is no formal reason for naming the pointer "`pn`" if we intend to use it to point to the memory address where `n` is stored. It's just that pointers can be a confusing concept, and so it is good to choose a naming convention that clearly distinguishes pointers from their respective variables while still denoting the association between them. In this case we are prepending the letter "p" onto the variable's name in order to designate that variable's pointer: `n` is an integer variable, and `pn` is variable pointing to the address where integer `n` resides in the computer's memory. The compiler, which cares not for our confusion nor for naming conventions, simply knows `pn` is a pointer for an integer number because it was declared `int *pn`. `n` and `pn` only become associated at the following line (`pn = &n;`) where the pointer `pn` is set equal to the address of `n`.

While admittedly confusing, pointers make it possible to write very memory-efficient and elegant code in the C language. They also expand the capability of *functions* in ways that are very powerful. To understand this, it is first imperative to understand an intentional limitation of C functions: arguments are treated on a *pass-by-value* basis. Recall that an "argument" to a function in C is a constant or variable found within the parentheses immediately following the function name: e.g. if we wrote a function named `cube()` to compute the cube of a number and called that function using the statement `cube(x)`, the value stored within variable x would be that function's sole argument. In C, every function receives a *copy* of each argument's value to act upon, and is unaware of the actual location of the argument in memory. To quote Brian Kernighan and Dennis Ritchie from their book *The C Programming Language*:

> . . . function arguments are passed by value, that is, the called function receives a private, temporary copy of each argument, not its address. This means that the function cannot affect the original argument in the calling function. Within a function, each argument is in effect a local variable initialized to the value with which the function was called. `[page 71]`

The alternative to pass-by-value is *pass-by-reference*, where the actual variable itself is passed along to the function when listed as an argument to that function. This would give the function full control over the passed variable, rather than just act upon whatever *value* that variable had stored in it at the time the function was called. Dennis Ritchie, the inventor of C, opted to make his language's functions pass-by-value which is simple but in some cases limiting. One of the applications of pointers is to circumvent this limitation: if we make a pointer one of the arguments to a function, that function will then "know" the memory location of the "pointed" variable, and thereby be able to edit the original variable rather than just work with a mere copy of that variable's value.

Consider the following program as an example of how pass-by-value works, using a simple function designed to increment the argument by one:

```
#include <stdio.h>

void increment (int);

int main (void)
{
  int count = 5;

  printf("Value is equal to %i\n", count);

  increment(count);   // "Call" the increment() function,
                      // "passing" count as the argument

  printf("Value is equal to %i\n", count);

  return 0;
}

void increment (int copy)
{
  copy = copy + 1;
}
```

Program output:

```
Value is equal to 5
Value is equal to 5
```

As you can see, the value of `count` is unaffected after the `increment()` function is called: it begins with a value of 5 and remains at a value of 5, even though the "private, temporary" version of it (`copy`) obviously gets incremented by one within the `increment()` function. If C happened to operate on a *pass-by-reference* basis, the `increment()` function's local variable `copy` would actually *become* the variable `count` and thereby be able to influence its value under that alias.

Now consider this pointer-based version of the same program, initializing a pointer variable to the `count` integer's memory address and then passing that pointer to the `increment()` function:

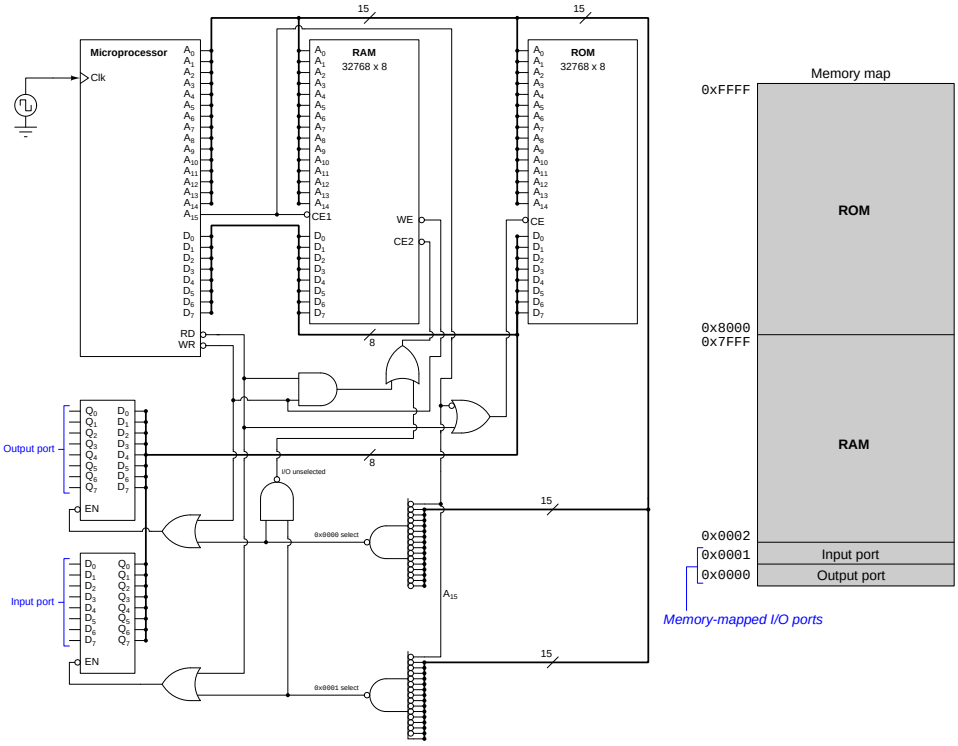
```
#include <stdio.h>

void increment(int *);

int main (void)
{
  int count = 5;
  int *point;
  point = &count;

  printf("Value is equal to %i\n", count);

  increment(point);   // "Call" the increment() function,
                      // "passing" point as the argument

  printf("Value is equal to %i\n", count);

  return 0;
}

void increment (int *copy)
{
  *copy = *copy + 1;
}
```

Program output:

```
Value is equal to 5
Value is equal to 6
```

`point`, of course, contains the memory address of `count`, and as the argument to the `increment()` function serves to pass its value (i.e. the address of `count`) to the "private, temporary" pointer named `copy` inside of that function. Within the function, that local pointer is de-referenced so that the incrementing function will act upon the data stored at that address (i.e. acting upon the value of `count`), and so when flow returns to the `main()` function we find `count` has actually incremented. Thus, pointers allow us to write functions acting on the argument variable(s) while still following C's "pass-by-value" policy for function arguments.

Another use of pointers is for referencing specific memory addresses in the computer system's hardware, such as *memory-mapped I/O*. This is where special functions provided by hardware modules use dedicated memory locations for their operation. An example of memory-mapped I/O appears in the next illustration, showing a partial schematic diagram for a simple 8-bit microprocessor system and its associated "memory map":



In this system two "octal" D-type latch ICs provide eight input and eight output lines where external devices (e.g. switches connected to inputs and LED indicator lamps to outputs) interface with the computer's eight-bit data bus. The two NAND logic gates enable one of these D-type latches when the correct address value appears on the microprocessor's 16-bit address bus. In this particular system, the output latch is enabled for the address value `0x0000` while the input latch is enabled for the address value `0x0001`. Thus, the 8-bit output data word "resides" at address `0x0000` and the 8-bit input data word "resides" at address `0x0001`, almost as though these 8-bit words were ordinary data stored in RAM rather than representing real-world bit signals as they really do.

A program written in C for this computer system could use an 8-bit integer to refer to the output data word, and another 8-bit integer to refer to the input data word, but if we simply declared two 8-bit integers the compiler would assign those integers to arbitrary locations in RAM memory. In order to be useful as I/O data, however, we require these two integers be associated with RAM memory addresses `0x0000` and `0x0001`. Therefore, we would have to declare and initialize *pointers* to these memory addresses in order to associate them with these specific addresses in memory. This type of "low-level" access to hardware is what makes the C programming language well-suited for embedded systems programming and other applications where the peculiarities of specific hardware

platforms must be accommodated in code.

For example, the following two lines of C code assign pointers named `pIN` and `pOUT` to the memory addresses of the I/O registers shown in the previous schematic diagram:

```
uint8_t * const pIN = (uint8_t *) 0x0001;
uint8_t * const pOUT = (uint8_t *) 0x0000;
```

Since each of these I/O registers is exactly eight bits in size, we use the 8-bit unsigned integer data type `uint8_t` for these pointers. Note also how we *cast* the hexadecimal values `0x0001` and `0x0000` to this same 8-bit pointer type using `(uint8_t *)`. The `const` qualifier declares the pointer as a constant which may be read but not written to. Mind you, the data input or output at each I/O port can and will vary over time as the computer runs, but each *pointer* will be properly fixed at its respective address.

Once declared and initialized as shown, the de-referenced `pIN` and `pOUT` pointers may be used to read and write data from and to these I/O registers:

```
// Testing bit 0 of input port for being in a high state
if (*pIN & 0b00000001)
  // (Then do something!)

// Setting bit 7 high on output port
*pOUT |= 0b10000000;
```

Pointers have many other uses in general-purpose programming too, but most of those require knowledge of different types of data which we will explore in subsequent sections.

3.18 Arrays

In mathematics we often use letters of the alphabet to represent *variables* and *constants*. However, due to the limited number of letters in any given alphabet there are many circumstances where there just aren't enough of them to adequately represent all the variables and/or constants we have in mind, or at least not enough of them to do so in a way that makes for sensible notation. For example, when mathematically analyzing multi-resistor electrical networks we typically use the capital letter R as the variable for *resistance*, but instead of using a different letter to represent each resistor's value we instead use *subscripts* to differentiate all the different resistor values from one another (e.g. R_1, R_2, R_3, etc.).

In the C language we also use letters of the alphabet to represent variables and constants, and indeed we may use whole words as unique variable names (e.g. `count`, `index`, `resistance`). However, C also provides a means of declaring a series of variables with the same lettered name, the variables in each series differentiated from each other by a numerical value (enclosed in bracket symbols) analogous to a mathematical subscript. For example, instead of R_1, R_2, and R_3 as we might see in a mathematical formula, in a C program we could use `R[1]`, `R[2]`, and `R[3]`.

Such a series of variables is called an *array*. The simple program shown below declares an array of three floating-point variables, then uses them to calculate an average:

```
#include <stdio.h>

int main (void)
{
  float value[3];
  int count;

  for (count = 0 ; count < 3 ; ++count)
  {
    printf("Enter value number %i: ", count);
    scanf("%f", &value[count]);
  }

  printf("Average = %f \n", (value[0] + value[1] + value[2])/3);

  return 0;
}
```

Program output:

```
Enter value number 0: -32.7
Enter value number 1: 2.004
Enter value number 2: 0.321
Average = -10.125000
```

As no doubt you can discern from the code listing, a three-element array is declared with the numerical value of *three* within the brackets of the declaration line, but actually consists of elements numbered 0, 1, and 2 because the element numbers always begin with zero. Just like single-variable data types, it is possible for us to declare and initialize an array with a single instruction, as shown in the following sample program using the same values entered by the user in the previous program:

```
#include <stdio.h>

int main (void)
{
  float value[3] = {-32.7, 2.004, 0.321};

  printf("Average = %f \n", (value[0] + value[1] + value[2])/3);

  return 0;
}
```

These three initializing values, of course, get assigned to the `value` array such that:

- `value[0] = -32.7`
- `value[1] = 2.004`
- `value[2] = 0.321`

Arrays aren't just useful as a method to represent subscripted variables, although they do an admiral job of that. One of the unique properties of arrays in C is that they *always* occupy sequential memory addresses, and this allows them to be accessed very efficiently using *pointers*. Here is a simple program using the `%p` pointer format specifier and the "address of" operator (`&`) within each `printf()` instruction to show memory addresses for the five integer variables within this array:

```
#include <stdio.h>

int main (void)
{
  int n[5];

  printf("Address of n[0] is %p \n", &n[0]);
  printf("Address of n[1] is %p \n", &n[1]);
  printf("Address of n[2] is %p \n", &n[2]);
  printf("Address of n[3] is %p \n", &n[3]);
  printf("Address of n[4] is %p \n", &n[4]);

  return 0;
}
```

Every time we run this program we will see different memory addresses displayed for the five variables, but those addresses values will always be separated by the same interval of four:

Program output:

```
Address of n[0] is 0x7fffc64fb040
Address of n[1] is 0x7fffc64fb044
Address of n[2] is 0x7fffc64fb048
Address of n[3] is 0x7fffc64fb04c
Address of n[4] is 0x7fffc64fb050
```

The reason for an address interval of four is because this example program was compiled and run on a 32-bit computer. This makes the default word size of an integer variable (`int`) 32 bits, or *four bytes* each.

If we modify the program to use *long* integers rather than normal integers for this array, we will see that the address interval will be doubled to eight bytes (64 bits):

```
#include <stdio.h>

int main (void)
{
  long n[5];

  printf("Address of n[0] is %p \n", &n[0]);
  printf("Address of n[1] is %p \n", &n[1]);
  printf("Address of n[2] is %p \n", &n[2]);
  printf("Address of n[3] is %p \n", &n[3]);
  printf("Address of n[4] is %p \n", &n[4]);

  return 0;
}
```

Every time we run this program we will see different memory addresses displayed for the five variables, but those address values will always be separated by the same interval of eight because 64 bits is eight bytes:

Program output:

```
Address of n[0] is 0x7fff9176fa70
Address of n[1] is 0x7fff9176fa78
Address of n[2] is 0x7fff9176fa80
Address of n[3] is 0x7fff9176fa88
Address of n[4] is 0x7fff9176fa90
```

If we declare an integer-type pointer variable, and then initialize that variable to the first element of the array, we may display those values using simple *pointer arithmetic*:

```
#include <stdio.h>

int main (void)
{
  int n[3] = {5, -7, 0};
  int *pnt;

  printf("Using subscripts:\n");
  printf("Value of n[0] is %i \n", n[0]);
  printf("Value of n[1] is %i \n", n[1]);
  printf("Value of n[2] is %i \n", n[2]);

  pnt = &n[0];

  printf("\nUsing pointers:\n");
  printf("Value of n[0] is %i \n", *(pnt));
  printf("Value of n[1] is %i \n", *(pnt + 1));
  printf("Value of n[2] is %i \n", *(pnt + 2));

  return 0;
}
```

As you can see, the printed results are identical:

Program output:

```
Using subscripts:
Value of n[0] is 5
Value of n[1] is -7
Value of n[2] is 0

Using pointers:
Value of n[0] is 5
Value of n[1] is -7
Value of n[2] is 0
```

A very important characteristic about pointer arithmetic is that any offset we add to or subtract from a pointer increments or decrements that pointer by however many bytes is necessary to go to the next element of the array. As we saw previously, if our array uses 32-bit variables (e.g. `int`) the address values stored in each pointer will be spaced at intervals of four bytes (32 bits), but if we use `long` integers instead the address values are spaced at intervals of eight bytes (64 bits). In

either case, though, we may use the same pointer arithmetic to increment from element to element – i.e. `pnt + 1` will always go to the *next memory address appropriate to the data type*. This, in fact, is why we need to specify a data type when declaring a pointer variable, so that the C compiler is aware of the type of variable that pointer will reference. This "type awareness" makes pointer arithmetic much simpler than if we needed to tell a pointer exactly how many bytes to advance for the next variable. The following program shows an example with two different types of integers:

```
#include <stdio.h>

int main (void)
{
  int m[3] = {0, 9, 4};
  int *pshort;
  pshort = &m[0];

  printf("Value of m[0] is %i at address %p \n", *pshort, pshort);
  printf("Value of m[1] is %i at address %p \n", *(pshort + 1), pshort + 1);
  printf("Value of m[2] is %i at address %p \n", *(pshort + 2), pshort + 2);

  long n[3] = {5, 7, 2};
  long *plong;
  plong = &n[0];

  printf("Value of n[0] is %li at address %p \n", *plong, plong);
  printf("Value of n[1] is %li at address %p \n", *(plong + 1), plong + 1);
  printf("Value of n[2] is %li at address %p \n", *(plong + 2), plong + 2);

  return 0;
}
```

As you can see, the same pointer arithmetic is useful for retrieving the stored data despite differences in data types and the number of bytes used in memory for each integer:

Program output:

```
Value of m[0] is 0 at address 0x7fff8a1cd130
Value of m[1] is 9 at address 0x7fff8a1cd134
Value of m[2] is 4 at address 0x7fff8a1cd138
Value of n[0] is 5 at address 0x7fff8a1cd110
Value of n[1] is 7 at address 0x7fff8a1cd118
Value of n[2] is 2 at address 0x7fff8a1cd120
```

Another way to handle pointer arithmetic is to use increment and/or decrement operators rather than absolute offsets added to or subtracted from the pointer. We may see these techniques at work in the following example program, using both prefix and postfix[21] operators:

```
#include <stdio.h>

int main (void)
{
  int m[3] = {0, 1, 2};
  int *pshort;
  pshort = &m[1];               // Initialize pointer to middle of array (m[1])

  printf("%i\n", *pshort);      // Show value stored in m[1]
  printf("%i\n", *(++pshort));  // Increment pointer and then show value (m[2])
  printf("%i\n", *(--pshort));  // Decrement pointer and then show value (m[1])
  printf("%i\n", *(pshort--));  // Show value (m[1]) and then decrement pointer
  printf("%i\n", *(pshort--));  // Show value (m[0]) and then decrement pointer

  return 0;
}
```

Program output:

```
1
2
1
1
0
```

This program deserves careful observation and reading of the comments.

[21] The "prefix" operator takes the form `++x` and increments the value before using that incremented value for the next operation. The "postfix" operator takes the form `x++` and uses the value *before* incrementing it. On a humorous note, the *C++* language is a pun on the postfix increment operator, the idea being that in the C++ language we first make use of all the basic syntax and capabilities of C and then augment it with additional features. Thus, C++ is really an *extension* of the C language.

A final and very important note about pointers and their use with arrays within the C language is that the name of an array is actually a pointer to its first element. This means the following two C instructions are functionally identical:

```
pshort = &m[0];
pshort = m;        // "m" is actually a pointer to m[0]
```

You will often find pointers initialized in C using the latter technique, capitalizing on the fact that an array's name is a pointer itself. An important caveat here is that an array's name is a *constant* pointer and therefore not subject to pointer arithmetic. That is to say, we are free to increment and decrement `pshort` at will, but we cannot change the value of `m` because it is a constant fixed at run-time when the computer first assigns memory addresses to all the array elements.

A very useful capability of arrays in the C language is that they may have more than one dimension. So far all the arrays we have explored were one-dimensional, which is to say they each had just one subscript. However, we can easily declare and use multi-subscript arrays such as in the following example showcasing a three-by-three array storing the integer numbers 1 through 9 in sequence:

```
#include <stdio.h>

int main (void)
{
  int x[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
  int n, m;

  for (n = 0 ; n < 3 ; ++n)
  {
    for (m = 0 ; m < 3 ; ++m)
    {
      printf("%i ", x[n][m]);
    }

    printf("\n");
  }

  return 0;
}
```

Program output:

```
1 2 3
4 5 6
7 8 9
```

3.19 Structures

A *structure* in the C programming language is a collection of variables, like an *array*, except that unlike arrays which must consist of a collection of variables of exactly the same type, a structure may encompass multiple types of variables. As a collection of multiple variables, often of mixed type, a structure really represents a *custom data type* that you may create in C.

```
#include <stdio.h>

int main (void)
{
  // Defining a new data structure type called "resistor" consisting
  // of two floating-point variables and one integer variable:
  struct resistor
  {
    float value;
    float tolerance;
    int stock;
  };

  struct resistor R1;  // Declaring an instance of that data type named "R1":

  // Here we initialize the elements of R1 with constant values:
  R1.value = 2.7e3;
  R1.tolerance = 0.05;
  R1.stock = 479;

  printf("We have %i resistors in stock that are ", R1.stock);
  printf("%g Ohms plus or minus ", R1.value);
  printf("%g percent.\n", R1.tolerance * 100.0);

  return 0;
}
```

Program output:

```
We have 479 resistors in stock that are 2700 Ohms plus or minus 5 percent.
```

Note how the data structure's definition is enclosed in curly-braces, and only after this definition is the compiler ready to declare an instance of this newly-defined structure. Once declared, we access each of the structure's *elements* (`value`, `tolerance`, and `stock`) using a dot symbol (`.`) separating the instance's name from the element names.

Shown here is another simple C program defining the same "`resistor`" data structure type and then declaring *two* instances of that type:

```
#include <stdio.h>

int main (void)
{
  struct resistor       // Defining the data structure type
  {
    float value;
    float tolerance;
    int stock;
  };

  struct resistor R1;   // Declaring an instance of that new type
  struct resistor R2;   // Declaring another instance of that new type

  R1.stock = 479;
  R2.stock = 210;

  printf("We have %i more R1 resistors in stock ", R1.stock - R2.stock);
  printf("than R2 resistors.\n");

  return 0;
}
```

Program output:

```
We have 269 more R1 resistors in stock than R2 resistors.
```

Once a new data type has been defined, we may declare as many instances of that type as we wish. Note how in this example we neither initialize nor use any of the nominal value or tolerance elements of `R1` or `R2`, but only the integer "stock" elements. Bear in mind that declared but uninitialized variables, whether stand-alone or as part of a larger data structure, may contain random values and should not be trusted for any program control functions.

Yet another way to declare instances of structures is to add the declarations immediately at the end of the structure definition, prior to the terminating semicolon. Below we see an example of this, which is a modification of the previous program, declaring structures `R1` and `R1`:

```
#include <stdio.h>

int main (void)
{
  struct resistor
  {
    float value;
    float tolerance;
    int stock;
  } R1, R2;         // Declaring R1 and R2 here

  R1.stock = 479;
  R2.stock = 210;

  printf("We have %i more R1 resistors in stock ", R1.stock - R2.stock);
  printf("than R2 resistors.\n");

  return 0;
}
```

These C program examples using structures to store attributes of various electrical resistors should make us wonder, *is it possible to create an array of structures in case we have a great many different resistors to catalogue?* The answer to this question is yes, as shown by the following program which creates a four-element[22] array of resistors all based on the same data structure type:

```
#include <stdio.h>

int main (void)
{
  struct resistor
  {
    float value;
    int stock;
  };

  struct resistor R[4];
  float sum;

  R[1].value = 2.7e3;
  R[1].stock = 479;
  R[2].value = 1.0e3;
  R[2].stock = 210;
  R[3].value = 7.9e3;
  R[3].stock = 1085;

  sum = ((R[1].value * R[1].stock)
       + (R[2].value * R[2].stock)
       + (R[3].value * R[3].stock));

  printf("If we connected all our in-stock resistors in series,\n");
  printf("the total resistance would be %g Ohms.\n", sum);

  return 0;
}
```

Program output:

```
If we connected all our in-stock resistors in series,
the total resistance would be 1.00748e+07 Ohms.
```

[22] This array contains structures `R[0]`, `R[1]`, `R[2]`, and `R[3]`. As we do not normally begin labeling resistors beginning with zero, though, this program only makes use of the last three.

Just like normal arrays, an array of structure types may also be initialized in the same line(s) of code used to declare. Consider this variation of the previous program:

```
#include <stdio.h>

int main (void)
{
  struct resistor
  {
    float value;
    int stock;
  };

  struct resistor R[4] = {0.0   , 0   ,
                          2.7e3 , 479 ,
                          1.0e3 , 210 ,
                          7.9e3 , 1085};
  float sum;

  sum = ((R[1].value * R[1].stock)
       + (R[2].value * R[2].stock)
       + (R[3].value * R[3].stock));

  printf("If we connected all our in-stock resistors in series,\n");
  printf("the total resistance would be %g Ohms.\n", sum);

  return 0;
}
```

Program output:

```
If we connected all our in-stock resistors in series,
the total resistance would be 1.00748e+07 Ohms.
```

Note how the initialization of the `R[]` array has four sets of value/stock constants shown, the first set consisting of zero and zero. This is due to the fact that any time we declare an array in C, the range of index values always begins at `[0]`. Thus, the `R[4]` four-element array consists of `R[0]`, `R[1]`, `R[2]`, and `R[3]`. Even though we're only using `R[1]` through `R[3]` later in the program, bulk initialization of the array demands we populate `R[0]` as well.

Not surprisingly, *pointers* may be used to reference structures in a manner similar to how they are used to reference arrays. An interesting difference with structures is that C introduces a new "arrow" operator (`->`) specifically for this purpose. Examine the following program to see four different ways we may reference elements of a structure, using either dot operators or arrow operators:

```
#include <stdio.h>

int main (void)
{
  struct resistor
  {
    float value;
    int stock;
  };

  struct resistor R1 = {2.7e3 , 479};
  struct resistor *pR1;

  pR1 = &R1;  // Initializing pointer pR1 to the first address of structure R1

  // Accessing elements of R1 using dot operators:
  printf("Resistance value = %g\n", R1.value);
  printf("Number in stock = %i\n", R1.stock);

  // Accessing elements of R1 using pointer and dot operators:
  printf("Resistance value = %g\n", (*pR1).value);
  printf("Number in stock = %i\n", (*pR1).stock);

  // Accessing elements of R1 using "address-of" and arrow operators:
  printf("Resistance value = %g\n", (&R1)->value);
  printf("Number in stock = %i\n", (&R1)->stock);

  // Accessing elements of R1 using pointer and arrow operators:
  printf("Resistance value = %g\n", pR1->value);
  printf("Number in stock = %i\n", pR1->stock);

  return 0;
}
```

When run, the four different sets of `printf()` instructions output the same results: the resistor value followed by the in-stock quantity.

Just as pointers may be incremented to index different elements of an array, here pointers may also be incremented to index different structure elements in an array of structures:

```
#include <stdio.h>

int main (void)
{
  struct resistor
  {
    float value;
    int stock;
  };

  struct resistor R[4] = {0.0   , 0   ,
                          2.7e3 , 479 ,
                          1.0e3 , 210 ,
                          7.9e3 , 1085};
  struct resistor *ptr;
  int n;

  ptr = &R[0];  // Initializing pointer to the first address of structure R[0]

  for (n = 0 ; n < 4 ; ++n)
  {
    printf("Elements of structure R[%i]:\n", n);
    printf("Resistance value = %g\n", ptr->value);
    printf("Number in stock = %i\n\n", ptr->stock);

    ++ptr;
  }

  return 0;
}
```

Program output:

```
Elements of structure R[0]:
Resistance value = 0
Number in stock = 0

Elements of structure R[1]:
Resistance value = 2700
Number in stock = 479

Elements of structure R[2]:
Resistance value = 1000
Number in stock = 210

Elements of structure R[3]:
Resistance value = 7900
Number in stock = 1085
```

3.20 Unions

Another custom data type in C is the *union* which appears quite similar in form to a structure but serves an entirely different purpose. Rather than binding multiple variables (and often, multiple data types) together into one data structure, the union reserves a location in the computer's memory large enough to hold the *largest* data type it contains, but allows access to all other data types under the same union name.

In the following example program we define a union type consisting of a 16-bit unsigned integer together with an 8-bit unsigned integer. Being a union, the compiler reserves 16 bits for it which is large enough to hold either the 16-bit integer outright or the 8-bit integer with eight more bits of "padding":

```
#include <stdio.h>
#include <stdint.h>

int main (void)
{
  union sample       // Defining the union
  {
    uint16_t x;      // 16-bit unsigned integer
    uint8_t y;       // 8-bit unsigned integer
  };

  union sample z;    // Declares one instance of the union named "z"

  z.x = 45;          // Initializes z.x to a value of 45

  printf("The value stored in X is: %i\n", z.x);
  printf("The value stored in Y is: %i\n", z.y);

  z.x = 1299;        // Initializes z.x to a value of 1299

  printf("The value stored in X is: %i\n", z.x);
  printf("The value stored in Y is: %i\n", z.y);

  return 0;
}
```

When we run this program, we see that storing the value of 45 into the union `z` results in that same value being displayed for `z.x` as well as for `z.y`. However, when we try storing the much larger value of 1299 into the union, we see a sharp difference between the 16-bit versus 8-bit representations of that number:

Program output:

```
The value stored in X is: 45
The value stored in Y is: 45
The value stored in X is: 1299
The value stored in Y is: 19
```

To understand what is happening at the computer's level, we must break down each of these quantities into their raw binary representations. The number 45 in binary is `0b101101`, a 6-bit value that easily fits into either an 8-bit or a 16-bit field. However, the number 1299 in binary is `0b10100010011`, an 11-bit value that fits easily into a 16-bit field but not into an 8-bit field. If we examine the least-significant eight bits of `0b10100010011`, however, we see they form the value 19 (`0b00010011`). In other words, the 8-bit integer `z.y` simply refers to the *last eight bits* of the quantity stored within this union.

This is one of the useful applications of a union in C: to view portions of a larger data type.

Another application of unions in C programming is to swap the order of bytes and words within variables read from one computer system to another. One of the supremely annoying idiosyncrasies of digital data formatting is that different computer system designs may store the same multi-byte numbers in different byte orders. For example, suppose a hypothetical computer stored a 32-bit floating-point value as four consecutive bytes in memory, which we will represent in this discussion as ABCD. A different computer might be designed to store that same four-byte value with the byte-pairs swapped (BADC), with 16-bit words swapped (CDAB), or both words and bytes swapped (DCBA). The consequence of these different byte-orderings between computers is that when one computer reads the 32-bit value from the other via a network there may be a need to perform byte-swaps and/or word-swaps to ensure the data is properly understood.

The following program shows how unions may be used to break down a 32-bit number into individual bytes, and then swap and reassemble those bytes in different orders, printing out the corresponding floating-point representations of each:

```
#include <stdio.h>
#include <stdint.h>

int main (void)
{
  union
  {
    uint32_t word;
    uint8_t byte[4];
  } in;

  union
  {
    uint32_t word;
    uint8_t byte[4];
    float real;
  } out;

  in.word = 0x1234CDEF;

  //  Maintaining the original order of the bytes (ABCD)
  out.byte[0] = in.byte[0]; // A
  out.byte[1] = in.byte[1]; // B
  out.byte[2] = in.byte[2]; // C
  out.byte[3] = in.byte[3]; // D
  printf("ABCD ordered hexadecimal value is %X\n", out.word);
  printf("ABCD ordered floating-point value is %f\n\n", out.real);

  //  Byte-swapping according to the pattern BADC
  out.byte[0] = in.byte[1]; // B
  out.byte[1] = in.byte[0]; // A
  out.byte[2] = in.byte[3]; // D
  out.byte[3] = in.byte[2]; // C
  printf("BADC ordered hexadecimal value is %X\n", out.word);
  printf("BADC ordered floating-point value is %f\n\n", out.real);

  //  Word-swapping according to the pattern CDAB
  out.byte[0] = in.byte[2]; // C
  out.byte[1] = in.byte[3]; // D
  out.byte[2] = in.byte[0]; // A
```

```
  out.byte[3] = in.byte[1]; // B
  printf("CDBA ordered hexadecimal value is %X\n", out.word);
  printf("CDAB ordered floating-point value is %f\n\n", out.real);

  //  Byte-swapping AND word-swapping according to the pattern DCBA
  out.byte[0] = in.byte[3]; // D
  out.byte[1] = in.byte[2]; // C
  out.byte[2] = in.byte[1]; // B
  out.byte[3] = in.byte[0]; // A
  printf("DCBA ordered hexadecimal value is %X\n", out.word);
  printf("DCBA ordered floating-point value is %f\n\n", out.real);

  return 0;
}
```

Program output:

```
ABCD ordered hexadecimal value is 1234CDEF
ABCD ordered floating-point value is 0.000000

BADC ordered hexadecimal value is 3412EFCD
BADC ordered floating-point value is 0.000000

CDBA ordered hexadecimal value is CDEF1234
CDAB ordered floating-point value is -501368448.000000

DCBA ordered hexadecimal value is EFCD3412
DCBA ordered floating-point value is -127014752317184782918557368320.000000
```

The ability of a union to interpret a common 32-bit field as either a single 32-bit integer, as four 8-bit integers (bytes), or as a single 32-bit floating-point number makes the task of byte- and word-swapping very simple and easy to understand. This program loads the given 32-bit value into the `in` union, then uses a series of assignment operations to copy each of the 8-bit bytes of that given value to different bytes of the `out` union, and then displays each of the `out` union's values as a 32-bit floating-point number.

3.21 Operating system calls

On occasion it is useful to write a program in C that "calls" a function native to the operating system under which the C program runs. For example, in the Linux operating system the command-line instruction `free` provides a display of available memory in the computer, while the instruction `df` lists all the filesystems currently "mounted" by the operating system. If we wished to write a program to execute either of these commands, a function within the `stdlib` library of the C language called `system()` allows us to do just that:

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

int main (void)
{
  int choice;

  while(1)
  {
    printf("Enter 1 to show memory, any other value to show filesystems: ");
    scanf("%i", &choice);

    if (choice == 1)
      system("free");

    else
      system("df");

    printf("\n");
  }

  return 0;
}
```

The `system()` expects a string-type argument, which may either be a string variable or a literal command enclosed in quotation marks as we have done in the example code. When executed, the `system()` function invokes the enclosed operating system command where the result gets reported to the standard output (console) of the operating system.

Literally any operating system utility or program available on that operating system may be thus invoked by your C program. This makes the `system()` function extremely powerful.

Demonstrating this program, we first enter `1` and then on the second pass we enter `5`:

Program output:

```
Enter 1 to show memory usage, any other value to show mounted filesystems: 1
               total        used        free      shared  buff/cache   available
Mem:        16055068     1020648    13994348      134192     1516840    15034420
Swap:         999420           0      999420

Enter 1 to show memory usage, any other value to show mounted filesystems: 5
Filesystem     1K-blocks     Used Available Use% Mounted on
udev             7986496        0   7986496   0% /dev
tmpfs            1605508     1804   1603704   1% /run
/dev/nvme0n1p2 489634808 43377368 421311932  10% /
tmpfs            8027532        0   8027532   0% /dev/shm
tmpfs               5120       12      5108   1% /run/lock
/dev/nvme0n1p1    523248     5972    517276   2% /boot/efi
tmpfs            1605504       48   1605456   1% /run/user/1000
```

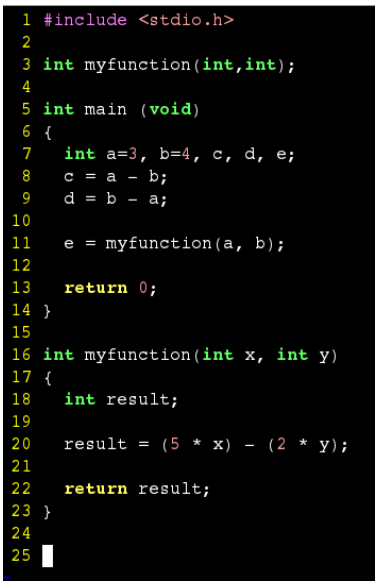
When used in the context of a modern multi-tasking operating system, the `system()` function empowers your C program to start up other applications that will then run concurrently with your program, though to do so without pausing the execution of your own program will require you invoke the utility or application to run in the console's "background" rather than in the foreground[23]. Applications are really only limited to your imagination, but may include things like:

- Allowing your program to start up an audio playback application to create sounds for the user to hear
- Allowing your program to start up networking applications (e.g. file transfer using `sftp`), utilities (e.g. enabling or blocking TCP ports using the `iptables` command), or diagnostics (e.g. `ping`)
- Allowing your program to terminate other programs (e.g. `kill`) or even shut off the computer (e.g. `shutdown`)
- Allowing your program to run pre-written script(s) containing multiple console commands

[23] In the Bash console shell which is very common for Linux operating systems, placing an ampersand character after the command-line invocation instructs the computer to run the command in the background so it does not commandeer the console's standard I/O. For example, typing `sleep 3` will cause the console to "go to sleep" for 3 seconds and be unresponsive to any keyboard input during that sleep time, but typing `sleep 3 &` seems to do nothing because the pause action takes place in the background while returning the console immediately back to you.

3.22 Debugging

A very powerful tool available in all modern C compiler suites is something called a *debugger*, which (among many other things) allows you to monitor the operation of your program as it executes line-by-line through the source code. In this section I will show some of the capabilities of the GNU Debugger (known as GDB), which is the companion debugging application to the popular GCC compiler. We will begin by viewing the source code of a very simple C program:



```
 1 #include <stdio.h>
 2
 3 int myfunction(int,int);
 4
 5 int main (void)
 6 {
 7   int a=3, b=4, c, d, e;
 8   c = a - b;
 9   d = b - a;
10
11   e = myfunction(a, b);
12
13   return 0;
14 }
15
16 int myfunction(int x, int y)
17 {
18   int result;
19
20   result = (5 * x) - (2 * y);
21
22   return result;
23 }
24
25
```

This screenshot shows the line-numbering option turned on in my text editor, where each line of code is numbered and shown in yellow on the left-hand edge of the display. These yellow numbers are not part of the source code, but merely a display feature of the text editor. Your text editor may look slightly different than mine, but should still provide an option to automatically number each line of code in the source file. This line numbering is very important for command-line debuggers, as it provides a reference to let you know where it is in the program's execution.

When compiling source code for a debugging session, a special option must be turned on to instruct the compiler to include debugging data within the executable file. When using GCC to compile the code, this will be the `-g` option:

```
gcc -g source.c
```

Without the `-g` option activated when compiling, the executable file will not have all the information that the debugger software needs to track the program's execution and relate that to specific lines of source code at run-time.

After compiling the source code and creating an executable file (named `a.out` by default when using GCC), you start GDB by typing `gdb a.out` at the command line. On this and the next page is a recording of a simple GDB debugging session conducted on this simple program:

```
GNU gdb (GDB) 7.5
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-slackware-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/tony/a.out...done.
(gdb) break main
Breakpoint 1 at 0x80483f5: file junk.c, line 7.
(gdb) run
Starting program: /home/tony/a.out

Breakpoint 1, main () at junk.c:7
7   int a=3, b=4, c, d, e;
(gdb) display a
1: a = -1208299532
(gdb) display b
2: b = 134513787
(gdb) display c
3: c = -1207963660
(gdb) display d
4: d = -1208298620
(gdb) display e
5: e = -1209667219
(gdb) s
8   c = a - b;
5: e = -1209667219
4: d = -1208298620
3: c = -1207963660
2: b = 4
1: a = 3
(gdb) s
9   d = b - a;
5: e = -1209667219
4: d = -1208298620
3: c = -1
2: b = 4
1: a = 3
(gdb) s
```

```
11   e = myfunction(a,b);
5: e = -1209667219
4: d = 1
3: c = -1
2: b = 4
1: a = 3
(gdb) s
myfunction (x=3, y=4) at junk.c:20
20   result = (5 * x) - (2 * y);
(gdb) display x
6: x = 3
(gdb) display y
7: y = 4
(gdb) display result
8: result = 134518564
(gdb) s
22   return result;
8: result = 7
7: y = 4
6: x = 3
(gdb) s
23 }
8: result = 7
7: y = 4
6: x = 3
(gdb) s
main () at junk.c:13
13   return 0;
5: e = 7
4: d = 1
3: c = -1
2: b = 4
1: a = 3
(gdb) s
14 }
5: e = 7
4: d = 1
3: c = -1
2: b = 4
1: a = 3
(gdb) s
0xb7e455a5 in __libc_start_main () from /lib/libc.so.6
(gdb) s
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 2061) exited normally]
```

Let's dissect this debugging session one command at a time.

```
GNU gdb (GDB) 7.5
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-slackware-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/tony/a.out...done.
(gdb)
```

After start-up, the command-line prompt always begins with `(gdb)` to let you know you're "inside" of a GDB debugging session and not accessing the regular console command line. My first typed command issued to GDB is `break main` which instructs GDB to insert a *breakpoint* into the program which will prompt it to halt execution and await further input before proceeding with the execution:

```
(gdb) break main
Breakpoint 1 at 0x80483f5: file junk.c, line 7.
```

Without any breakpoints set up, the program will simply run from beginning to end without stopping. The whole point of using a debugger is to halt the program in mid-execution to examine how it is functioning (or malfunctioning!), and so we need to specify some condition(s) for GDB to halt execution. Note that a breakpoint may be specified in ways other than by function name as I did here, including by line number in the source code file. Next, I enter `run` which tells GDB to begin running the program:

```
(gdb) run
Starting program: /home/tony/a.out

Breakpoint 1, main () at junk.c:7
7   int a=3, b=4, c, d, e;
```

After confirming that the program is now running, GDB announces that it reaches the first breakpoint at the `main()` function on line 7 of the source code. Consulting our text editor display, we see line 7 is not the title line of the `main()` function, but rather the first instruction within the `main()` function. Next, GDB shows the line number (7) followed by the actual source code on that line.

At this point in time, GDB has not allowed that line to execute yet, but is merely showing you what it will do when executed.

Next, I issue GDB five commands, telling it to display the variables `a` through `e` in the program. These `display` commands will remain active so long as the program's execution stays within the `main()` function.

```
(gdb) display a
1: a = -1208299532
(gdb) display b
2: b = 134513787
(gdb) display c
3: c = -1207963660
(gdb) display d
4: d = -1208298620
(gdb) display e
5: e = -1209667219
```

After typing in each `display` command, GDB responds by showing me the value of each specified variable. Note how none of these variables have sensible values in them (yet). Variables `a` and `b` will become initialized only *after* line 7 executes, which is has not yet. The other variables require computation by other lines in the program before they are initialized. All those huge numerical values are completely random, resulting from whatever data happened to be in those addresses of the computer's memory at the time the program started.

Now we need to tell GDB what to do after it's stopped at the first breakpoint. The option I apply in this example is `step`, which may be issued in its abbreviated form `s`. The "step" command tells GDB to proceed to the next line of code in the source file and stop again. Here we see three successive "steps" in the program's execution, and at each one we see the five displayed variable values:

```
(gdb) s
8   c = a - b;
5: e = -1209667219
4: d = -1208298620
3: c = -1207963660
2: b = 4
1: a = 3
(gdb) s
9   d = b - a;
5: e = -1209667219
4: d = -1208298620
3: c = -1
2: b = 4
1: a = 3
(gdb) s
11   e = myfunction(a,b);
5: e = -1209667219
4: d = 1
3: c = -1
2: b = 4
1: a = 3
```

Note how one by one these variables become initialized to their proper values following the execution of each instruction listed in the source code. Again, it is important to remember that the source line listed prior to each set of displayed variables *has not yet executed*, but will execute following the next "step" command. This is why, for example, we see `c = -1207963660` listed after the line `8    c = a - b;`, and `c = -1` after the next `s` command (while waiting to execute `9    d = b - a;`).

Single-stepping a program in this manner is the simplest way to use a debugger to examine a program's execution, but it is not the only way. GDB offers options to step a certain number of times with one command, for example, as well as a similar command named `next` (abbreviated `n`) designed to skip past certain lines in a program that are usually of less interest. As this is not a full instruction manual for GDB, the reader is encouraged to reference official documentation for the GDB debugger to learn more about the other options.

Following the next step command, the program execution jumps to `myfunction()`. In the interest of tracking all variable values in this program, I now[24] issue three more "display" commands for the local variables within this function:

```
(gdb) s
myfunction (x=3, y=4) at junk.c:20
20   result = (5 * x) - (2 * y);
(gdb) display x
6: x = 3
(gdb) display y
7: y = 4
(gdb) display result
8: result = 134518564
```

At this point in the program's execution we see `x` and `y` already initialized to the proper values by the function call (passing `a` and `b` to `x` and `y`, respectively), but `result` shows up with a random value because it has not yet been initialized.

Issuing more "step" commands to proceed through `myfunction()`:

```
(gdb) s
22   return result;
8: result = 7
7: y = 4
6: x = 3
(gdb) s
23 }
8: result = 7
7: y = 4
6: x = 3
```

Now we have reached the end of `myfunction()` and are awaiting return to `main()`. We see this happen with the next "step" command:

```
(gdb) s
main () at junk.c:13
13   return 0;
5: e = 7
4: d = 1
3: c = -1
2: b = 4
1: a = 3
```

Finally all the program's variables have their properly-calculated values.

[24] This was not an option at the start-up of GDB, because the variables `x`, `y`, and `result` are local to `myfunction()` and thus unknown to `main()`. GDB only allows you to call for the display of variables it's already aware of in the debugging session.

The next three step commands bring the program to its natural terminus:

```
(gdb) s
14 }
5: e = 7
4: d = 1
3: c = -1
2: b = 4
1: a = 3
(gdb) s
0xb7e455a5 in __libc_start_main () from /lib/libc.so.6
(gdb) s
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 2061) exited normally]
```

3.23 Simple graphics using C

The official C language standard offers no built-in graphics capabilities. Unlike text functions such as `printf()` which are part of C's Standard I/O (`stdio`), C has no intrinsic function(s) for drawing graphical shapes on a computer screen.

C, like all complete programming languages, is *extensible* which means it is possible to extend its capabilities by writing special algorithms using standard C functions and then include those new functions in C code libraries which may be imported into other C-based programming projects. At a fundamental level, any programming language capable of writing data to a computer's memory as well as to peripheral hardware is capable of instructing that computer to do anything it is physically capable of doing, including the generation of graphic images. A graphics library written for this purpose simplifies the task for you, the programmer, by providing convenient functions you can call within your code while the library handles all the detailed and confusing work of writing data to the correct memory addresses and I/O busses. The major problem with graphics programming, though, is that there exists no industry-wide standard for graphics hardware which means at some level any C code written to produce arbitrary graphical images will only work with certain types of computer hardware (and/or certain specific operating system software). What might be an easy-to-use graphics library written for Intel x86 processors running DOS or Windows might be useless for a Motorola-based processor running some other operating system. This problem is known as *portability*, where code written for one machine and/or operating system software will not work on another machine and/or operating system. Portability problems exist for a great many programming tasks, but graphics programming seems especially plagued by issues of non-portability.

Some modern graphics-programming libraries such as *OpenGL* constitute an admirable attempt to make graphics programming *more* standardized, but this cross-platform solution is both complicated and computationally expensive. For the beginning C programmer who just wants to paint a few simple images on a screen, the learning curve imposed by OpenGL is daunting. For the C programmer who wishes to add graphics capability to an embedded computer system with limited processing speed, memory, and video capability, OpenGL may not work at all.

As you no doubt have ascertained by now, this is not a comprehensive tutorial on the C programming language. There is much about the standard, built-in features of C we simply have not and will not explore within these pages. In keeping with this "introductory" theme, you will doubtless be unsurprised by the fact that we will not teach you how to draw complicated graphics using any C libraries, either. However, we will explore a couple of ways you may use C to produce *simple* and *relatively portable* graphics for virtually any size of computer.

The first graphics technique we will introduce is a library known as *ncurses*, developed as a way to vastly improve upon the plain-text capabilities of `printf()`. This library lets you place colored monospaced-type text anywhere on the computer's console, with locations specified by x and y coordinates.

The second graphics technique is more or less cheating, by using existing software applications to render graphics from plain-text output from a C program you write. This way, your C code doesn't actually have to handle any graphics rendering on its own, but merely produces a stream of text describing how some other "back-end" application should. For certain applications such as mathematical visualization (e.g. plotting graphs), this is perhaps the simplest and fastest way to get the job done, and should be learned by any new student of C.

3.23.1 Ncurses

The legacy Unix operating system was created as a multi-user operating system for mainframe computers to serve a great many users logged in through simple "dumb" terminals capable only of transmitting ASCII characters typed at a keyboard and displaying ASCII characters in monospaced typeface on a monochromatic screen. These user terminals communicated with the mainframe computer via serial networks such as RS-232, acting essentially as remote keyboards and monitors for the single mainframe computer. As user interfaces they were well-suited to the simple `printf()` and `scanf()` programs you've explored in this Tutorial, but were incapable of displaying complex graphical images such as polygons or photographs. The display palette of a terminal was divided into *cells* addressed by *row* and *column* numbers, each cell capable of displaying a single alphanumeric character.

Some terminals had capabilities beyond the display of plain ASCII text, including the ability to render text in different colors and to place text in arbitrary cell positions rather than always left-to-right and top-to-bottom. These special features were triggered by sending "escape characters" that are part of the ASCII standard, instructing the terminal how to render subsequent ASCII alphanumeric characters on the screen. A program written to display text on such a terminal could be written to include these escape characters in the data transmitted to the terminal, causing it to display text in more sophisticated ways. However, since terminal capabilities beyond basic ASCII alphanumeric text were not standardized, this meant writing non-portable code for specific models of terminal.

A solution to this portability problem was a code library named *curses*, providing a set of standardized C functions for placing and coloring text at arbitrary locations on a terminal's screen, as well as accepting typed user input. This was included within AT&T's System V version of the Unix operating system, and it greatly simplified the task of writing application programs to produce fancy output on terminal screens. No longer did the application programmer have to master the idiosyncratic specifications of each terminal model, but now only had to learn a few `curses` function calls.

In 1993 a software developer named Thomas Dickey wrote an open-source work-alike version of AT&T's System V Release 4.0 `curses`, calling it *ncurses* (for "new" `curses`). This free version has found its way into most Linux operating system distributions and enjoys continued support to this day (2022).

One might be prompted to wonder, *why consider a primitive character-cell based graphics library now that we live in the 21st century?* The answer to this (good) question is two-fold: (1) `ncurses` is far simpler to learn than most any modern graphics library which is good news for beginning programmers, and (2) `ncurses` requires far less computational power and memory usage than any "modern" graphics programming technique which is strongly advantageous for applications running on embedded systems where processing power, memory, and network bandwidth are often quite limited. For example, an `ncurses` application running on a "host" computer may be displayed remotely by any other computer via an `ssh` connection to that host, using very little network bandwidth. An important engineering lesson resides here: so long as the underlying standards (e.g. ASCII encoding, `terminfo` terminal specifications) continue to be supported, a programming technique is not obsolete. New and sophisticated does not always mean *better*. Sometimes an "old" solution is optimal for reasons such as speed, reliability, economy, etc.

Many tutorials and references exist for the `ncurses` API[25], and so here we will explore only a few of the most important features and functions of `ncurses`.

- Your C program must include the header file `ncurses.h` at or near the beginning of your source file.
- When compiling, you must link the `ncurses` library. For a command-line compiler such as GCC, this means specifying `-lncurses`.
- An `ncurses` session is invoked by calling the `initscr()` function.
- An `ncurses` session is ended by calling the `endwin()` function.
- Use `printw()` to print text to the `ncurses` "window" rather than `printf()`. Otherwise, syntax is the same.
- Position the cursor using the `move()` function, which takes two integer arguments. The first argument specifies the y-axis location (i.e. the *row* number) while the second argument specified the x-axis location (i.e. the *column* number). For example, `move(5,8)` moves the cursor to row 5 and column 8. The coordinate $(0, 0)$ resides at the upper-left corner.
- Two constants within `ncurses` are automatically initialized at start-up which detect the console's dimensions: `LINES` specifies the number of rows (y axis) in the display and `COLS` specifies the number of columns (x axis). This is important when writing `ncurses` applications for a window of unknown size, as you can use these pre-defined constants to set boundaries for how far you may `move()` the cursor!
- The `start_color()` function initiates `ncurses`' ability to place colored text on the display. Following this function call, one must define foreground/background color schemes using the `init_pair()` function which takes three arguments: an identifying integer number, the foreground color, and the background color. For example, `init_pair(1, COLOR_RED, COLOR_YELLOW)` defines color scheme #1 as having red text against a yellow background. Usually you will call `init_pair()` multiple times to define sets of color schemes for later use. To activate any color scheme, use the `attron()` function specifying which numbered color pair you previously set up (e.g. `attron(COLOR_PAIR(1))`) activates color scheme #1. De-activate a currently-active color scheme using `attroff()` (e.g. `attroff(COLOR_PAIR(1))`) de-activates color pair #1 to return to the default color scheme (#0) which is white text on a black background.
- Use `clear()` to clear the `ncurses` display of all text.

To see a working example of an `ncurses` demonstration program, refer to section 2.4 beginning on page 18.

[25] A programming library such as `ncurses` written for the purpose of simplifying the creation of new software applications typically provides special functions and data structures and constants that the applications programmer must be aware of in order to properly use the library. Just as a computer's *user interface* is a collection of software components and conventions intended to make it easier for human beings to use a computer for practical purposes, an *applications programming interface* or *API* is the programming library and its associated conventions intended to make life easier for the person(s) coding a new application.

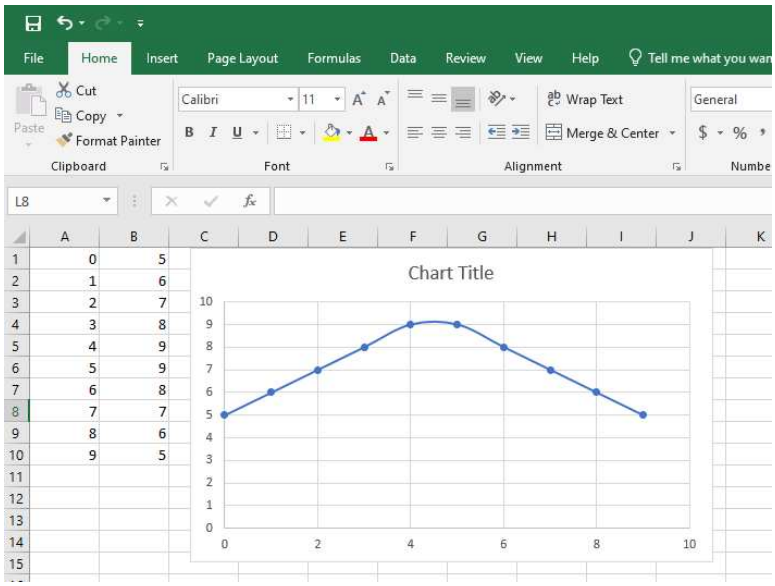
3.23.2 Graphical back-end rendering

CSV data plotting

Many common and robust software applications have the capability of reading strings of plain-text and translating them into graphical images. A good example of this is the *comma-separated variable* (`csv`) format accepted by all computer spreadsheet programs for numerical data. Consider the following `csv` data in plain text:

```
0 , 5
1 , 6
2 , 7
3 , 8
4 , 9
5 , 9
6 , 8
7 , 7
8 , 6
9 , 5
```

When read into any spreadsheet application, this list of ordered number pairs will populate ten rows and two columns within the spreadsheet. Once imported into the spreadsheet application, it is a simple matter to select that data and use it to generate a *scatter plot*, *histogram*, or other visualization to create a two-dimensional object on the computer's screen. Below we see this data set read and displayed by Microsoft Excel:



Software applications other than spreadsheets also exist to convert CSV text data into high-quality graphs. One such example is the open-source program `gnuplot`.

Writing a program in C to create a `csv` list is simple: just use `printf()` instructions, and then "redirect" your program's output to a designated file name rather than display it to the console as standard output. For example, if you compiled your C source code into an executable file named `a.out`, you could run it and direct its output to another file named `data.csv` using the following command-line instruction:

```
./a.out > data.csv
```

Alternatively, you may write your C program in such a way that it directly writes its own data file containing comma-separated text, so that `data.csv` gets created (or overwritten) immediately upon execution of `a.out` without any need for command-line redirection.

For plotting numerical data, writing a program in C using `printf()` statements to output comma-separated variable text, and then importing that plain-text data into an existing visualization tool, is probably the quickest and easiest solution.

HTML for text

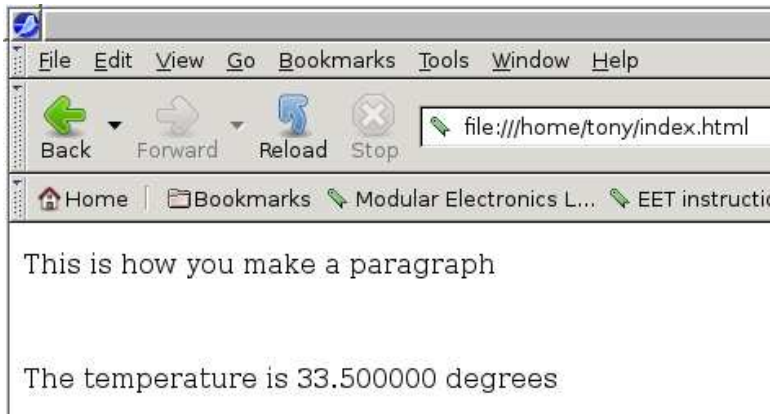
Another back-end technique is to write your C program to output text compatible with some *markup language* such as HTML (HyperText Markup Language), as illustrated here:

```
#include <stdio.h>

int main (void)
{
  float temp = 33.5;
  printf("<html> \n");
  printf("\n");
  printf(" <head> \n");
  printf("     <title>This is my web page</title> \n");
  printf(" </head> \n");
  printf("\n");
  printf(" <body> \n");
  printf("     <p>This is how you make a paragraph</p> \n");
  printf("     <br> \n");
  printf("     <p>The temperature is %f degrees</p> \n", temp);
  printf(" </body> \n");
  printf("\n");
  printf("</html> \n");

  return 0;
}
```

Output:

```
<html>

 <head>
     <title>This is my web page</title>
 </head>

 <body>
     <p>This is how you make a paragraph</p>
     <br>
     <p>The temperature is 33.500000 degrees</p>
 </body>

</html>
```

When this program is run and its output redirected to a file named `index.html`, any web browser application will be able to read that file and display the following page:



Of course, this is a trivial example. A more practical application might be a C program that actually reads input from a temperature sensor to update the value of its floating-point variable `temp`, and then loops continually to write and over-write HTML code to the `index.html` file using file-access functions rather than command-line redirection. Either way, the point here is to show how you may leverage the plain-text nature of a markup language such as HTML to give your C programs a user interface far more sophisticated than the simple console.

Inline SVG for graphic images

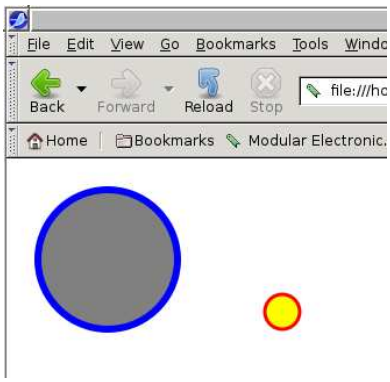
While HTML is a markup language originally designed to place text within a web browser application's display, modern web browsers also understand *Scalable Vector Graphics* (*SVG*) code embedded in HTML source files. Like HTML, SVG is its own text-based markup language, just designed to represent graphical objects rather than text. SVG tags specify coordinate points, colors, linewidths, and other features necessary to draw graphical images on a web page. Unlike raster (bitmap) images where each pixel must be specified, SVG is a *vector* image standard which means the markup text specifies geometric locations and sizes, letting the web browser application draw those images using full available screen resolution.

SVG follows the Extensible Markup Language (XML) standard, using text characters and conventions very similar to HTML. An example of a SVG tags embedded within an HTML source file (called *inline* SVG) is shown here:

Inline SVG source code to draw two circles

```
<html>
<body>

<svg width="200" height="150">
  <circle cx="50" cy="50" r="40" stroke="blue" stroke-width="4" fill="grey"/>
  <circle cx="150" cy="80" r="10" stroke="red" stroke-width="2" fill="yellow"/>
</svg>

</body>
</html>
```

The result when viewed on a web browser looks like this:



Since SVG images are vector and not raster, they may be *scaled* by magnifying or de-magnifying the web browser's view to any desired degree without becoming "pixelated" (i.e. edges and curves rendered as jagged steps).

A C program suitable for writing this inline-SVG code using `printf()` instructions appears below. Note the use of ¨ wherever `printf()` must print a literal double-quotation mark symbol (the "backslash" character acting as an *escape* to tell `printf()` that the very next character is to be treated literally and not in its normal functional sense):

C code for printing inline SVG code

```
#include <stdio.h>

int main (void)
{
  printf("<html> \n");
  printf("<body> \n");

  printf("<svg width=\"200\" height=\"150\"> \n");

  printf("  <circle cx=\"50\" cy=\"50\" r=\"40\" stroke=\"blue\" \
  stroke-width=\"4\" fill=\"grey\"/> \n");

  printf("  <circle cx=\"150\" cy=\"80\" r=\"10\" stroke=\"red\" \
  stroke-width=\"2\" fill=\"yellow\"/> \n");

  printf("</svg> \n");

  printf("</body> \n");
  printf("</html> \n");

  return 0;
}
```

Single-backslash symbols also appear at the ends of each `printf()` statement's first line in order to declare that the text continues on the next line of code. Normally C ignores whitespace which means we may break long lines of code without any ill effect, but here since all whitespace between the functional quotation-marks of each `printf()` statement matters, the single-backslash characters are necessary.

Again, this is a trivial example showing how C may be used to generate inline-SVG source code, which in turn may be read and displayed by web browser software. In a real application we would likely have the C program write this SVG code directly to a file, and also have routines within the C program to update parameters such as object dimensions and color based on variable states. For example, if we were writing a C program for a weather station monitoring outdoor temperature, we might have the sampled value for temperature dictate the color and/or radius of a circle (representing the Sun).

Also, it should be realized that SVG is capable of drawing much more than colored circles! The

reader is encouraged to find a technical reference document on SVG markup in order to learn all the interesting graphics capability of this text-based markup standard. In addition to circles, SVG supports polygons, lines, polylines, text embedded within images, and arbitrary drawing paths to name a few.

Chapter 4

Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

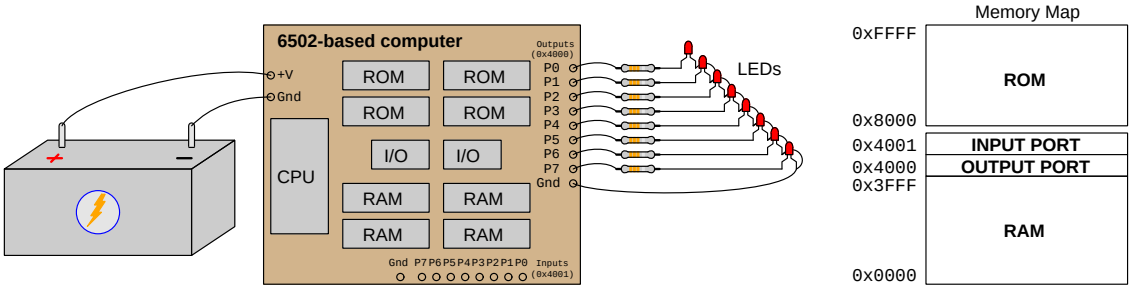
4.1 Introduction to assembly language programming

Microprocessors only understand *machine code* – instructions and data encoded as binary values, often written in hexadecimal "shorthand" form for better human-readability – but even in hexadecimal these codes are non-intuitive and confusing for human programmers to manage. *Assembly code* improves upon this situation by representing each machine-code instruction as an word-abbreviation called a *mnemonic*. The convenience of mnemonics, though, comes at price: since the mnemonics themselves are nonsense to the microprocessor, a special software package called an *assembler* must be used to translate these mnemonics into machine code that the microprocessor can understand. Just as microprocessors each have their own unique instruction set defining which codes perform which functions, assembler software has its own unique "vocabulary" and "grammar" one must abide by in order to write functioning programs. Assembly programming, therefore, is its own *language* and like all languages has specific rules.

Furthermore, assembly-language lacks the strict standardization found in higher-level programming languages such as C, C++, Java, and Python. Just as machine-code programming is specific to the model of microprocessor, assembly-language programming rules are often specific to the assembler software version, which in turn is often closely coupled to the microprocessor model. It is thus impossible to write a generic tutorial on assembly programming, just as it is impossible to write a generic tutorial on machine-code programming. What we will explore here are features of assembly code common to *most* assemblers.

We will use a specific hardware application as the foundation for this lesson on assembly language programming, in order to have a practical context for understanding what the program does and how it works. This means the examples given here will not work with any microprocessor other than the system described here, but that is okay. Many of the principles learned here find general application to other systems.

Our hypothetical computer is shown below, based on a Motorola model 6502 8-bit microprocessor. A single output port mapped to memory address `0x4000` provides the means for our program to turn LEDs on and off, by writing bit-states to that byte located at `0x4000`. Another port at address `0x0401` provides inputs, where our program may read bit-states of the byte stored there to detect logic signals applied to those pins by external circuitry (not shown). ROM begins at address `0x8000` and extends through address `0xFFFF`. RAM begins at address `0x0000` and extends through address `0x3FFF`:



The 6502 provides an Accumulator register plus two general-purpose registers (named X and Y) for temporary data storage, each one eight bits wide.

4.1.1 Machine code to blink an LED

Suppose we wish to make the LED connected to output pin P0 on the computer blink on and off. This means writing a 1 and then a 0 to bit 0 of the byte located at `0x4000` while clearing (making zero) all the other bits at that address. Our program will execute the following steps:

1. Write the binary value `0b00000001` to address `0x4000`
2. Toggle the least-significant bit of that byte in address `0x4000`
3. Go back to second instruction and repeat indefinitely

The same steps, using instructions available in the 6502's instruction set:

1. **Load** the binary value `0b00000001` into the Accumulator register
2. **Store** the Accumulator's value to address `0x4000`
3. Apply the **Exclusive-OR** function to the LSB in the Accumulator using the *mask*[1] `0b00000001`.
4. **Jump** to the second instruction and repeat indefinitely

Researching opcodes and operand formats for the model 6502 microprocessor, we find the following:

- The opcode for "Load Accumulator with immediate value" is `0xA9` followed by the desired value to load (`0x01`)
- The opcode for "Store Accumulator value to absolute memory address" is `0x8D` followed by two bytes specifying the destination address in little-endian order (low byte first, high byte last: `00 40` for address `0x4000`)
- The opcode for "Exclusive-OR with immediate value" is `0x49` followed by the mask value (`0x01`)
- The opcode for "Unconditional Jump to absolute memory address" is `0x4C` followed by the target address in little-endian order

[1] In programming, a *mask* is a set of bits intended to apply a bit-wise logical operation to bits within a larger word of data. Here, our mask of `0b00000001` means the LSB will be XOR'd with 1 (i.e. toggled, to make it switch states from whatever it was before to the opposite of that) while all other bits within the Accumulator's 8-bit word will be XOR'd with 0 (i.e. left alone).

If we write all these opcodes and operands in order, one line per complete instruction, we get the following *hand-assembled* machine code:

```
A9 01
8D 00 40
49 01
4C ?? ??
```

The question-marks are there in our code because we need to determine where our program will begin in the computer's ROM memory space in order to know which address we need to "jump" to in the last instruction. Let's assume our program starts at the very first address in ROM (`0x8000`) and re-write the program showing the starting address of each line[2]:

```
8000 A9 01
8002 8D 00 40
8005 49 01
8007 4C ?? ??
```

Recall that the purpose of our "Jump" instruction (last line, `0x4C`) was to go back to the "Store Accumulator" instruction (`0x8D`) so that our recently XOR'd data will be re-written to the output port at address `0x4000`. Therefore, the address we need to jump to is `0x8002`. Knowing this, we may edit our machine code listing to include this address in the last instruction, in "little-endian" byte order because the model 6502 microprocessor happens to be a little-endian machine:

```
8000 A9 01
8002 8D 00 40
8005 49 01
8007 4C 02 80
```

If we were to write this program to the computer's ROM and then read it back to display in conventional "hex dump" format, it would look like this:

```
8000 A9 01 8D 00 40 49 01 4C 02 80
```

If the ROM IC(s) in our computer are socketed and therefore easily removed for programming by an external device, we could use a PROM programmer[3] to write this short program into a programmable ROM memory chip and then re-insert it into our computer's board to run.

[2] This is beginning to resemble the common "hex dump" memory display format, except that each line of text is limited to just one instruction

[3] Commercially-available PROM programming tools consist of a unit connected to a personal computer with a

4.1.2 Assembly code to blink an LED

Now that we have seen the "hand-assembly" method of creating a simple LED-blinking program for our 6502-based computer, let us explore how we could do the same using *assembly language*. Starting with the original program specification telling us what we need the computer to do:

1. Write the binary value `0b00000001` to address `0x4000`
2. Toggle the least-significant bit of that byte in address `0x4000`
3. Go back to second instruction and repeat indefinitely

Next, we refer to instructions available in the 6502's instruction set, but this time we write the *mnemonic* abbreviations given in that instruction set instead of opcodes:

1. `LDA` value `0b00000001`
2. `STA` to `0x4000`
3. `EOR` mask `0b00000001`
4. `JMP` to second instruction

We are almost done with our assembly-language program! All we need to do now is write it using the proper syntax[4] expected by our assembler software, being sure to include a *directive*[5] to the assembler to begin at address `0x8000`.

```
      .ORG  $8000
      LDA   #$01
LOOP
      STA   $4000
      EOR   #$01
      JMP   LOOP
```

This program you see above is completely functioning, ready to be assembled into machine code and written to the computer's ROM.

zero-insertion-force (ZIF) IC socket to facilitate easy plugging and unplugging of memory ICs. Software provided with the programmer allow you to type the hex dump data into an editor window (or into a plain-ASCII text file) and then have that data written to the memory IC with the click of a button or a command-line instruction typed into the personal computer. If you *really* desire a low-level learning experience, you can program the PROM chip by connecting address, data, and write-enable lines to toggle switches, then toggling those switches to specify addresses, data to be written to those addresses, and pressing the write-enable switch to "burn" that data into the chip when you are ready. Needless to say, the latter option is the most tedious.

[4] We are assuming here that our assembler does not understand binary notation and expects all numerical values to be expressed in hexadecimal instead.

[5] A "directive" is an instruction given not to the microprocessor, but rather to the assembler software. It tells the assembler to translate the assembly "source" code into machine code in some particular manner.

Optional *comments* help make our code easier to read:

```
      .ORG  $8000   ; Begin at address 0x8000
      LDA   #$01    ; Load 0x01 into Accumulator
LOOP                ; Mark the beginning of the loop
      STA   $4000   ; Store Accumulator value to Output port
      EOR   #$01    ; XOR the least-significant bit
      JMP   LOOP    ; Jump to the beginning of the loop
```

Take note of some important details of this assembly-language program:

- Order of execution is left-to-right, top-to-bottom in the same order as reading English-language text.
- Any left-justified text (e.g. `LOOP`) is considered a *label* rather than an *instruction*, whether on its own line or preceding an instruction.
- Instructions (e.g. `LDA`, `STA`, etc.) must be preceded by whitespace. The number of space or tab characters doesn't matter.
- Operands to those instructions (e.g. `$4000`, etc.) must be separated to the right of those instructions by some whitespace as well. The number of space or tab characters doesn't matter. In this listing I've adjusted the number of spaces to make the columns neatly align.
- Any text to the right of a semicolon (`;`) character is considered a comment, ignored by the assembler and not included in the machine code at all.
- Any instruction preceded by a dot (e.g. `.ORG`) is a *directive* to the assembler, telling how to do some aspect of the translation into machine code, but not appearing in the final machine code itself.
- Note how the Jump instruction's target address was resolved by the assembler, with no need for us to count address numbers to figure out where it should jump to. We simply place a label and let the assembler figure out those details for us.
- This assembler uses a dollar-symbol (`$`) to denote any hexadecimal value.
- The pound symbol (`#`) denotes an *immediate* value, meaning the literal number value specified. Otherwise, the operand value is considered to be a memory address location.

It should be noted that some of these conventions vary from assembler to assembler. For example, some assemblers require all labels to contain a colon at the end (e.g. `LOOP:` rather than `LOOP`). Some assemblers allow C-style hexadecimal notation (e.g. `0x4000`) while others insist on the `$` character. Some assemblers allow binary notation (e.g. `0b00000001` or `%00000001`) while others don't. Some assemblers require directives be preceded by a dot (e.g. `.ORG`) while others insist directives *not* be preceded by any character. As always when using software, refer to the manufacturer's documentation for details!

4.1.3 Slowing down the blinking

If we were to actually assemble and write this program to ROM, then start the computer to initiate program execution, we would likely find the LED blinking on and off so fast that it appeared to be steadily lit (albeit dimmer than usual). The reason for this is the fast fetch/execute cycle time of a typical microprocessor. A model 6502 running at a clock speed of 1 MHz would blink the LED on and off at a rate far too quick for the human eye to see[6].

In order to make the blinking rate slow enough to see, we must somehow *delay* the loop's repetition. One easy way to do this is to insert a "counting" loop inside of our program's blinking loop. This counting loop keeps the microprocessor occupied by doing nothing but sequentially counting, in order to purposely waste time and thereby delay its toggling of the LED output bit.

Here is a section of assembly code using common 6502 instructions to perform this delay task by forcing the microprocessor to count backwards from 255 (`0xFF`) until it reaches zero, complete with explanatory comments:

```
      LDX #$FF          ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX #$00          ; Compare that value to zero
      BNE DELAY_LOOP    ; If unequal, "branch" to DELAY_LOOP to repeat
```

One way to incorporate this delay code into our program is to simply insert it "in-line" with the original code between the `EOR` and `JMP` instructions, like this:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      EOR   #$01        ; XOR the least-significant bit
      LDX   #$FF        ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX   #$00        ; Compare that value to zero
      BNE   DELAY_LOOP  ; If unequal, "branch" to DELAY_LOOP to repeat
      JMP   LOOP        ; Jump to the beginning of the loop
```

[6] According the the model 6502 manual,, the `STA` instruction requires 4 clock cycles, the `EOR` instruction 2 clock cycles, and the `JMP` instruction 3 clock cycles. This means 9 clock cycles would be required for every pass through the program, with two passes required for a full cycle of the LED's blink (i.e. on *and* off). Dividing 1 MHz by the 18 clock cycles necessary to fully cycle the LED gives an LED blinking frequency of 55.556 kHz!

While this solution works quite well, there is a more sophisticated way to achieve the same "loop within a loop" structure, and that is to place the delay-time code within its own *subroutine*. A "subroutine" is a section of code that stands apart from the rest, ready to be *called* by the main portion of the program whenever needed.

Subroutines are particularly useful when that code must be invoked at multiple points within the main program. Instead of copying-and-pasting the necessary code repeatedly in-line where needed, we simply insert a "call" or "jump to subroutine" instruction where it's needed and the microprocessor will jump to that new address (its Program Counter being preset as needed). Then, at the end of the subroutine we place a "return" instruction that tells the microprocessor to resume where it left off in the main program.

The following shows a listing of the assembly code for the slow-blinking program using a subroutine called `DELAY`[7]:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      EOR   #$01        ; XOR the least-significant bit
      JSR   DELAY       ; Call the DELAY subroutine
      JMP   LOOP        ; Jump to the beginning of the loop

DELAY
      LDX   #$FF        ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX   #$00        ; Compare that value to zero
      BNE   DELAY_LOOP  ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS               ; Return to the main program
```

Admittedly the use of subroutines doesn't appear to be any better than simply inserting the `DELAY` code in-line with the original program. However, if we had a need to call this subroutine multiple times within our program, all we would need to add is another `JSR DELAY` instruction. Thus, the subroutine strategy becomes more efficient than the in-line strategy proportional to how many times that routine must execute.

[7] The blank line between `JMP LOOP` and `DELAY` is there for esthetic purposes only, to help our eyes see the distinction between the main program and the subroutine. We could eliminate this blank line (or add more!) and the program would execute just as well.

An useful tool provided by most assemblers is a *disassembly* option. This takes the assembled machine code and translates it "backwards" into assembly code, displaying the memory addresses, machine code hex dump, and equivalent assembly side-by-side for comparison:

```
Address    Hexdump    Disassembly
---------------------------------
$8000      A9 01      LDA #$01
$8002      8D 00 40   STA $4000
$8005      49 01      EOR #$01
$8007      20 0D 00   JSR $000D
$800A      4C 02 80   JMP $8002
$800D      A2 FF      LDX #$FF
$800F      CA         DEX
$8010      E0 00      CPX #$00
$8012      D0 FB      BNE $800F
$8014      60         RTS
```

Note how the "jump" and "branch" instructions all specify address locations in one way or another, but not the "return instruction" at the end of the subroutine. How does the microprocessor know which memory address to return to after completing the subroutine? The answer lies in a feature of the microprocessor called the *stack*: a section of volatile memory used by the processor to remember such things as previous Program Counter values. A jump-to-subroutine instruction causes the current memory address value held in the Program Counter to be "pushed" onto the stack before jumping to the subroutine's starting memory address. A "return" instruction causes the microprocessor to "pop" the former address value off the stack and into the Program Counter again, so that execution resumes right where it left off[8]. The model 6502 processor uses a portion of its RAM memory space for its stack (`0x0100` through `0x01FF`).

If you examine this disassembled code closely, you will notice something strange with the "branch-if-not-equal" instruction: the disassembled code says `BNE $800F` but the machine code does not actually contain the `0x800F` address. Instead, it only contains the opcode for `BNE` (`D0`) and a byte with a value of `0xFB`. This is an example of *relative addressing*, where the operand to the instruction declares not the address itself, but rather *how many addresses to skip, either forward or backward*. As an eight-bit signed number, `0xFB` is equal to negative five. This tells the microprocessor to decrement its Program Counter by five to repeat the `DELAY_LOOP`. If you count from address `0x8013` where the `0xFB` operand was resides to `0x800F` where the delay loop begins, you count five addresses (inclusive). Relative addressing is more efficient than absolute addressing because we only need one byte telling the instruction how far to jump instead of two bytes to specify a 16-bit address. Interestingly, the disassembler opted to show us an absolute address even though that's not really how the 6502's `BNE` instruction works.

[8]Stacks are analogous to a pile (stack) of paper notes. If a person is reading a book and they suddenly get told to turn to a different chapter to read a passage there, they may write the current page number on a note and "push" that note to the top of the stack so they won't forget it while turning to the new passage. After reading the new passage, the person retrieves their note from the top of the stack (i.e. "popping" it off the stack) and references it to return to the page where they left off. This may occur more than once, and the stack will "remember" not only the page numbers but also keep everything in the right order as the person eventually returns to their original place in the book.

4.1.4 Simplifying with symbols

Another technique useful for making our assembly-language programs easier to read and to maintain is the use of *symbols* to represent numerical values. Consider the last version of our LED-blinking program re-written to incorporate three symbols, defined at the very beginning of the code listing:

```
DTIME .EQU  $FF          ; Create a symbol "DTIME" for the delay time parameter
OUTPT .EQU  $4000        ; Create a symbol "OUTPT" for the Output port address
LED   .EQU  $01          ; Create a symbol "LED" for the LED's bit number

      .ORG  $8000        ; Begin at address 0x8000
      LDA   #LED         ; Load 0x01 into Accumulator
LOOP                     ; Mark the beginning of the loop
      STA   OUTPT        ; Store Accumulator value to Output port
      EOR   #LED         ; XOR the least-significant bit
      JSR   DELAY        ; Call the DELAY subroutine
      JMP   LOOP         ; Jump to the beginning of the loop

DELAY
      LDX   #DTIME       ; Load value 0xFF into register X
DELAY_LOOP
      DEX                ; Decrement (subtract 1 from) value stored in X
      CPX   #$00         ; Compare that value to zero
      BNE   DELAY_LOOP   ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS                ; Return to the main program
```

The `.EQU` directive tells the assembler to treat the symbol (on the left) as an alias of the value (on the right). This lets us use lettered symbols within our code rather than numerical values for important parameters such as I/O addresses, delay time, etc. These symbols may be used as many times as desired, and they will always mean the same thing (e.g. the `LED` symbol is used twice in this program, and it means `$01` both times).

4.1.5 Using the stack

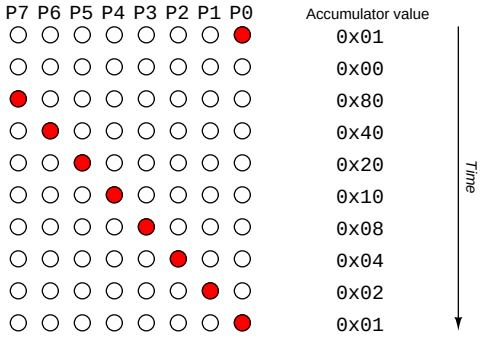
Previously we mentioned the microprocessor's *stack*, a section of RAM used to hold data in sequential order. Stacks may be thought of as a *Last-In First-Out* shift register, where data retrieved from the stack is in reverse order of how data is placed onto the stack. The analogy of a microprocessor's stack being a literal stack of paper sheets is helpful here: if we pull papers from the top of the stack, we will find their sequence is in reverse order of how we placed those sheets on the stack.

Microprocessors use their stack to manage subroutine calls, "pushing" the last Program Counter memory address to the stack prior to jumping to the subroutine's address, then "popping" that old address off the stack when the subroutine completes so it knows where to resume its previous place in the main program. This form of stack usage is automatic, being built-in to the finite state machine sequence as part of each subroutine "call" instruction and each subroutine "return" instruction. *Interrupts* also use a stack to remember where to jump back in the main program after completing the interrupt service routine (ISR).

Certain instructions exist to make use of the stack in ways that are not necessarily related to subroutines or interrupts, and these can be very useful. Here we will explore one such practical use of the stack. Consider this simple "chasing LED" program using the 6502's "Rotate Right" instruction to shift the place of a single "1" bit in the Accumulator byte, the goal being to create a "chasing" LED display on our computer where the light appears to repeatedly sweep along the row of LEDs:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      ROR               ; Rotate Accumulator bits one place right
      JMP   LOOP        ; Jump to the beginning of the loop
```

This code is every bit as simple as our original LED-blinking program. When run, it produces the following pattern of light (sequence shown chronologically from top to bottom):



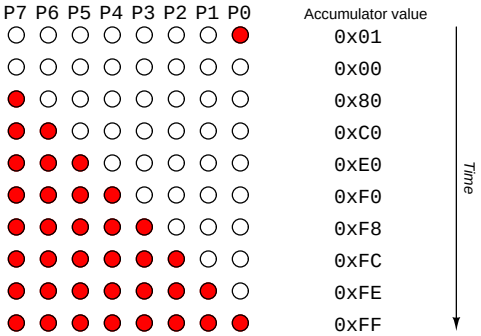
During the step where no LEDs are lit, the "1" bit resides in the *Carry* bit of the microprocessor's Status register, a special register used to store the results of certain mathematical and logical operations.

This pattern of light is what we expect the `ROR` instruction to produce after the Accumulator is initially loaded with `0x01`. All is well, except for the same problem we had with our original "blinking LED" program: the sequence runs too fast for our eyes to discern. It just looks like a blur of eight LEDs all (dimly) lit!

We already know how to slow programs down, by inserting a counting loop that "wastes" the microprocessor's time, so let's modify this program accordingly:

```
      .ORG  $8000        ; Begin at address 0x8000
      LDA   #$01         ; Load 0x01 into Accumulator
LOOP                     ; Mark the beginning of the loop
      STA   $4000        ; Store Accumulator value to Output port
      ROR                ; Rotate Accumulator bits one place right
      JSR   DELAY        ; Call the DELAY subroutine
      JMP   LOOP         ; Jump to the beginning of the loop

DELAY
      LDX   #DTIME       ; Load value 0xFF into register X
DELAY_LOOP
      DEX                ; Decrement (subtract 1 from) value stored in X
      CPX   #$00         ; Compare that value to zero
      BNE   DELAY_LOOP   ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS                ; Return to the main program
```

However, when we run this program we get a different light sequence:



For some reason, the "0" states rotated off the LSB-end of the byte are becoming "1" states to fill the MSB. Recall that the `ROR` instruction draws from the *Carry* bit of the Status register to fill

the MSB at each iteration. A reasonable hypothesis is that something other than the ROR is setting the Carry bit.

Indeed something *does* act to set the Carry bit when we don't want it to: the "compare X" CPX instruction inside our *DELAY* subroutine. According to the 6502 instruction set manual, the CPX sets the Carry bit if ever the X register's value is equal to or greater than the value it's being compared against. In fact, this is how the BNE instruction knows when to branch: it checks the Status register which is updated by all mathematical, logical, and comparison instructions. Given the design of our time-delay subroutine where the X register begins at a large value and counts down toward the comparison value of zero, we are *guaranteed* to return from that subroutine with the Carry bit set.

This causes problems for our ROR instruction, which takes the "1" value left in the Carry bit from the subroutine's CPX instruction and adds it to our chasing light sequence, which we do not want. Somehow we need the ROR to act on the Carry bit *it* generated, not the *new* Carry bit generated by the subroutine's CPX instruction.

Our stack ends up being a simple solution to this problem. All we need to do is "push" the Status register's state to the stack prior to calling the subroutine, then "pop" that old data back off the stack and into the Status register again before the ROR instruction reads the Carry bit. In other words, we can use the stack as temporary storage for the Status bits, and recall those bits after the subroutine is done using the Status register for its own purposes.

All this requires is the addition of two new instructions surrounding the "jump to subroutine": a PHP instruction ("push processor status on stack") prior to the jump, and PLP instruction ("pull processor status from stack") after the jump:

```
      .ORG  $8000       ; Begin at address 0x8000
      LDA   #$01        ; Load 0x01 into Accumulator
LOOP                    ; Mark the beginning of the loop
      STA   $4000       ; Store Accumulator value to Output port
      ROR               ; Rotate Accumulator bits one place right
      PHP               ; Push Status register to stack
      JSR   DELAY       ; Call the DELAY subroutine
      PLP               ; Pop Status register off stack
      JMP   LOOP        ; Jump to the beginning of the loop

DELAY
      LDX   #DTIME      ; Load value 0xFF into register X
DELAY_LOOP
      DEX               ; Decrement (subtract 1 from) value stored in X
      CPX   #$00        ; Compare that value to zero
      BNE   DELAY_LOOP  ; If unequal, "branch" to DELAY_LOOP and repeat
      RTS               ; Return to the main program
```

The stack is a very useful feature of any microprocessor, but it does have its limitations. If we push far more data onto the stack than we pop off, we can get a *stack overflow* where the oldest

data gets overwritten and is lost. Also, we need to be very careful that we push from the correct sources and pop to the correct destinations. For example, in the above program we pushed the Status register to the stack, then immediately after that the `JSR` instruction pushed the Program Counter value to the stack before going to the subroutine. When the subroutine completed, it popped the old Program Counter value off the stack, and then immediately after that we popped the old Status register off the stack. This works because those sources and destinations came in the correct sequence, and so the data going on and off the stack went where it should.

4.2 ASCII character codes

ASCII characters consist of seven-bit digital words. The following table shows all 128 possible combinations of these seven bits, from 0000000 (ASCII "NUL" character) to 1111111 (ASCII "DEL" character). For ease of organization, this table's columns represent the most-significant three bits of the seven-bit word, while the table's rows represent the least-significant four bits. For example, the capital letter "C" would be encoded as 1000011 in the ASCII standard.

↓ LSB / MSB →	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	˜
1111	SI	US	/	?	O	_	o	DEL

It is worth noting that the ASCII codes for the Arabic numerals 0 through 9 are simply the four-bit binary representation of those numbers preceded by 011. For example, the number six (0110) is represented in ASCII as 0110110; the number three (0011) in ASCII as 0110011; etc. This is useful to know, for example, if you need to program a computer to convert single decimal digits to their corresponding ASCII codes: just take each four-bit numerical value and add forty-eight (0x30 in hexadecimal) to it.

4.3 GCC quick reference

Here are some common options used with the GNU C Compiler, shown here with the example source file `hello.c`:

`gcc hello.c` – *this is the basic instruction for preprocessing/compiling/assembling/linking*

`gcc -E hello.c` – *stop shy of compiling the files, dumping results to the console*

`gcc -S hello.c` – *stop shy of assembling the files, leaving an assembly file named* `hello.s`

`gcc -c hello.c` – *stop shy of linking the files, leaving an object file named* `hello.o`

`gcc -g hello.c` – *include debugging data within the compiled code, that will work with the* `GDB` *debugger application*

`gcc -o hello.exe hello.c` – *renames executable file as* `hello.exe`

`gcc -lm hello.c` – *links the C math library with your code after compilation*

`gcc file1.c file2.c file3.c` – *compiles multiple source-code files into one executable*

`gcc -Wall hello.c` – *shows all warnings*

Chapter 5

Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read[1] the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture[2], the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is *independence*, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of *experimentation* and *computer simulation* (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of *metacognition* within these statements: they are designed to foster a regular habit of examining one's own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

[1] Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book textitReading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

[2] Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.

General challenges following tutorial reading

- Summarize as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.
- Simplify a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.
- Where did the text make the most sense to you? What was it about the text's presentation that made it clear?
- Identify where it might be easy for someone to misunderstand the text, and explain why you think it could be confusing.
- Identify any new concept(s) presented in the text, and explain in your own words.
- Identify any familiar concept(s) such as physical laws or principles applied or referenced in the text.
- Devise a proof of concept experiment demonstrating an important principle, physical law, or technical innovation represented in the text.
- Devise an experiment to disprove a plausible misconception.
- Did the text reveal any misconceptions you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.
- Describe any useful problem-solving strategies applied in the text.
- Devise a question of your own to challenge a reader's comprehension of the text.

General follow-up challenges for assigned problems

- Identify where any fundamental laws or principles apply to the solution of this problem, especially before applying any mathematical techniques.
- Devise a thought experiment to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.
- Describe in detail your own strategy for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?
- Is there more than one way to solve this problem? Which method seems best to you?
- Show the work you did in solving this problem, even if the solution is incomplete or incorrect.
- What would you say was the most challenging part of this problem, and why was it so?
- Was any important information missing from the problem which you had to research or recall?
- Was there any extraneous information presented within this problem? If so, what was it and why did it not matter?
- Examine someone else's solution to identify where they applied fundamental laws or principles.
- Simplify the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a limiting case (i.e. altering a variable to some extreme or ultimate value).
- For quantitative problems, identify the real-world meaning of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.
- For quantitative problems, try approaching it qualitatively instead, thinking in terms of "increase" and "decrease" rather than definite values.
- For qualitative problems, try approaching it quantitatively instead, proposing simple numerical values for the variables.
- Were there any assumptions you made while solving this problem? Would your solution change if one of those assumptions were altered?
- Identify where it would be easy for someone to go astray in attempting to solve this problem.
- Formulate your own problem based on what you learned solving this one.

General follow-up challenges for experiments or projects

- In what way(s) was this experiment or project easy to complete?
- Identify some of the challenges you faced in completing this experiment or project.

- Show how thorough documentation assisted in the completion of this experiment or project.
- Which fundamental laws or principles are key to this system's function?
- Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.
- What will happen if (component X) fails (open/shorted/etc.)?
- What would have to occur to make this system unsafe?

5.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking[3]. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor's task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student's needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your *reasoning* than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that *you* are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

[3] *Analytical* thinking involves the "disassembly" of an idea into its constituent parts, analogous to dissection. *Synthetic* thinking involves the "assembly" of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.

5.1.1 Reading outline and reflections

"Reading maketh a full man; conference a ready man; and writing an exact man" – Francis Bacon

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, journal their own reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

☑ Briefly SUMMARIZE THE TEXT in the form of a journal entry documenting your learning as you progress through the course of study. Share this summary in dialogue with your classmates and instructor. Journaling is an excellent self-test of thorough reading because you cannot clearly express what you have not read or did not comprehend.

☑ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

☑ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

☑ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

☑ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

☑ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

5.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module's full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

Thought experiments as a problem-solving strategy

Interpreting intermediate results as a problem-solving strategy

Source code

5.1.3 Writing your first C program

Write your own program in the C programming language, compile it, run it, and document the results (i.e. the text it displays on your computer's console). Then, answer the following questions:

- Explain what each line of your program does
- Identify the order in which each line of code gets executed when your program runs
- Explain what *compilation* means, and why it is necessary when programming in C
- Explain the difference between *source code* and *executable code*

You have multiple options for compiling C code on your computer, listed here in order from least difficult to most difficult:

1. Use a cloud-based editor/compiler such as `OnlineGDB`, requiring nothing but web browser software and an internet connection.
2. Install a commercial IDE (Integrated Development Environment) such as Microsoft's Visual Studio to your computer and use it for editing source code and for compiling executable code.
3. If you are using Microsoft Windows as your operating system, install Windows Subsystem for Linux (*WSL*) and a Linux distribution along with `GCC` and its associated libraries which will provide you with a Linux operating system console within a virtual machine. If you are using a Unix-based operating system (e.g. Android, Apple OS X, ChromeOS, Linux, etc.), simply start up a console and install `GCC` (if necessary), using whatever text-editor software is already installed to write your C source code.

Challenges

- Modify your program, experimenting with using its instructions in different ways. What are you able to determine about the usage of C instructions by these experiments?
- Intentionally (or unintentionally) place errors in your program, seeing how the compiler generates warning or error messages helping you to identify those mistakes.

5.1.4 Re-writing a "Hello world!" program

Edit the following program to use multiple `printf` instructions, each one of those instructions printing just a *single character* to the computer's console while still printing "Hello world!" as a complete sentence:

```
#include <stdio.h>

int main (void)
{
  printf("Hello world!\n");

  return 0;
}
```

Also, insert at least one comment in your code explaining how it functions.

Challenges

- Edit your code so that it generates a blank line both before and after the line reading "Hello world!"

5.1.5 Writing a power calculation program

Write your own program in the C programming language accepting voltage and current values from the user and displaying the corresponding power value, for any DC circuit component. Be prepared to explain how each line of your program works!

Some helpful advice when writing a new program is to write the first version as simply as possible, with no unnecessary features. Only after proving that the code performs its most basic function(s) should you then invest time making the program user-friendly and professional in appearance. It is also helpful, especially when first learning to code, to begin by copying and pasting code from some elementary program you know functions, then modifying that program as opposed to writing one from scratch.

For example, in this case I would recommend starting with a simple "Hello World" example program and modify it to accept two numerical values from the user (voltage and current) and then compute and display the calculated power value. Only after you've got this simple version running well should you add other features such as prompting to instruct the user what to enter.

Challenges

- A good educational exercise in a classroom environment is to have students identify errors in each other's programs. The instructor can solicit examples from students (received by email) and then display them for the entire class to examine in an anonymous format where no one knows who wrote the errant code.
- Another good educational exercise in a classroom environment is to have students streamline each other's programs, condensing them into simpler forms, re-organizing for better readability, etc. The instructor can solicit examples from students (received by email) and then display them for the entire class to examine in an anonymous format where no one knows who wrote the original code.

5.1.6 Resonant frequency calculator program

Examine the following program, intended to compute the resonant frequency for a simple inductor-capacitor (LC) network:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float L, C, f;

  printf("Enter inductor value in milliHenrys: ");
  scanf("%f", &L);

  L = L / 1000.0;  // Converts mH into H

  printf("Enter capacitor value in microFarads: ");
  scanf("%f", &C);

  C = C / 1e6;  // Converts uF into F

  f = 1 / (2 * M_PI * sqrt(L * C));

  printf("Resonant frequency = ");

  if (f < 1e3)
    printf("%f Hz\n", f);

  else if (f >= 1e3 && f < 1e6)
    printf("%f kHz\n", f/1000.0);

  else if (f >= 1e6 && f < 1e9)
    printf("%f MHz\n", f/1000000.0);

  else
    printf("%e Hz\n", f);

  return 0;
}
```

Next, trace the flow of execution (i.e. identify which instructions get executed in which order) for the following user entries:

- $L = 100$ milliHenrys ; $C = 47$ microFarads
- $L = 15$ milliHenrys ; $C = 0.33$ microFarads
- *Make up your own user-entered values resulting in a different flow of execution than either of the above examples*

Next, answer the following questions:

- Describe what *resonance* is.
- Identify some practical applications of resonance in AC circuits.

Challenges

- Modify this program so that it repeatedly prompts the user for new C and L values and then computes their resonant frequency.

5.1.7 Decibel calculator program

Examine the following ratio-to-decibel conversion program:

```
#include <stdio.h>
#include <math.h>

float ratio2db(float);

int main (void)
{
  float gain;

  printf("Enter amplifier power gain ratio: ");
  scanf("%f", &gain);

  printf("A power gain ratio of %f is equivalent to %f decibels.\n",
          gain, ratio2db(gain));

  return 0;
}

float ratio2db(float x)
{
  if (x < 0)
    x = -x;

  return 10 * log10(x);
}
```

Next, answer the following questions:

- Where is the `ratio2db` function being prototyped?
- What purpose does the `if` statement serve?
- Trace the execution of this program, line by line and step by step.

Challenges

- Modify this program to accept a voltage or current gain ratio value rather than power gain ratio value.
- Divide this program into two separate source files, then compile and demonstrate its operation.

- Identify a user-entered value resulting in a nonsensical output from this program, and modify the program to avoid this problem.

5.1.8 Logical-AND versus bitwise-AND

Compare and contrast the following expressions in C:

```
Result1 = A && B;
Result2 = A & B;
```

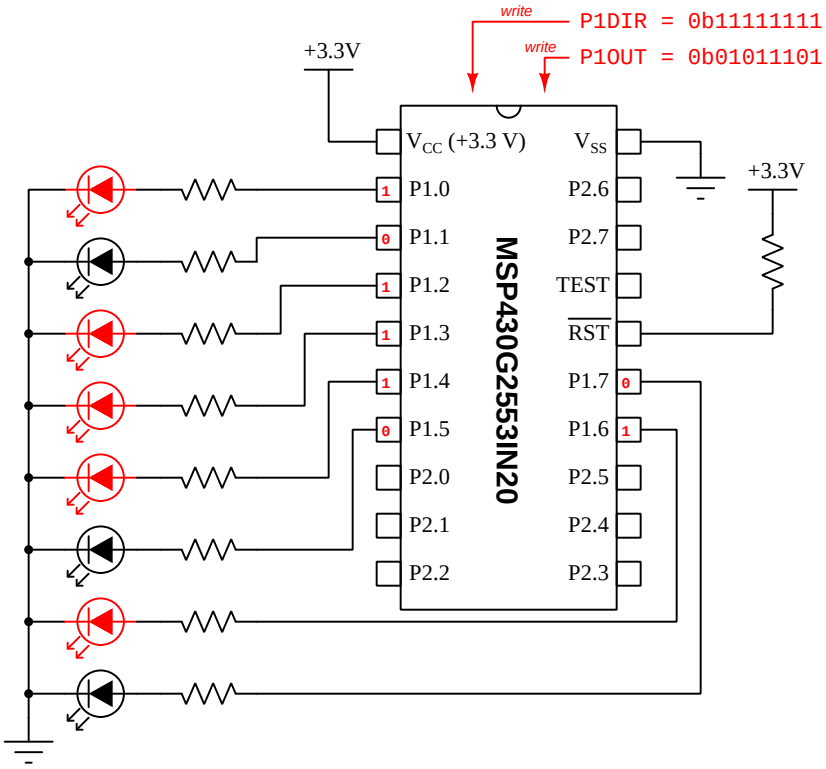
Provide an example showing how each of these expressions would be processed in a real C program.

Challenges

- What happens when integer variables rather than Boolean variables are used in a logical-AND expression?

5.1.9 Driving microcontroller output bits

Write a single line of C code that will turn on the `P1.1` and `P1.7` LEDs without affecting any of the other output line states (and without having to specify their current states):



Then, write another single line of C code to turn off the `P1.0`, `P1.2`, and `P1.3` LEDs, again without affecting or specifying any of the other lines' states:

Challenges

- What is the purpose of the `P1DIR` register in this microcontroller?

5.1.10 Bit-rotate program

Test-run the following program, and then explain how it *rotates* bits using the bitwise-shift function:

```
#include <stdio.h>

int main (void)
{
  int n = 0b11100000;
  int recycle, count;

  for (count = 0 ; count < 16 ; ++count)
  {
    recycle = n & 1;
    n = n >> 1;
    n |= (recycle << 7);
    printf("n = %X \n", n);
  }

  return 0;
}
```

Then, modify this program so that it rotates the bits in the opposite direction.

Challenges

- Describe a practical application for bitwise rotation.

5.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as "test cases[4]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[5] on an answer key!

[4] In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial's answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

[5] This approach is perfectly in keeping with the instructional philosophy of these learning modules: *teaching students to be self-sufficient thinkers*. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be "answer keys" available for the problems you will have to solve.

5.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) of uncertainty in the final digits: for example, the magnetic permeability of free space value given as $1.25663706212(19) \times 10^{-6}$ H/m represents a center value (i.e. the location parameter) of $1.25663706212 \times 10^{-6}$ Henrys per meter with one standard deviation of uncertainty equal to $0.0000000000019 \times 10^{-6}$ Henrys per meter.

Avogadro's number (N_A) = **6.02214076** $\times 10^{23}$ **per mole** (mol^{-1})

Boltzmann's constant (k) = **1.380649** $\times 10^{-23}$ **Joules per Kelvin** (J/K)

Electronic charge (e) = **1.602176634** $\times 10^{-19}$ **Coulomb** (C)

Faraday constant (F) = **96,485.33212...** $\times 10^{4}$ **Coulombs per mole** (C/mol)

Magnetic permeability of free space (μ_0) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space (ϵ_0) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space (Z_0) = 376.730313668(57) Ohms (Ω)

Gravitational constant (G) = $6.67430(15) \times 10^{-11}$ cubic meters per kilogram-seconds squared ($m^3/kg\text{-}s^2$)

Molar gas constant (R) = **8.314462618... Joules per mole-Kelvin** (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant (h) = **6.62607015** $\times 10^{-34}$ **joule-seconds** (J-s)

Stefan-Boltzmann constant (σ) = **5.670374419...** $\times 10^{-8}$ **Watts per square meter-Kelvin**^{4} ($W/m^2 \cdot K^4$)

Speed of light in a vacuum (c) = **299,792,458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Complete Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2018 CODATA Adjustment.

5.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a *spreadsheet*. Available on most personal computers (e.g. Microsoft Excel), *spreadsheet* software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell `B3`, cell `A8`). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

	A	B	C	D
1	Distance traveled	46.9	Kilometers	
2	Time elapsed	1.18	Hours	
3	Average speed	= B1 / B2	km/h	
4				
5				

Text labels contained in cells `A1` through `A3` and cells `C1` through `C3` exist solely for readability and are not involved in any calculations. Cell `B1` contains a sample distance value while cell `B2` contains a sample time value. The formula for computing speed is contained in cell `B3`. Note how this formula begins with an "equals" symbol (`=`), references the values for distance and speed by lettered column and numbered row coordinates (`B1` and `B2`), and uses a forward slash symbol for division (`/`). The coordinates `B1` and `B2` function as *variables*[6] would in an algebraic formula.

When this spreadsheet is executed, the numerical value `39.74576` will appear in cell `B3` rather than the formula `= B1 / B2`, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell `B1` or a different value for time is entered into cell `B2`, cell `B3`'s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell `B3` may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in `B1` and `B2`. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

[6] Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels "names"), but for simple spreadsheets such as those shown here it's usually easier just to use the standard coordinate naming for each cell.

Common[7] arithmetic operations available for your use in a spreadsheet include the following:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Powers (`^`)
- Square roots (`sqrt()`)
- Logarithms (`ln()` , `log10()`)

Parentheses may be used to ensure[8] proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the *quadratic formula*, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

	A	B
1	x_1	= (-B4 + sqrt((B4^2) - (4*B3*B5))) / (2*B3)
2	x_2	= (-B4 - sqrt((B4^2) - (4*B3*B5))) / (2*B3)
3	a =	9
4	b =	5
5	c =	-2

This example is configured to compute roots[9] of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and -2 have been inserted into cells `B3`, `B4`, and `B5`, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of *any* second-degree polynomial expression simply by entering the new a, b, and c coefficients into cells `B3` through `B5`. The numerical values appearing in cells `B1` and `B2` will be automatically updated by the computer immediately following any changes made to the coefficients.

[7] Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

[8] Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

[9] Reviewing some algebra here, a *root* is a value for x that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells `B1` and `B2`, respectively upon execution of the spreadsheet.

Alternatively, one could break up the long quadratic formula into smaller pieces like this:

$$y = \sqrt{b^2 - 4ac} \qquad z = 2a$$

$$x = \frac{-b \pm y}{z}$$

	A	B	C
1	x_1	= (-B4 + C1) / C2	= sqrt((B4^2) - (4*B3*B5))
2	x_2	= (-B4 - C1) / C2	= 2*B3
3	a =	9	
4	b =	5	
5	c =	-2	

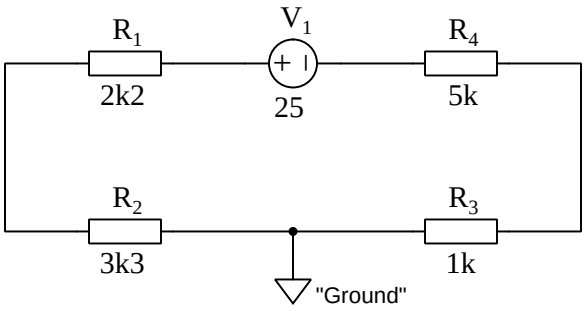
Note how the square-root term (y) is calculated in cell `C1`, and the denominator term (z) in cell `C2`. This makes the two final formulae (in cells `B1` and `B2`) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

[10] My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out *how* I constructed a solution. This is a general principle I believe all computer programmers should follow: *document and arrange your code to make it easy for other people to learn from it.*

5.2.3 Using C to analyze a series resistor circuit

Examine the program (written in the C language) to analyze the following DC series resistor circuit:




```
#include <stdio.h>

int main (void)
{
  float v1 = 25.0, r1 = 2.2e3, r2 = 3.3e3, r3 = 1e3, r4 = 5e3;
  float rtotal, i, vr1, vr2, vr3, vr4;

  rtotal = r1 + r2 + r3 + r4;
  i = v1 / rtotal;
  vr1 = i * r1;
  vr2 = i * r2;
  vr3 = i * r3;
  vr4 = i * r4;

  printf("Current = %f Amperes = %f milliAmperes\n", i, i*1000.0);
  printf("Resistor voltages = %f  %f  %f  %f  (Volts)\n", vr1, vr2, vr3, vr4);
  printf("Total resistor voltages = %f  (Volts)\n", vr1 + vr2 + vr3 + vr4);
  printf("Resistor powers = %f  %f  %f  %f  (Watts)\n", vr1*i, vr2*i,
          vr3*i, vr4*i);

  return 0;
}
```

Compile and run this code, and then answer the following questions:

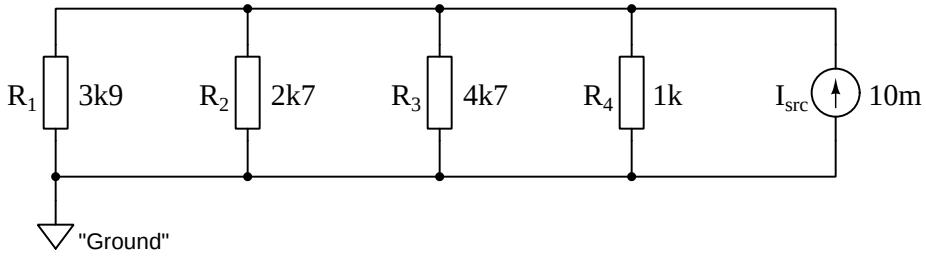
- Identify the order of execution for each line of code in this program (i.e. which line executes in which order)
- Identify the data type used for all the variables in this program, and how the `printf` statements properly reference this data type
- Identify how component voltages relate to each other in a series network, and how this property is expressed in the code
- Identify how component currents relate to each other in a series network, and how this property is expressed in the code
- Identify how component resistances relate to each other in a series network, and how this property is expressed in the code
- Modify the code to apply Kirchhoff's Voltage Law
- Modify the code to simulate a resistor failing shorted.
- Modify the code to simulate a resistor failing open.

Challenges

- What purpose, if any, is served by the Ground connection in this circuit?
- What is the purpose of the code lines beginning with `float`?
- What is the purpose of the `\n` symbols within the `printf` instructions?
- Modify the code to simulate adding a fifth resistor to this circuit.
- Modify the code to substitute a current source for the voltage source shown.
- Modify the code to calculate voltage between Ground and the negative terminal of the source.

5.2.4 Using C to analyze a parallel resistor circuit

Examine the program (written in the C language) to analyze the following DC parallel resistor circuit:




```
#include <stdio.h>

int main (void)
{
  float isrc = 10e-3, r1 = 3.9e3, r2 = 2.7e3, r3 = 4.7e3, r4 = 1e3;
  float rtotal, v, ir1, ir2, ir3, ir4;

  rtotal = 1/(1/r1 + 1/r2 + 1/r3 + 1/r4);
  v = isrc * rtotal;
  ir1 = v / r1;
  ir2 = v / r2;
  ir3 = v / r3;
  ir4 = v / r4;

  printf("Voltage = %f Volts\n", v);
  printf("Resistor currents = %f  %f  %f  %f  (milliAmperes)\n", ir1*1e3,
          ir2*1e3, ir3*1e3, ir4*1e3);
  printf("Total resistor currents = %f  (Amperes)\n", ir1 + ir2 + ir3 + ir4);
  printf("Resistor powers = %f  %f  %f  %f  (Watts)\n", ir1*v, ir2*v,
          ir3*v, ir4*v);

  return 0;
}
```

Compile and run this code, and then answer the following questions:

- Identify the order of execution for each line of code in this program (i.e. which line executes in which order)
- Identify how proper mathematical order-of-operation is maintained in the line(s) of code calculating parallel resistance
- Identify where any of the Joule's Law calculations could have been performed differently
- Identify how component voltages relate to each other in a parallel network, and how this property is expressed in the code
- Identify how component currents relate to each other in a parallel network, and how this property is expressed in the code
- Identify how component resistances relate to each other in a parallel network, and how this property is expressed in the code
- Modify the code to apply Kirchhoff's Current Law
- Modify the code to simulate a resistor failing shorted.
- Modify the code to simulate a resistor failing open.

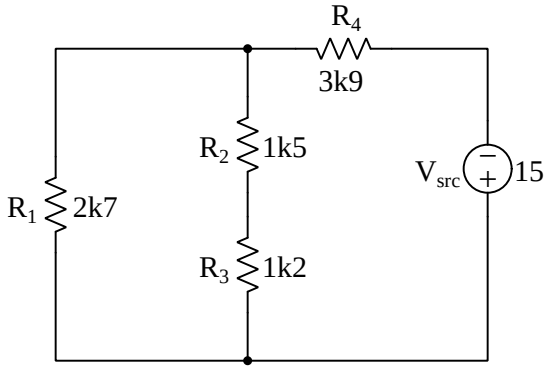
Challenges

- What purpose, if any, is served by the Ground connection in this circuit?
- What is the purpose of the code lines beginning with `float`?
- What is the purpose of the `\n` symbols within the `printf` instructions?
- Modify the code to simulate adding a fifth resistor to this circuit.
- Modify the code to substitute a voltage source for the current source shown.
- Modify the code to calculate current through either wire between resistors R_3 and R_4.

5.2.5 Using C to analyze a series-parallel resistor circuit

Examine the program (written in the C language) to analyze the DC series-parallel resistor circuit shown in schematic form on the following page:

```
#include <stdio.h>

int main (void)
{
  float vsrc = 15.0, r1 = 2700.0, r2 = 1500.0, r3 = 1200.0, r4 = 3900.0;
  float rtotal, vr1, ir1, vr2, ir2, vr3, ir3, vr4, ir4;
  float r23, r123, isrc;

  r23 = r2 + r3;
  r123 = 1/(1/r1 + 1/r23);
  rtotal = r123 + r4;
  isrc = vsrc / rtotal;
  ir4 = isrc;
  vr4 = ir4 * r4;
  vr1 = vsrc - vr4;
  ir1 = vr1 / r1;
  ir2 = ir3 = vr1 / r23;
  vr2 = ir2 * r2;
  vr3 = ir3 * r3;

  printf("Source current = %f milliAmperes\n", isrc*1000.0);
  printf("Resistor currents = %f  %f  %f  %f  (milliAmperes)\n", ir1*1000.0,
          ir2*1000.0, ir3*1000.0, ir4*1000.0);
  printf("Resistor voltages = %f  %f  %f  %f  (Volts)\n", vr1, vr2, vr3, vr4);
  printf("Resistor powers = %f  %f  %f  %f  (milliWatts)\n", ir1*vr1*1e3,
  ir2*vr2*1e3, ir3*vr3*1e3, ir4*vr4*1e3);

  return 0;
}
```



Compile and run this code, and then answer the following questions:

- Explain what Ohm's Law is, and how it is applied in the code
- Modify the line(s) of code implementing Joule's Law so that the results are expressed in Watts rather than in milliWatts
- Identify how various properties of series networks are applied in the code
- Identify how various properties of parallel networks are applied in the code
- Modify the code to apply Kirchhoff's Voltage Law
- Modify the code to apply Kirchhoff's Current Law
- Identify any lines of code whose order could be swapped without affecting the accuracy of the final results

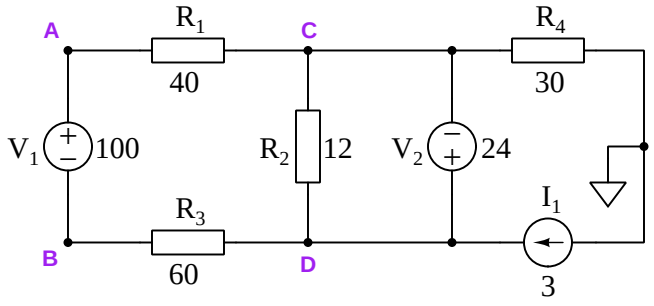
Challenges

- Modify the code to make this a four-resistor series circuit.
- Modify the code to make this a four-resistor parallel circuit.

5.2.6 Using C to analyze a multi-source circuit

Examine the program (written in the C language) to analyze the multi-source circuit shown in schematic form below the source-code listing:

```
#include <stdio.h>

int main (void)
{
  float v1 = 100.0, v2 = 24.0, i1 = 3.0;
  float r1 = 40.0, r2 = 12.0, r3 = 60.0, r4 = 30.0;
  float vac, vbd, vcd, vc, vd, iv1;

  vcd = -v2;
  vc = i1 * r4;
  vd = vc + v2;
  iv1 = (v1 - vcd) / (r1 + r3);
  vac = iv1 * r1;
  vbd = -(iv1 * r3);

  printf("V_A = %f Volts\n", vac + vc);
  printf("V_B = %f Volts\n", vbd + vd);
  printf("V_C = %f Volts\n", vc);
  printf("V_D = %f Volts\n", vd);

  return 0;
}
```



Compile and run this code, and then answer the following questions:

- Identify the electrical principle(s) used in the calculation `vcd = -v2;`
- Identify the electrical principle(s) used in the calculation `vc = i1 * r4;`
- Identify the electrical principle(s) used in the calculation `vd = vc + v2;`
- Identify the electrical principle(s) used in the calculation `iv1 = (v1 - vcd) / (r1 + r3);`
- Identify the electrical principle(s) used in the calculation `vac = iv1 * r1;`
- Identify the electrical principle(s) used in the calculation `vbd = -(iv1 * r3);`
- How could you modify this program to accept user-input values for V_1, V_2, and I_1?

Challenges

- How much current flows through R_2, and in which direction?
- How much current flows through V_2, and in which direction?
- Which of these components function as sources, and which function as loads?

5.2.7 Using C to calculate capacitive reactance

Examine the program (written in the C language) to calculate reactance for a capacitor given frequency and capacitance values entered by the user at run-time:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float f, C;

  printf("Enter frequency in Hertz: ");
  scanf("%f", &f);

  printf("Enter capacitance in microFarads: ");
  scanf("%f", &C);

  C = C / 1e6;           // Converts uF to F

  printf("Reactance = %f Ohms\n", 1/(2*M_PI*f*C));

  return 0;
}
```

Compile and run this code, and then answer the following questions:

- Explain what reactance is, and how it differs from resistance
- How would this reactance value be expressed as an *impedance* in rectangular form?
- How would this reactance value be expressed as an *impedance* in polar form?
- Modify the code to calculate reactance for an inductor

Challenges

- In what ways are real capacitors more complicated than just being pure reactances in AC circuits?

5.2.8 Using C to analyze a series AC RLC circuit

Examine the program (written in the C language) to analyze an AC series resistor-inductor-capacitor circuit:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float R, C, L, f, vsrc, xc, xl, xtotal, ztotal, i;

  printf("Enter resistor value in Ohms: ");  scanf("%f", &R);
  printf("Enter capacitor value in microFarads: ");  scanf("%f", &C);
  printf("Enter inductor value in milliHenrys: ");  scanf("%f", &L);
  printf("Enter source frequency in Hertz: ");  scanf("%f", &f);
  printf("Enter source voltage in Volts: ");  scanf("%f", &vsrc);

  C = C * 1e-6 ;   L = L * 1e-3;
  xc = 1/(2 * M_PI * f * C);
  xl = 2 * M_PI * f * L;
  xtotal = xl - xc;
  ztotal = sqrt(pow(R,2) + pow(xtotal,2));
  i = vsrc / ztotal;

  printf("Z_r = %f - 0j Ohms\n", R);
  printf("Z_c = 0 - %fj Ohms\n", xc);
  printf("Z_l = 0 + %fj Ohms\n", xl);
  printf("Z total (rectangular) = %f + %fj Ohms\n", R, xtotal);
  printf("Z total (polar) = %f Ohms @ %f degrees\n", ztotal,
         atan((xtotal)/R)*180/M_PI);
  printf("Current = %f mA\n", i * 1000);
  printf("Resistor voltage = %f V\n", i * R);
  printf("Capacitor voltage = %f V\n", i * xc);
  printf("Inductor voltage = %f V\n", i * xl);

  return 0;
}
```

Compile and run this code, and then answer the following questions:

- Identify how various properties of series networks are applied in the code
- Find a combination of values resulting in a condition at or near *resonance*

- How is total impedance calculated in this program without using complex-number variables?
- How are metric prefixes accommodated in this program?
- How could this program be simplified to use fixed component values rather than receive input from the user?
- Identify where proper mathematical order-of-operations are enforced by the use of parentheses

Challenges

- Suppose we run this program with values that do not achieve resonance. How could we tell from the output of the program which way we need to adjust the source frequency to approach resonance?

5.2.9 Using C to analyze a parallel AC RLC circuit

Examine the program (written in the C language) to analyze an AC parallel resistor-inductor-capacitor circuit:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float R, C, L, f, vsrc, ztotal, xc, xl, ir, ic, il, itotal;

  printf("Enter resistor value in Ohms: ");  scanf("%f", &R);
  printf("Enter capacitor value in microFarads: ");  scanf("%f", &C);
  printf("Enter inductor value in milliHenrys: ");  scanf("%f", &L);
  printf("Enter source frequency in Hertz: ");  scanf("%f", &f);
  printf("Enter source voltage in Volts: ");  scanf("%f", &vsrc);

  C = C * 1e-6 ;   L = L * 1e-3;
  xc = 1/(2 * M_PI * f * C);
  xl = 2 * M_PI * f * L;
  ir = vsrc / R;   ic = vsrc / xc;   il = vsrc / xl;

  itotal = sqrt(pow(ir,2) + pow(il - ic,2));
  ztotal = vsrc / itotal;

  printf("Z_r = %f - 0j Ohms\n", R);
  printf("Z_c = 0 - %fj Ohms\n", xc);
  printf("Z_l = 0 + %fj Ohms\n", xl);
  printf("Z total (polar) = %f Ohms @ %f degrees\n", ztotal,
         atan((il-ic)/ir)*180/M_PI);
  printf("I total = %f mA\n", itotal * 1000.0);
  printf("Resistor current (rectangular) = %f + 0j mA\n", ir * 1000.0);
  printf("Capacitor current (rectangular) = 0 + %fj mA\n", ic * 1000.0);
  printf("Inductor current (rectangular) = 0 - %fj mA\n", il * 1000.0);

  return 0;
}
```

Compile and run this code, and then answer the following questions:

- Explain what "polar" and "rectangular" refer to in AC circuit calculations
- Identify how various properties of parallel networks are applied in the code

- Find a combination of values resulting in a condition at or near *resonance*
- How is total impedance calculated in this program without using complex-number variables?
- Explain the purpose of the `1e-6` and `1e-3` multiplying coefficients for `C` and `L`
- How are metric prefixes accommodated in this program?

Challenges

- Suppose we run this program with values that do not achieve resonance. How could we tell from the output of the program which way we need to adjust the source frequency to approach resonance?

5.2.10 Using C to analyze a series AC resistor-capacitor circuit

The following program accepts user input for resistor and capacitor values and AC voltage source values, then proceeds to calculate voltage dropped across the resistor for different user-entered frequencies in this series network:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float f, R, C, Vsrc, Xc, Ztotal, I;

  printf("Enter the AC source's value in Volts: ");
  scanf("%f", &Vsrc);

  printf("Enter the resistor's value in Ohms: ");
  scanf("%f", &R);

  printf("Enter the capacitor's value in Farads: ");
  scanf("%f", &C);

  while(1)
  {
    printf("\nEnter the AC source's frequency in Hertz: ");
    scanf("%f", &f);

    Xc = 1 / (2 * M_PI * f * C);
    Ztotal = sqrt(pow(R,2) + pow(Xc,2));
    I = Vsrc / Ztotal;
    printf("Resistor voltage drop = %f Volts\n", I * R);
  }

  return 0;
}
```

Explain the purpose for each line of code found within this program.

Explain in detail the foundational concepts of electric circuits applied in this program.

What type of *filter* network is this program simulating, and how may we tell?

How might we modify the code to simulate a different type of filter network than this?

Challenges

- Is the math library necessary to be linked with this code during compilation? Why or why not?

5.2.11 Using complex numbers in C to analyze a series-parallel AC RLC circuit

The following C program simulates a series-parallel RLC circuit powered by an AC voltage source:

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main (void)
{
  float R, C, L, f, Vsrc;
  float complex j, Zl, Zc, Zt, It, Vl, Vc, Vr, Ic, Il, Ir;
  j = I;    // I prefer using "j" rather than "I"

  printf("Enter resistor value in Ohms: ");  scanf("%f", &R);
  printf("Enter capacitor value in microFarads: ");  scanf("%f", &C);
  printf("Enter inductor value in milliHenrys: ");  scanf("%f", &L);
  printf("Enter source frequency in Hertz: ");  scanf("%f", &f);
  printf("Enter source voltage in Volts: ");  scanf("%f", &Vsrc);

  C = C * 1e-6 ;   L = L * 1e-3;
  Zc = 1/(2 * M_PI * f * C * j);
  Zl = 2 * M_PI * f * L * j;
  Zt = 1/(1/R + 1/Zc) + Zl;

  It = Il = Vsrc / Zt;
  Vl = It * Zl;
  Vc = Vr = Vsrc - Vl;
  Ic = Vc / Zc;
  Ir = Vr / R;

  printf("Total current = %f @ %f deg \n", cabs(It), carg(It) * 180 / M_PI);
  printf("Resistor voltage = %f @ %f deg \n", cabs(Vr), carg(Vr) * 180 / M_PI);
  printf("Resistor current = %f @ %f deg \n", cabs(Ir), carg(Ir) * 180 / M_PI);
  printf("Capacitor voltage = %f @ %f deg \n", cabs(Vc), carg(Vc) * 180 / M_PI);
  printf("Capacitor current = %f @ %f deg \n", cabs(Ic), carg(Ic) * 180 / M_PI);
  printf("Inductor voltage = %f @ %f deg \n", cabs(Vl), carg(Vl) * 180 / M_PI);
  printf("Inductor current = %f @ %f deg \n", cabs(Il), carg(Il) * 180 / M_PI);

  return 0;
}
```

Examine this program, and based on the source code listing determine the topology of this RLC

circuit: i.e. which components are in parallel, which in series, etc.

How does the use of complex numbers simplify the programming to perform all the necessary AC calculations?

Challenges

- Is the math library necessary to be linked with this code during compilation? Why or why not?
- Explain why some of the variables in this program are simply declared as regular floating-point, while others must be declared as *complex* floating-point.

5.2.12 Using C to plot a sine wave

Examine the program (written in the C language) intended to plot a crude sine wave pattern to the computer's standard output (terminal screen):

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= (2 * M_PI) ; x = x + 0.2)
  {
    for (n = 0 ; n < (10 * sin(x) + 10) ; ++n)
      printf(" ");

    printf("*\n");
  }

  return 0;
}
```

First, compile and run this code to examine what its output looks like.

Next, compile and run the following program using the same `for()` loops as the first, but outputting the values of `x` and `n` with each "pass" of execution through each loop rather than space and "`*`" characters as before:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float x;
  int n;

  for (x = 0 ; x <= (2 * M_PI) ; x = x + 0.2)
  {
    printf("\nx=%0.1f ; n=", x);

    for (n = 0 ; n < (10 * sin(x) + 10) ; ++n)
      printf("%i,", n);
  }

  return 0;
}
```

Finally, explain in your own words the flow of execution when run (i.e. which lines of code execute in which order, and the values of both `x` and `n` throughout that execution), for the first few "passes" through the outer `for()` loop, and how the original version of the program plots a sine wave to the console using these two loops.

Challenges

- How may you edit the code to make the sine wave have a greater amplitude?
- How may you edit the code to make the sine wave have a greater frequency?
- How may you edit the code to shift the center of the sine wave?
- How may you edit the code to make more cycles appear of the sine wave?
- Estimate the RMS value of this sine wave.

5.2.13 Geometric sequence counting program

A *geometric sequence* is a sequence of numbers where each successive number is a fixed-multiple of the one preceding it. For example, a geometric sequence with a factor of 3 looks like this: 1, 3, 9, 27, 81, $\cdots$

Write a program that prints the first six numbers of any geometric sequence, requesting the user to enter the starting value for the sequence and the multiplying factor (both integer values). The following is some code to get you started:

```
#include <stdio.h>

int main (void)
{
  int start, factor;  // User-entered values
  int num, count;     // "num" is the sequence number that gets incremented
                      // "count" will control your loop

  printf("Enter the starting value: ");
  scanf("%i", &start);

  printf("Enter the multiplication factor: ");
  scanf("%i", &factor);

  //
  // YOUR LOOP GOES HERE!
  //

  return 0;
}
```

After you get your geometric-sequence program working well, modify it to produce an *arithmetic* sequence instead, where successive values are separated by a constant difference (e.g. 1, 5, 9, 13, 17, 21).

Challenges

- Modify this program so that the user controls how many elements of the geometric sequence get printed to the console.
- Modify the program and identify user-entered values resulting in a geometric sequence that grows smaller rather than larger.

5.2.14 RC time-constant calculator program

Write a program to calculate the time constant of a simple resistor-capacitor (RC) network, requesting the user to enter the resistance value in Ohms and the capacitance value in Farads. The following is some code to get you started:

```
#include <stdio.h>

int main (void)
{
  float R, C;

  printf("Enter the resistor's value in Ohms: ");
  scanf("%f", &R);

  //
  // Rest of your code goes here
  //

  return 0;
}
```

After you have a basic version of your program functional, add more code to it so that it additionally displays "Time constant is greater than 1 minute" if indeed the calculated time constant value is greater than one minute.

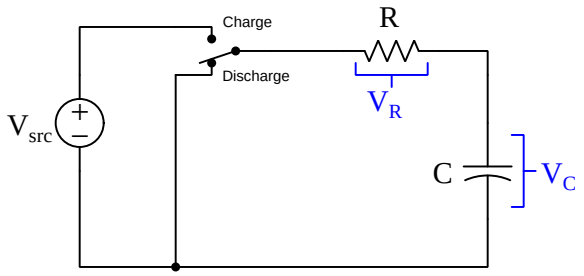
After that, "fool-proof" your program against anyone who might enter a *negative* value for either resistance or capacitance so that the program does not generate a non-sensical negative value for time constant.

Challenges

- Modify your program to display the time constant in milliseconds rather than seconds.

5.2.15 Using C to analyze an RC charging-discharging circuit

Examine the program (written in the C language) to analyze a resistor-capacitor circuit with a SPDT switch to make the capacitor charge or discharge:




```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float Vsrc, R, C, tau, time;

  printf("Enter value of source in Volts: ");
  scanf("%f", &Vsrc);

  printf("Enter value of resistor in Ohms: ");
  scanf("%f", &R);

  printf("Enter value of capacitor in microFarads: ");
  scanf("%f", &C);

  C = C / 1e6;
  tau = R * C;

  printf("\nCharging\n");

  for (time = 0.0 ; time <= 5*tau ; time = time + (tau / 3.0))
  {
    printf("Time = %f sec \t",time);
    printf("V_c = %f Volts \t", Vsrc * (1 - exp(-time / tau)));
    printf("V_r = %f Volts \n", Vsrc * exp(-time / tau));
  }

  printf("\nDischarging\n");
```

```
  for (time = 0.0 ; time <= 5*tau ; time = time + (tau / 3.0))
  {
    printf("Time = %f sec \t",time);
    printf("V_c = %f Volts \t", Vsrc * exp(-time / tau));
    printf("V_r = %f Volts \n", Vsrc * exp(-time / tau));
  }

  return 0;
}
```

Compile and run this code, and then answer the following questions:

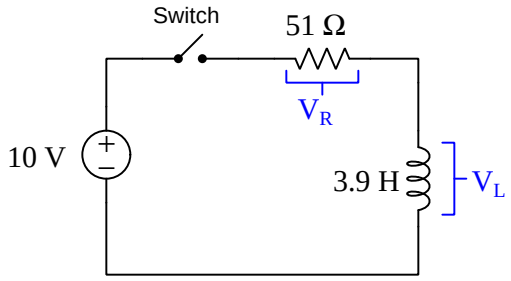
- Qualitatively predict the trajectories of the resistor and capacitor voltages starting at the time when switch moves to the "Charge" position, assuming the capacitor begins in a zero-energy state
- Qualitatively predict the trajectories of the resistor and capacitor voltages starting at the time when switch moves to the "Discharge" position, assuming the capacitor began at full source voltage
- Which components are functioning as sources and which as loads during the "Charge" period?
- Which components are functioning as sources and which as loads during the "Discharge" period?
- Which portion(s) of the code listing determine the time increments over which the charging and discharging analyses take place?
- Identify how to modify the program so that the charging and discharging analyses proceed up to and including *ten* time constants' worth of time.
- Identify how to modify the program so that the charging and discharging analyses each begin at some point in time *after* zero.
- Identify how to modify the program so that the charging and discharging analyses halt when a certain voltage or current value is reached, rather than halting after a certain amount of time has passed.

Challenges

- Modify this program to calculate current in either the charge or the discharge period

5.2.16 Writing an LR time-delay analysis program

Write a program in the C language to compute inductor voltage and resistor voltage over time after closing the switch in the following LR circuit:



Be sure to write your program so that voltage values are printed from time zero to at least five time constants' worth of time (5τ) following switch closure.

Challenges

- Explain why it is a good problem-solving technique to analyze the circuit on your own (i.e. "by hand" using just a calculator) before attempting to write a program that does the same.
- What would happen in a real circuit if the switch were opened after the inductor had opportunity to fully energize (e.g. after resistor voltage had risen to equal source voltage)?

5.2.17 Writing a cutoff frequency calculator program

Write a program in the C language to compute the cutoff frequency for a simple resistor-capacitor (RC) or inductor-resistor (LR) filter network, either high-pass or low-pass, given component values specified by the user. The following is some code to get you started:

```
#include <stdio.h>

int main (void)
{
  int choice;
  float R, C, L;

  printf("Enter 1 for RC filter or 2 for LR filter: ");
  scanf("%i", &choice);

  if (choice == 1)
  {
    // Your code goes here
  }

  else
  {
    // Your code goes here
  }

  return 0;
}
```

Why do we not need the user to select whether their filter network will be high-pass or low-pass?

Also, identify a practical application for a filter network.

Challenges

- What is the meaning of the "cutoff frequency" for a filter network?
- Modify this program so that only one pair of `printf`/`scanf` instructions are necessary to solicit the resistor's value from the user.

5.2.18 Writing a low-pass filter simulation program

Write a program in the C language to compute the output signal voltage from a low-pass filter consisting of a 20 kΩ resistor and a 3.3 nF capacitor, printing input signal frequency values and output signal voltage values from 2000 Hz to 3000 Hz in 100 Hz increments assuming a constant AC source voltage of 2 Volts.

Also, sketch a schematic diagram for this type of filter network.

Challenges

- A good problem-solving strategy is to analyze the filter network on your own (i.e. with just a hand calculator) before trying to write a program to do the same. How exactly will analyzing the circuit on your own assist you in writing software to do it automatically?
- Modify your program to analyze a high-pass filter network consisting of the same components, differently arranged.

5.2.19 Bitwise operation practice

Choose arbitrary values for `n` and `mask` which you may then test using the following program. Write the binary values for both of your chosen quantities and show how each bitwise operation will work on them, before running the program to check your answers.

```
#include <stdio.h>

int main (void)
{
  int n, mask;

  printf("Enter value of first operand: ");
  scanf("%i", &n);

  printf("Enter value of second operand (i.e. the ‘mask’): ");
  scanf("%i", &mask);

  printf("\n");
  printf("Bitwise-AND between 0x%X and 0x%X = %X \n", n, mask, n & mask);
  printf("Bitwise-OR between 0x%X and 0x%X = %X \n", n, mask, n | mask);
  printf("Bitwise-XOR between 0x%X and 0x%X = %X \n", n, mask, n ^ mask);
  printf("Bitwise-complement of 0x%X = 0x%X \n", n, ~n);
  printf("Left-shift 0x%X by four bits = 0x%X \n", n, n << 4);
  printf("Right-shift 0x%X by three bits = 0x%X \n", n, n >> 3);

  return 0;
}
```

Next, modify this program to prompt the user to choose how many places to shift the bits of the first operand (instead of having fixed bit-shifts of four and three).

Challenges

- Modify the program to multiply the first operand by four, without using the multiplication (*) operator.

5.2.20 XOR cryptography

Translate the following message into ASCII codes, then encrypt each of those bytes with the key `0b 0000 0110` using bitwise-XOR operations. Show what that "cyphertext" message looks like when viewed as ASCII characters:

> Hello, world!

Challenges

- How difficult would it be to guess the key if all you had access to was the cypertext?

5.2.21 Byte-shifting algorithm

Some computer systems are designed in such a way that digital words are stored in memory and/or communicated over serial networks in *byte-swapped* order, where the least-significant and most-significant bytes of a 16-bit word appear in reverse order. For example, the number `0x1F4C` might be formatted such that the least-significant byte `4C` comes first and the most-significant byte `1F` comes second.

Complete the following program, where the user will enter a 16-bit number in hexadecimal format and then display both the user-entered value (`input`) and the byte-swapped value (`output`):

```
#include <stdio.h>

int main (void)
{
  int input, output;

  printf("Enter number: ");
  scanf("%i", &input);

  // Place your code here!

  printf("Original value = %x \n", input);
  printf("Byte-swapped value = %x \n", output);

  return 0;
}
```

Challenges

- How would your program need to be written if the purpose was to swap the first and second 16-bit words within a 32-bit integer?

5.2.22 Sine calculator program

Suppose a student writes the following program to calculate sine values between 0 and 90 degrees:

```
#include <stdio.h>
#include <math.h>

int main (void)
{
  float angle;

  for (angle = 0.0 ; angle <= 90.0 ; angle = angle + 10.0)
    printf("Angle = %.0f \t sine(%.0f) = %1.5f\n", angle, angle, sin(angle));

  return 0;
}
```

However, the student is disappointed in the results after compiling and running this code. Instead of seeing the sine of the increasing angle increase from 0 to 1 as the angle progresses from 0 to 90, a sequence of seemingly random values appears, some positive and some negative. It is then that the student realizes that the `sin()` function in C assumes the use of *radians* rather than *degrees* as the unit of angular measurement.

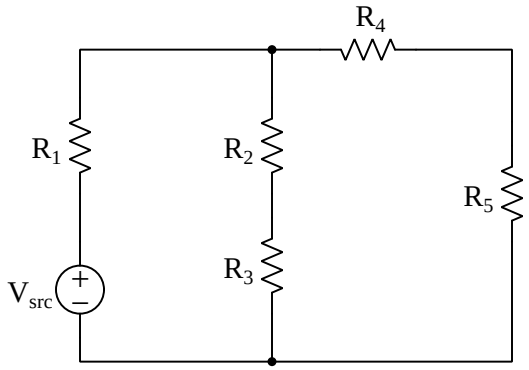
Modify this program to convert degrees into radians so that the `sin()` function will be satisfied, and place this degree-to-radian conversion within its own function that gets called by the `printf` instruction.

Challenges

- Modify this program so that it increments the angle in smaller steps.
- Modify this program to start at 90 degrees and decrement to 0 degrees.

5.2.23 Using C arrays to analyze a resistor circuit

Suppose we need a program (written in the C language) to analyze the following circuit:



Here is the basic outline of a program to do this, using arrays to store all the resistance, voltage, and current values:

```
#include <stdio.h>

int main (void)
{
  float v[6];
  float i[6];
  float r[6];

  // Your code goes here!

  return 0;
}
```

Your task is to make this program complete, so that in the end all resistor voltages and currents are displayed to the user, for the following component values: $V_{src} = 28$ Volts, $R_1 = 2\ \Omega$, $R_2 = 4\ \Omega$, $R_3 = 6\ \Omega$, $R_4 = 8\ \Omega$, and $R_5 = 2\ \Omega$.

- Explain what Ohm's Law is, and how it is applied in your code
- Identify how various properties of series networks are applied in your code
- Identify how various properties of parallel networks are applied in your code

Challenges

- Modify the code to apply Kirchhoff's Voltage Law around any loop (of your choice) and prove an algebraic sum of zero.

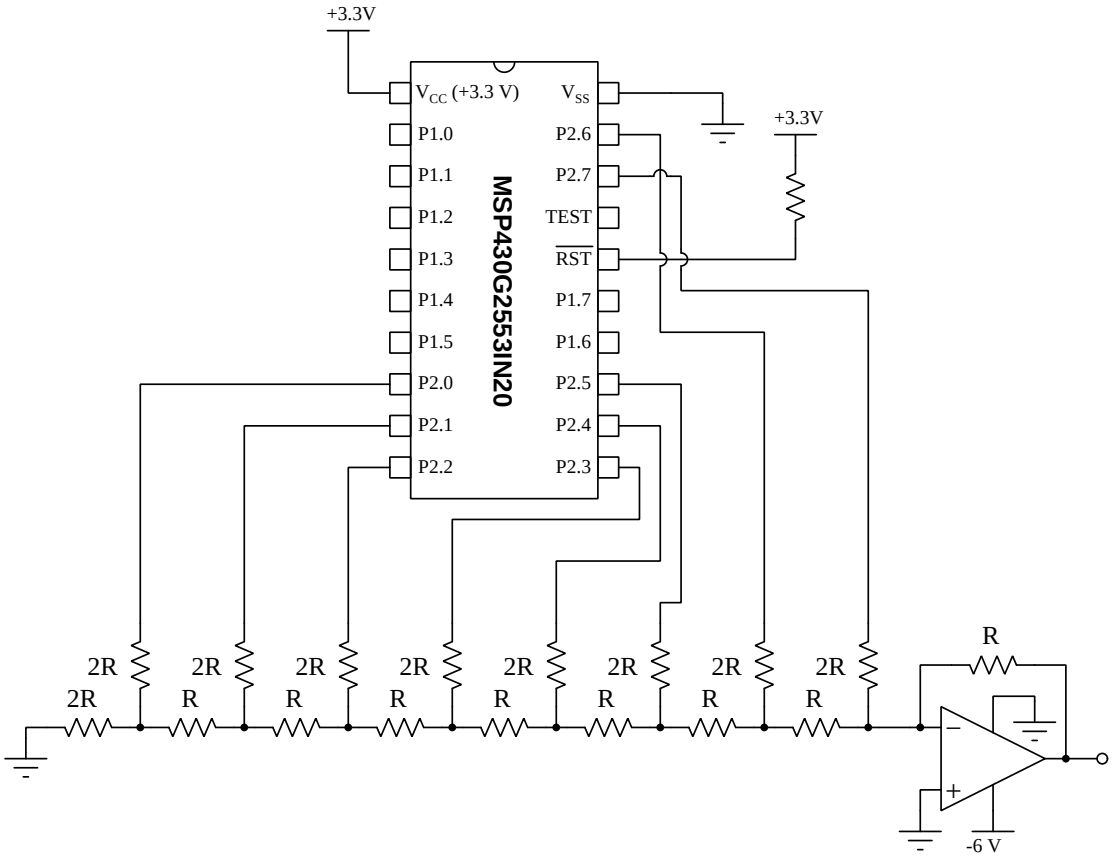
5.2.24 Sine look-up table

The following program written in C for the Texas Instruments MSP430G2553 microcontroller uses an array of "character" (eight-bit integer) values representing samples along one cycle of a sine wave function:

C code listing

```
#include "msp430G2553.h"

unsigned char wave[127] =     // Sine table ranging 0 to 255
 {127,134,140,146,153,159,165,171,177,182,
  188,194,199,204,209,214,218,223,227,230,
  234,237,240,243,246,248,250,251,253,254,
  255,255,255,255,254,253,252,251,249,247,
  245,242,239,236,232,229,225,220,216,211,
  206,201,196,191,185,180,174,168,162,156,
  149,143,137,131,124,118,112,106,99,93,
  87,81,75,70,64,59,54,49,44,39,
  35,30,26,23,19,16,13,10,8,6,
  4,3,2,1,0,0,0,0,1,2,
  4,5,7,9,12,15,18,21,25,28,
  32,37,41,46,51,56,61,67,73,78,
  84,90,96,102,109,115,121};

void main(void)
{
  unsigned int i;             // Used in for() loop

  WDTCTL = WDTPW + WDTHOLD;   // Stop WDT
  DCOCTL = 0xE0;              // Maximum DCO frequency range (DCO = 7)
  BCSCTL1 = 0x0F;             // Maximum basic clock frequency (RSEL = 15)
  BCSCTL2 = 0x00;             // Sets DCOCLK as the source for MCLK and SMCLK
  P2SEL = 0x00;               // Sets P2.6 + P2.7 to standard output mode
  P2DIR = 0xFF;               // All P2.x pins are outputs

  while(1)
  {
    for (i = 0; i < 127; ++i)
      P2OUT = wave[i];
  }
}
```

The purpose of this program is to drive the microcontroller's Port 2 pins high and low in the proper order to synthesize an analog sinusoidal signal, effectively making a sine-wave signal generator. An $R/2R$ resistor "ladder" network functions as a simple digital-to-analog converter to turn Port 2's binary output values into an analog signal voltage:



Explain how the array gets used in this program to "look up" the sine wave amplitude values one at a time.

Identify where in the table of data we see the sine wave reaching its positive and negative peak values.

Challenges

- Identify what would need to be modified in this program to alter the output frequency.
- Identify what would need to be modified in this program to provide more resolution in the time domain, so that each step of the `for()` loop represented a smaller angle increment for the sine function.

5.2.25 Array-reversal program

The following program accepts input from the user to populate a five-element integer array, then prints those five values in the same order:

```
#include <stdio.h>

int main (void)
{
  int value[5], count;

  for (count = 0 ; count < 5 ; ++count)
  {
     printf("Enter value for element %i :", count);
     scanf("%i", &value[count]);
  }

  for (count = 0 ; count < 5 ; ++count)
  {
     printf("Element %i = %i\n", count, value[count]);
  }

  return 0;
}
```

Modify this program to display the entered values in reverse order from how the user entered them.

Then, modify the program to use pointers when printing the stored values, rather than array subscripts.

Challenges

- How could you write a program that actually swaps the array elements' values, rather than merely printing them in reverse order?

5.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what's wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.

5.3.1 Find mistakes in a very simple program

Multiple errors exist in this program. Find them all, then test your corrections by seeing whether or not your edited source code will compile and run properly:

```
include <stdio.h>

int main (void)
{
  printf("This is a simple program/n");
  printf("but it has mistakes in it.\n");

  return 0
}
```

Challenges

- What suggestions would you offer to anyone in order to help them find such errors?

5.3.2 Find mistakes in a millimeter conversion program

Multiple errors exist in this program, which is supposed to convert inches into millimeters. Find them all, then test your corrections by seeing whether or not your edited source code will compile and run properly:

```
#include (stdio.h)

int main (void)
{
  float inches;

  printf("Enter the dimension in inches: ");
  scanf("%f", inches);

  printf("The dimension is %f millimeters.\n", inches / 25.4);

  return 0:
}
```

Challenges

- After fixing the errors, re-write this program to perform some other numerical conversion.
- Could we use an integer variable for `inches` rather than a floating-point? Why or why not?

5.3.3 Case-sensitivity in variable names

Suppose a student is wondering whether the C programming language differentiates between upper- and lower-case characters in variable names. For example, would `value` and `VALUE` and `Value` and `vALuE` be considered distinct and different variables when declared, or would all those names be treated as synonymous by the compiler?

Write a program in C to prove whether or not capitalization matters in variable names.

Challenges

- What qualities are necessary in a "test program" such as this to confirm or refute a hypothesis, such as variable names being case-sensitive?

5.3.4 Find mistake in a function-calling program

An error exists in this program. First, compile and run this program to see what the problem is in its output, and then fix the error:

```
#include <stdio.h>
#include <math.h>

void doubler(int);

int main (void)
{
  int n;

  printf("Enter an integer: ");
  scanf("%i", &n);

  printf("The entered value is %i\n", n);
  doubler(n);
  printf("The doubled value is %i\n", n);

  return 0;
}

void doubler(int n)
{
  n = 2 * n;
}
```

Challenges

- A diagnostic technique is to insert `printf` instructions at different points in the program to display such things as variable values. Identify where you might insert a `printf` instruction within the original (faulty) code to help diagnose the problem.

5.3.5 Automatic and static function variable testing

Variables declared within a function are known as *automatic* variables in the C language, and their scope is local to that function. Variables internal to a function declared as `static` are still local to that function, but behave differently from automatic variables.

Examine the following C program and its output when run, explaining in your own words what you think is happening in terms of these internal function variables and where they are located in the computer's memory:

```
#include <stdio.h>

void incr_m(void);
void incr_n(void);
void incr_q(void);
void incr_r(void);

void main (void)
{
  incr_m();   incr_n();   incr_q();   incr_r();
  printf("\n");
  incr_m();   incr_n();   incr_q();   incr_r();
}

void incr_m()
{
  int m;
  ++m;
  printf("m = %i    Address of m = %p \n", m, &m);
}

void incr_n()
{
  int n;
  ++n;
  printf("n = %i    Address of n = %p \n", n, &n);
}

void incr_q()
{
  static int q;
  ++q;
  printf("q = %i    Address of q = %p \n", q, &q);
}
```

```
void incr_r()
{
  static int r;
  r = r + 10;
  printf("r = %i    Address of r = %p \n", r, &r);
}
```

Program output:

```
m = -1078869827    Address of m = 0xbfb1c004
n = -1078869826    Address of n = 0xbfb1c004
q = 1    Address of q = 0x80496f0
r = 10    Address of r = 0x80496f4

m = -1078869825    Address of m = 0xbfb1c004
n = -1078869824    Address of n = 0xbfb1c004
q = 2    Address of q = 0x80496f0
r = 20    Address of r = 0x80496f4
```

Challenges

- Why do `m` and `n` begin as negative values?

Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

Appendix B

Instructional philosophy

B.1 First principles of learning

- **Anyone can learn anything** given appropriate time, effort, resources, challenges, encouragement, and expectations. Dedicating time and investing effort are the student's responsibility; providing resources, challenges, and encouragement are the teacher's responsibility; high expectations are a responsibility shared by both student and teacher.

- **Transfer is not automatic.** The human mind has a natural tendency to compartmentalize information, which means the process of taking knowledge learned in one context and applying it to another usually does not come easy and therefore should never be taken for granted.

- **Learning is iterative.** The human mind rarely learns anything perfectly on the first attempt. Anticipate mistakes and plan for multiple tries to achieve full understanding, using the lessons of those mistakes as feedback to guide future attempts.

- **Information is absorbed, but understanding is created.** Facts and procedures may be memorized easily enough by repeated exposure, but the ability to reliably apply principles to novel scenarios only comes through intense personal effort. This effort is fundamentally creative in nature: casting new concepts in one's own words, running mental experiments to perceive patterns and trends, etc. These acts of making knowledge "one's own" need not be perfect in order to be effective, as the value lies in the activity and not necessarily the finished product. Depth of understanding and permanence of memory are both directly related to mental effort during the learning process – i.e. *easy learning is shallow and fleeting.*

- **Education trumps training.** There is no such thing as an entirely isolated subject, as all fields of knowledge are connected. Training is narrowly-focused and task-oriented. Education is broad-based and principle-oriented. When preparing for a life-long technical career, education beats training every time.

- **Character matters.** Poor habits are more destructive than deficits of knowledge or skill. This is especially true in collective endeavors, where a team's ability to function depends on trust between its members. Simply put, no one wants an untrustworthy person on their team. An essential component of education then, is character development.

- **People learn to be responsible by bearing responsibility.** An irresponsible person is someone who has never *had* to be responsible for anything that mattered enough to them. Just as anyone can learn anything, anyone can become responsible if the personal cost of irresponsibility becomes high enough.

- **What gets measured, gets done.** Accurate and relevant assessment of learning is key to ensuring all students learn. Therefore, it is imperative to measure what matters.

- **Failure is nothing to fear.** Every human being fails, and fails in multiple ways at multiple times. Eventual success only happens when we don't stop trying.

B.2 Proven strategies for instructors

- Assume every student is capable of learning anything they desire given the proper conditions. Treat them as capable adults by granting real responsibility and avoiding artificial incentives such as merit or demerit points.
- Create a consistent culture of high expectations across the entire program of study. Demonstrate and encourage patience, persistence, and a healthy sense of self-skepticism. Anticipate and de-stigmatize error. Teach respect for the capabilities of others as well as respect for one's own fallibility.
- Replace lecture with "inverted" instruction, where students first encounter new concepts through reading and then spend class time in Socratic dialogue with the instructor exploring those concepts and solving problems individually. There is a world of difference between observing someone solve a problem versus actually solving a problem yourself, and so the point of this form of instruction is to place students in a position where they *cannot* passively observe.
- Require students to read extensively, write about what they learn, and dialogue with you and their peers to sharpen their understanding. Apply Francis Bacon's advice that "reading maketh a full man; conference a ready man; and writing an exact man". These are complementary activities helping students expand their confidence and abilities.
- Use artificial intelligence (AI) to challenge student understanding rather than merely provide information. Find productive ways for AI to critique students' clarity of thought and of expression, for example by employing AI as a Socratic-style interlocutor or as a reviewer of students' journals. Properly applied, AI has the ability to expand student access to critical review well outside the bounds of their instructor's reach.
- Build frequent and rapid feedback into the learning process so that students know at all times how well they are learning, to identify problems early and fix them before they grow. Model the intellectual habit of self-assessing and self-correcting your own understanding (i.e. a cognitive *feedback loop*), encouraging students to do the same.
- Use "mastery" as the standard for every assessment, which means the exam or experiment or project must be done with 100% competence in order to pass. Provide students with multiple opportunity for re-tries (different versions of the assessment every time).
- Require students to devise their own hypotheses and procedures on all experiments, so that the process is truly a scientific one. Have students assess their proposed experimental procedures for risk and devise mitigations for those risks. Let nothing be pre-designed about students' experiments other than a stated task (i.e. what principle the experiment shall test) at the start and a set of demonstrable knowledge and skill objectives at the end.
- Have students build as much of their lab equipment as possible: building power sources, building test assemblies[1], and building complete working systems (no kits!). In order to provide

[1] In the program I teach, every student builds their own "Development Board" consisting of a metal chassis with DIN rail, terminal blocks, and an AC-DC power supply of their own making which functions as a portable lab environment they can use at school as well as take home.

this same "ground-up" experience for every new student, this means either previous students take their creations with them, or the systems get disassembled in preparation for the new students, or the systems grow and evolve with each new student group.

- Incorporate external accountability for you and for your students, continuously improving the curriculum and your instructional methods based on proven results. Have students regularly network with active professionals through participation in advisory committee meetings, service projects, tours, jobshadows, internships, etc. Practical suggestions include requiring students to design and build projects for external clients (e.g. community groups, businesses, different departments within the institution), and also requiring students attend all technical advisory committee meetings and to dialogue with the industry representatives at those meetings.
- Repeatedly explore difficult-to-learn concepts across multiple courses, so that students have multiple opportunities to build their understanding.
- Relate all new concepts, whenever possible, to previous concepts and to relevant physical laws. Challenge each and every student, every day, to *reason* from concept to concept and to explain the logical connections between. Challenge students to verify their conclusions by multiple approaches (e.g. double-checking their work using different methods). Ask *"Why?"* often.
- Maintain detailed records on each student's performance and share these records privately with them. These records should include academic performance as well as professionally relevant behavioral tendencies.
- Hold mandatory "check-in" meetings between all program faculty and each new student during their first term. Offer these to all other students as an option, except for any students continuing to manifest unprofessional behaviors, poor academic performance, or who have some other need for a face-to-face meeting with faculty.
- Address problems while they are small, before they grow larger. This is equally true for tutoring technical concepts as it is for helping students build professional habits.
- Build rigorous quality control into the curriculum to ensure every student masters every important concept, and that the mastery is retained over time. This includes (1) review questions added to every exam to re-assess knowledge taught in previous terms, (2) cumulative exams at the end of every term to re-assess all important concepts back to the very beginning of the program, and (3) review assessments in practical (hands-on) coursework to ensure critically-important skills were indeed taught and are still retained. What you will find by doing this is that it actually boosts retention of students by ensuring that important knowledge gets taught and is retained over long spans of time. In the absence of such quality control, student learning and retention tends to be spotty and this contributes to drop-out and failure rates later in their education.
- Finally, *never rush learning.* Education is not a race. Give your students ample time to digest complex ideas, as you continually remind yourself of just how long it took you to achieve mastery! Long-term retention and the consistently correct application of concepts are always the result of *focused effort over long periods of time* which means there are no shortcuts to learning.

B.3 Proven strategies for students

The single most important piece of advice I have for any student of any subject is to take responsibility for your own development in all areas of life including mental development. Expecting others in your life to entirely guide your own development is a recipe for disappointment. This is just as true for students enrolled in formal learning institutions as it is for auto-didacts pursuing learning entirely on their own. Learning to think in new ways is key to being able to gainfully use information, to make informed decisions about your life, and to best serve those you care about. With this in mind, I offer the following advice to students:

- **Approach all learning as valuable**. No matter what course you take, no matter who you learn from, no matter the subject, there is something useful in every learning experience. If you don't see the value of every new experience, you are not looking closely enough!
- **Continually challenge yourself**. Let other people take shortcuts and find easy answers to easy problems. The purpose of education is to stretch your mind, in order to shape it into a more powerful tool. This doesn't come by taking the path of least resistance. An excellent analogy for an empowering education is productive physical exercise: becoming stronger, more flexible, and more persistent only comes through intense personal effort.
- **Master the use of language**. This includes reading extensively, writing every day, listening closely, and speaking articulately. To a great extent language channels and empowers thought, so the better you are at wielding language the better you will be at grasping abstract concepts and articulating them not only for your benefit but for others as well.
- **Do not limit yourself to the resources given to you**. Read books that are not on the reading list. Run experiments that aren't assigned to you. Form study groups outside of class. Take an entrepreneurial approach to your own education, as though it were a business you were building for your future benefit.
- **Express and share what you learn**. Take every opportunity to teach what you have learned to others, as this will not only help them but will also strengthen your own understanding[2].
- Realize that **no one can give you understanding**, just as no one can give you physical fitness. These both must be *built*.
- **Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable**. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[3] effort, and never give up! That concepts don't immediately come to you is not a sign of something wrong, but rather of something right: that you have found a worthy challenge!

[2] On a personal note, I was surprised to learn just how much my own understanding of electronics and related subjects was strengthened by becoming a teacher. When you are tasked every day with helping other people grasp complex topics, it catalyzes your own learning by giving you powerful incentives to study, to articulate your thoughts, and to reflect deeply on the process of learning.

[3] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

B.4 Design of these learning modules

"The unexamined circuit is not worth energizing" – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits. Every effort has been made to embed the following instructional and assessment philosophies within:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[4] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[5] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[6] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[4] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[5] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[6] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

Appendix E

References

Eastwood, Brian, "The 10 Most Popular Programming Languages to Learn in 2021", Northeastern University Graduate Programs, `http://www.northeastern.edu/graduate/blog/most-popular-programming-languages/`, 18 June 2020.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Prentice Hall, New Jersey, 1978.

Matloff, Norman, "Introduction to the Unix Curses Library", 4 January 2007.

Padala, Pradeep, "NCURSES Programming HOWTO" version 1.9, 20 June 2005.

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

4 October 2026 – added another Challenge question to the "Decibel calculator program" Conceptual Reasoning question, and added instructor notes to another.

2-7 September 2026 – added some instructor notes, and also incorporated suggestions on `for()` loops per Wyatt Gleason. Also added more bullet points to the Introduction chapter.

31 August 2026 – added some instructor notes.

26 August 2026 – minor edits to source code shown in various Questions, making sure all floating-point initializations have decimal points.

24 August 2026 – elaborated on the disinctions between compiling and interpreting.

24 July 2026 – added a Diagnostic Reasoning question illustrating memory usage for local (automatic) variables within functions, as well as `static` variables.

21 July 2026 – added a Tutorial section on the system() function call.

3 February 2026 – minor edit to the "Pointers" section of the Tutorial chapter.

20-21 January 2026 – corrected an error in the "Bitwise operators" section of the Tutorial chapter, regarding the default number of bits in an `int` variable for the bitwise demonstration program. I said that these integer variables are 16-bit by default when I should have said 32-bit. Also added a Challenge question to the "XOR cryptography" Quantitative Reasoning question.

6-8 October 2025 – added reference to the `const` modifier to the "Data types" section of the Tutorial. Also added explanations of what the `#define` preprocessor directive does.

2 September 2025 – minor additions to the "Pointers" and "Arrays" sections of the Tutorial chapter, as well as to the Introduction chapter.

19 August 2025 – corrected a repeated typographical error in the bitwise instructions section of the Tutorial where `BIT0` was defined as `0b00000000` instead of `0b00000001` as it should have been.

18 August 2025 – minor edits made to the Introductory chapter.

28 June 2025 – added another program example to to the "Loops" section of the Tutorial demonstrating how the condition-check portion of a `for` loop need not be the direct value of its test variable, and also some exposition on defensive programming in the form of initializing variable to safe values rather than assuming they will begin at zero.

27 June 2025 – minor edits to some C source code formatting to make the use of whitespace more consistent in variable assigment (e.g. `Vsrc = 12.0` rather than `Vsrc=12.0`) for some of the example code shown in questions. Also added some more instructor notes. Also, corrected a rather embarrassing error in the description of the "Using C to plot a sine wave" Quantitative Reasoning question, where it began by describing a completely different question!

14 January 2025 – added some new instructor notes.

3 September 2024 – minor edits to the "Arrays" section of the Tutorial chapter, and additions made to instructor notes.

26-27 August 2024 – fixed a pair of errors where I showed the math library linking option before the source-file name rather than after as it should have been. Also added a new Quantitative Reasoning question where complex numbers are used in C to analyze a series-parallel RLC circuit powered by an AC voltage source. Also fixed some errors in some of the instructor notes.

20-21 August 2024 – added a bullet-item to the Introduction chapter on bitwise operations. Also corrected some omitted lines on a program's output in the "Using the C math library" Tutorial section. Also, minor edits made to instructor notes.

18 August 2024 – divided the Introduction chapter into three sections, one with recommendations for students, one showing challenging concepts related to the module's topics, and one with recommendations for instructors with sample learning outcomes and measures.

31 May to 2 June 2024 – added a new Tutorial section on Structures, and also elaborated on data types.

21 February 2024 – added the `printf()`/`scanf()` identifier for double-precision floating-point variables, `%lf`, as well as added `double` as a common data type, to those respective sections of the Tutorial chapter.

31 January 2024 – fixed a typo in the "Arrays" section of the Tutorial chapter, where I stated "four" but really meant to say "three". Fixed another typo in the same section courtesy of Joe Archer, where I said "initialized" but should have said "declared".

26 January 2024 – minor edits to the Tutorial section on pointers, made for clarity.

16-17 January 2024 – added more questions to the Introduction chapter. Also edited some C code to specify numerical entry in hex.

5 October 2023 – edited some source-code listings to use the word "percent" rather than an escaped-% symbol which causes problems with some compilers.

29 August 2023 – corrected a typo in a C program comment that said "value numbers" rather than "valid numbers". Also commented on the use of `float` test variables within `for` loops.

23 August 2023 – added the `%g` formatting option to that section of the Tutorial. Also added a reference to the INFINITY constant contained in the C math library.

27 July 2023 – added a Quantitative Reasoning question showing a C program analyzing a multi-source DC circuit.

22-27 June 2023 – corrected an error in the Tutorial's section on pointers where I used the word "initialize" instead of "declare" with regard to two variables in a program. Also elaborated on the use of pointers for hardware-specific addressing such as memory-mapped I/O, and corrected a few mis-spellings throughout the Tutorial.

10-11 May 2023 – added some instructor notes as well as made edits to many of the Quantitative Reasoning questions asking more programming-centric queries instead of only (or mostly) being circuit-centric.

30 May 2022 – added explanation and programming example showing the use of multi-dimensional arrays.

21 April 2022 – minor additions to the Introduction, and corrected a minor source code discrepancy in the "Bitwise operators" section of the Tutorial where a line should have read "bit 5 is HIGH!" rather than "bit 5 is high". Also added another sentence of text to the "Bitwise operators" section identifying other input conditions that would satisfy the "bit 5 is high" condition in the subsection "Testing bit states".

12 March 2022 – added more sub-questions to the "Resonant frequency calculator program" Conceptual Reasoning question (requesting students trace the flow of program execution), as well as added instructor notes. Ditto for the "Writing your first C program" Conceptual Reasoning question and some of the Quantitative Reasoning questions, and for the same reason: challenging students to identify the order in which lines of code execute. Also added some content to the Introduction chapter.

23-24 January 2022 – added a new Tutorial section introducing graphics programming in C. Also fixed syntax errors in some of the conditional example programs, thanks to Jacob Stormes for identifying these.

20 January 2022 – minor additions to the Introduction, as well as some new Challenge questions.

22 December 2021 – moved the Case Tutorial section on complex numbers to its own dedicated section in the regular Tutorial.

13-16 December 2021 – wrote a new Tutorial section on arrays.

6 December 2021 – added a Quantitative Reasoning question on writing a byte-shifting program.

28 November - 1 December 2021 – added some content to the Tutorial section on pointers.

19 November 2021 – added some commentary on using bitwise-XOR for encryption and decryption.

9-12 November 2021 – wrote a new Tutorial section on bitwise instructions.

7-8 November 2021 – began writing a new section on logic functions in C, and also added a list of formatting specifiers to the section on standard output using printf().

6 October 2021 – added a new Challenge question.

2-3 September 2021 – added a new Tutorial section on using the GDB debugger.

26 August to 1 September 2021 – added content to the "Functions" section of the Tutorial, and also added new Case Tutorial sections. Also added instructor notes and made minor error corrections.

25 August 2021 – added instructor notes to certain questions.

1-21 August 2021 – added content to the Tutorial.

31 July 2021 – added chapters and sections to the Tutorial.

4 July 2021 – document first created.

Index

Accumulator, 156
Adding quantities to a qualitative problem, 236
Address resolution, 160
Addressing, relative, 163
AND function, 74
Annotating diagrams, 235
API, 147
Applications programming interface, 147
Argument, 50, 51, 67, 68, 75, 77, 101, 105, 109
Array, multi-dimensional, 122
Arrow operator, 128
Assembly code, 156
Assembly language, 21

Boolean algebra, 75, 79

C99 standard, 75
Calling a subroutine, 162
Casting, 47, 113
Checking for exceptions, 236
Checking your work, 236
Clock, 161
Code, computer, 245
Comment, 160
Compiler, 21
Curses, 146

De-reference, 108, 111, 113
Defensive programming, 73
Delay, 161
DeMorgan's Theorem, 82
Dickey, Thomas, 146
Dimensional analysis, 235
Directive, 159
Disassembler, 163
Dot operator, 123, 128

Edwards, Tim, 246
Endianness, 157
Executable code, 21

Fetch/execute cycle, 161
Format specifier, printf(), 37
Function, 109

Graph values to solve a problem, 236
Greenleaf, Cynthia, 171

Hex dump, 158
How to teach with these modules, 243
Hwang, Andrew D., 247

I/O, memory-mapped, 112
Identify given data, 235
Identify relevant principles, 235
Intermediate results, 235
Interpreter, 21
Interrupt, 165
Interrupt service routine, 165
Inverted instruction, 243
iptables utility, 136
ISR, 165

Jump instruction, 157

kill utility, 136
Knuth, Donald, 246

Label, 160
Lamport, Leslie, 246
Limiting cases, 236
Little-endian, 157
Logical function, 74

Machine code, 156
Machine language, 21

Memory-mapped I/O, 112
Metacognition, 176
Moolenaar, Bram, 245
MOS 6502 microprocessor, 156
Murphy, Lynn, 171

Ncurses, 146
NOT function, 74

Opcode, 157
Open-source, 245
OpenGL, 145
Operand, 157
OR function, 74

Pass by reference, 109
Pass by value, 109
ping utility, 136
Pointer de-reference, 108, 111, 113
Popping from the stack, 163, 165
Portability, 145
Precedence, 79
Preprocessor directive, 54, 95
Problem-solving: annotate diagrams, 235
Problem-solving: check for exceptions, 236
Problem-solving: checking work, 236
Problem-solving: dimensional analysis, 235
Problem-solving: graph values, 236
Problem-solving: identify given data, 235
Problem-solving: identify relevant principles, 235
Problem-solving: interpret intermediate results, 235
Problem-solving: limiting cases, 236
Problem-solving: qualitative to quantitative, 236
Problem-solving: quantitative to qualitative, 236
Problem-solving: reductio ad absurdum, 236
Problem-solving: simplify the system, 235
Problem-solving: thought experiment, 235
Problem-solving: track units of measurement, 235
Problem-solving: visually represent the system, 235
Problem-solving: work in reverse, 236
Product-of-sums, 82
Program counter, 162
Programming, defensive, 73
Pushing to the stack, 163, 165

Qualitatively approaching a quantitative problem, 236

Read-only variable, 43
Reading Apprenticeship, 171
Reductio ad absurdum, 236, 242, 243
Register, 156
Relative addressing, 163
Return from subroutine, 162

Scalable Vector Graphics, 152
Schoenbach, Ruth, 171
Scientific method, 176
sftp utility, 136
Shift register, 165
shutdown utility, 136
Simplifying a system, 235
Socrates, 242
Socratic dialogue, 243
Source code, 21
SPICE, 171
Stack, 163, 165
Stallman, Richard, 245
Standard I/O, 145
Status register, 166, 167
Subroutine, 162
Sum-of-products, 81
SVG image, 152

Thought experiment, 235
Time delay, 161
Torvalds, Linus, 245
Truth table, 74

Unconditional jump, 157
Units of measurement, 235

Visualizing a system, 235

Work in reverse to solve a problem, 236
WYSIWYG, 245, 246

XML, 152