Computer Science CS1

Donald Alan Retzlaff
Principal Lecturer

Philip Hamilton Sweany
Associate Professor

Table of Contents

Section 3 - Conditionals

Section 4 - Additional C Operators

Section 5 - Loops

Section 6 - Strings

Section 7 - User-Defined C Functions

Section 8 - Arrays

Section 9 - Pointers

Appendix

Introduction

Computer Science CS1 is a course designed for computer science majors. The class, and these notes, discusses most of the common aspects of C and programming in general, tailored for beginners who have little or no background in computer programming.

The major themes of the course include:

- Developing students' expertise in the C programming language
- Preparing students to build C software in a Linux environment
- Introducing students to the algorithmic thinking that pervades computer disciplines
- Helping students develop an accurate and useful abstract model of how computers work

These notes directly address students' ability to program in C and are designed as a supplement to your instructor's in-class lectures. They will be presented in the classroom utilizing the online version of these pages, making it easy for you to follow along, without having to tediously attempt to copy the programming examples into your personal notes. These pages are formatted to make it easy for you to add your own notes to the pages as the material is presented in class.

Learning how to program is a skill that requires practice. Lots of practice! Running the program examples that are in the notes is a good start, but it rarely is enough for most students. One problem with most beginners is they're not sure what to try, or they don't know how to come up with programming exercises to write. This is why, after many of the program examples shown in this text, there are supplemental programming problems, all of them similar in nature to the corresponding exercise in the notes. These are designed to give you the opportunity to practice to your heart's content.

Authors

Don Retzlaff is a Principal Lecturer for the Computer Science and Engineering Department at the University of North Texas. He has taught programming for over forty years, thirty-three of which have been at UNT. He received his Masters Degree in Computer Science from North Texas State University (later renamed to UNT) in 1978, and has taught a variety of software development undergraduate courses at UNT since that time, including Basic, Assembly, Pascal, C, C++, Java, and PHP. He also is the coordinator and primary instructor for the department's senior-level Software Development courses where students develop and maintain large-scale web-based applications.

For most of the courses Don teaches, he has developed his own custom set of course notes, utilizing various programming features and the web to enhance the student's learning experience.

Don enjoys traveling with his wife Elisa, writing, programming, and spending time at home with their four cats.

Phil Sweany is an Associate Professor in the Computer Science and Engineering Department at the University of North Texas. He has been a computer science faculty member for 20 years, with the last 8 at UNT. Phil received both M.S. and Ph.D degrees from Colorado State University in 1986 and 1992 respectively, and has taught a variety of Computer Science courses both at Michigan Technological University and UNT. His research focuses on compiler optimization for heterogeneous multiprocessors on a chip.

Unlike Don, Phil enjoys staying at home where he and his wife Margaret are dutiful staff members responsible for the care and feeding of three cats. So, given the authors' choices in pets, I hope we'll be forgiven for interjecting a few "catty remarks" from time to time both in these notes and in class.

Instructional Objectives

As an aide to both students and instructors, we are including a list of instructional objectives that we anticipate that each student will meet during the term. The concept of instructional objectives and their relevance to this course will be discussed in class early in the term, but in general, please remember that exam questions will come directly from these objectives and an excellent way to prepare for exam(s) is to make sure that you can meet each objective "covered" on an exam. So, without further ado, here is a list of objectives that students will meet.

1. Find an electronic version of the web page for Fall 2011's CSCE 1030.
2. Draw a diagram that shows the logical relationships among the following components of a "typical" computer.
    a. CPU
    b. Arithmetic-Logic Unit
    c. Main Memory
    d. Secondary Storage
    e. I/O Devices
3. Given a diagram as described in the previous objective describe how the various logical components of the computer work together to execute a program.
4. Describe the fetch-execute cycle.
5. When shown a program, determine whether that program is a machine-language program, an assembly-language program or a high-level language program.
6. Describe the purpose of the C Standard Library.
7. Describe Moore's law.
8. Describe how each of the following pieces of software helps to convert a C program into a program ready to execute.
    a. Preprocessor
    b. Compiler
    c. Assembler
    d. Linker
    e. Loader
9. Use readable and consistent style in writing C programs.
10. Write complete C programs that use assignment statements, printf and scanf.
11. Write complete C programs that use simple IF statements.
12. Given an assignment statement, and input values for each right-hand side operand, determine what value will be assigned during program execution.
13. Describe, both pictorially and in words, how a C assignment statement changes the state (memory) of the computer during execution.
14. Show a precedence table for the following C binary operators.
    a. +, -, *, /, %
    b. ==, !=, <, >, >=, <=
15. Use C's floating point and/or mixed-mode arithmetic to produce "more meaningful" answers to

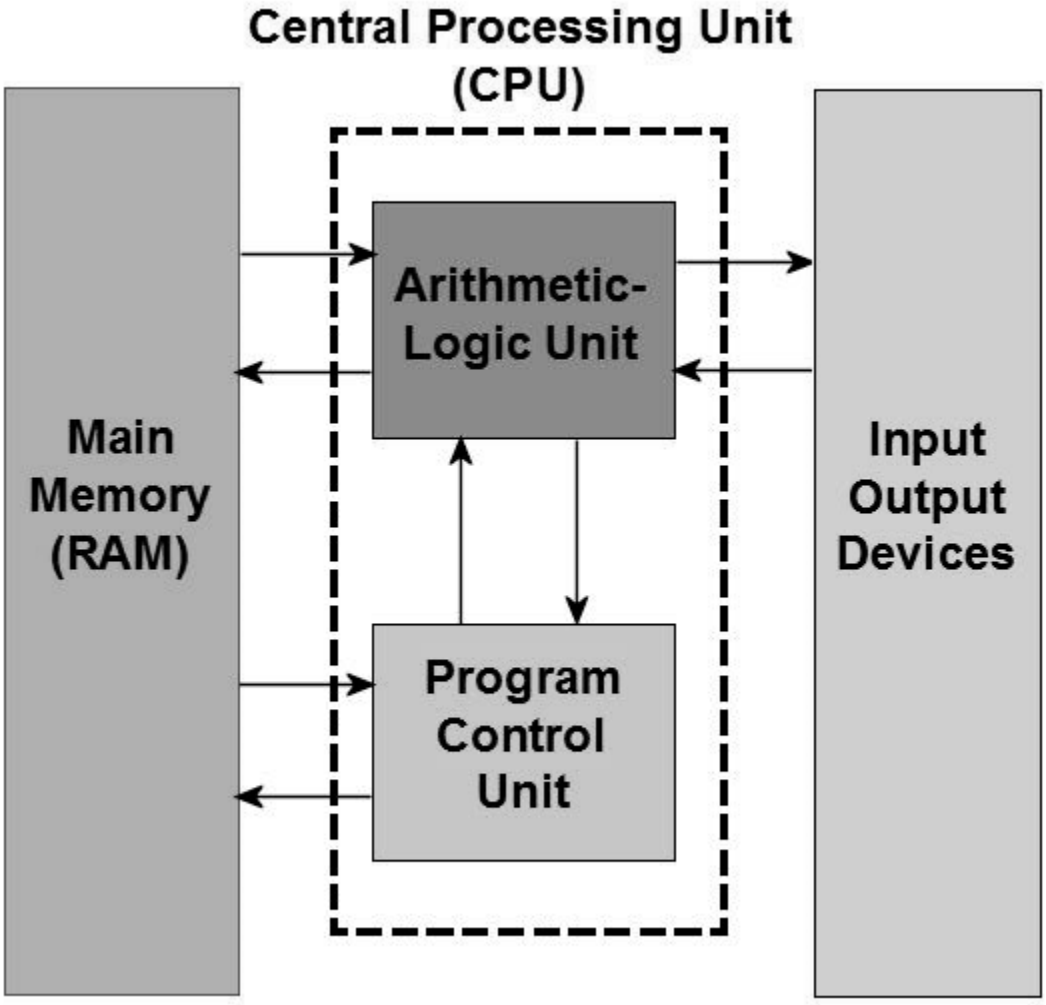
integer problems.

16. Develop algorithms through the process of stepwise refinement.
17. Use pseudocode in design of algorithms.
18. Write complete C programs that use WHILE statements.
19. Write complete C programs that use C's rich set of assignment operators.
20. Write complete C programs that use C's increment and decrement operators.
21. Write complete C programs that use IF-THEN-ELSE statements.
22. Write complete C programs that use logical operators and complex conditional statements.
23. Write complete C programs that use FOR statements.
24. Write complete C programs that use DO ... WHILE statements.
25. Write complete C programs that use SWITCH statements.
26. Write complete C programs that use BREAK statements.
27. Write complete C programs that use CONTINUE statements.
28. Write complete C programs that use the following subset of C's rich set of unary, binary, and ternary operators:
    a. (), [], ->, sizeof
    b. ++, --
    c. binary operators +, -, *, /, %, <<, >>
    d. unary -, &, *, ~
    e. &, *
    f. relational operators <=, <, >, >=, ==, !=
    g. logical operators &&, ||, !
    h. assignment operators =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=
    i. bitwise operators &, |, ^
29. Describe how C's ternary operator, ?:, works and why some (including Dr. Sweany) suggest not to use it.
30. Describe how C's bit manipulation operators (shift, bitwise AND, OR, and NOT) can be used to alter the operands upon which they are used.
31. Given an operator precedence table and an arbitrary C expression, list the order in which the operators will be evaluated.
32. Design significant software using stepwise refinement and modular design techniques.
33. Write complete C programs that include calls to C-supplied functions.
34. Write complete C programs that include user-defined function definitions other than main.
35. Write complete C programs that include user-defined function definitions in more than one .c file.
36. Write complete C programs that include user-defined header (.h) files.
37. Write complete C programs that make significant use of the random function to generate and use random integer data.
38. Write complete C programs that use recursion as a design and programming tool.
39. Describe C's conventions for supporting function calls, including:
    a. run-time stack
    b. activation records
    c. call by value
    d. call by reference

40. Describe C's storage classes, namely:
    a. auto
    b. static
    c. register
    d. extern
41. Given a multi-function, multi-file program, identify the scope of variable(s) within that program.
42. Write declaration statements for both single and multi-dimension arrays in C.
43. Use #define constants to specify both array sizes in declarations and loop bounds in code accessing arrays.
44. Use C's [ ] operator to access both single and multi-dimension arrays in C.
45. Use loops and loop indices in manipulating arrays in C programs.
46. Initialize arrays either in loops or in initializer lists within a declaration statement.
47. Declare both single and multi-dimension arrays in function definitions (as parameters) and in function prototypes.
48. Pass arrays as arguments to C functions.
49. Use array data structures in design and C implementation of algorithms, including but not limited to:
    a. printing histrogram(s) of frequency counts
    b. bubble sort
    c. merge sort
    d. binary search
    e. reading (scanf) and writing (printf) strings as character arrays
    f. manipulating characters within an array
    g. printing tables
50. Debug programs with array access out-of-bounds errors.
51. Declare a C enumerated type.
52. Use a C enumerated type in a C program.
53. Declare a C variable to be a pointer.
54. Dereference C pointer variables.
55. Use C's pointers to "simulate" pass-by-reference of scalar variables.
56. Describe how X[i] can possibly be interpreted the same as i[X] in C.
57. Given a C expression that uses pointer arithmetic define the value of the expression in symbolic terms using the address of the pointer variable.
58. Describe how C typically allows one-dimensional arrays and pointers to be used interchangeably.
59. Use malloc to dynamically allocate enough space to hold a data structure (such as an array) in C.
60. Use gdb to find, and fix, bugs in programs using arrays and pointers.

Section 1 - Introduction

Logical Relationships among the Components of a "Typical" Computer



MEMORY - The computer's memory is a relatively small storage area that can be accessed quickly. Data from the input sources are stored here, as well as the instructions for the programs that are run. Memory maintains the output data from the program until it is placed on the output devices. The information in memory is usually volatile, meaning it is typically lost when the computer's power is turned off. Memory is also commonly referred to as primary memory, or also RAM, or Random Access Memory.

ARITHMETIC-LOGIC UNIT (ALU) - This section of the computer performs the various mathematical calculations required by the computer programs, including addition, subtraction, multiplication, and division. It also contains circuits that are designed to perform decisions or comparisons, such as comparing two values in memory and determining if they are they're equal, or one is numerically greater than the other.

PROGRAM CONTROL UNIT - This section controls the operation of the computer, managing the communication between the other components. The Arithmetic-Logic Unit and Program Control Unit are often combined to form what is called the **CENTRAL PROCESSING UNIT, or CPU**.

INPUT UNIT - This is where the programs and data originate. Most users enter data and commands via a keyboard and mouse. Other devices are also used, such as disk drives, CD/DVD drives, and now USB thumb-drives.

OUTPUT UNIT - Output devices are used to access information from the computer's primary memory. Most output today is displayed on screens, printed on paper, or played on audio speakers. Output can also be written to external drives (disk, CD/DVD, or thumb-drives), or even remote locations on the Internet.

SECONDARY STORAGE - Secondary storage is usually long-term, high-capacity storage media, that can be disconnected from the computer itself and still maintain its data. Secondary storage devices are said to be persistent, in that they keep their data even without power being applied to them. The cost of secondary storage is much less than primary memory.

Instruction Fetch-Execute Cycle

What actually happens when an instruction gets executed in a computer? The process involves a key series of events. What's interesting is this sequence happens for each instruction that is executed:

- At the beginning of each cycle the CPU presents the value of the program counter on the address bus.
- The CPU then fetches the instruction from primary memory via the data bus into the instruction register (a register is one of a small set of very-high-speed memory locations).
- From the instruction register the data forming the instruction is decoded and passed to the control unit which sends a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction, such as reading values from registers, passing them to the ALU to add them together, and writing the result back to a register.
- The program counter is then incremented to address the next instruction and the cycle is repeated.

Moore's Law

Moore's Law describes a long-term trend in the history of computing hardware. The law is named after Intel co-founder Gordon E. Moore, who described the trend in his 1965 paper:

> The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.

This trend has continued for more than half a century and is expected to continue until 2015 or 2020 or later.

The capabilities of many digital electronic devices are strongly linked to Moore's law: processing speed, memory capacity, storage technology, and even the number and size of pixels in digital cameras.

The capabilities of our personal computers today far exceed the capabilities of the multi-million dollar mainframe computers of the past. Every day we see new innovations in technology that amaze even the most ardent technophile, providing computer scientists with a marvelous playground in which to work.

Telling the Difference Between Programming Languages

Programmers write instructions for the computer in a variety of programming languages. Some use a notation very close to the computer's own internal binary language, while others are high-level, supporting a notation similar to the real-world problems being solved, and therefore require intermediate translation steps.

Machine language is the native or "natural" language understood by a specific computer architecture. Machine languages are machine dependent, meaning that a specific machine language can only be used on one type of computer. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) to represent instructions and data values.

```
01010100000101011111101010101010100
```

Assembly languages are symbolic representations of the machine language, converting instructions into symbols that are meaningful to the programmer, each one representing individual instructions in the computer architecture.

```
LOAD value
ADD TOTAL
STOR value
DW 200
```

Translator programs called **Assemblers** are used to convert the assembly language instructions to their binary, machine language equivalent.

Although assembly language is much easier to work with than machine language, it still is tedious when complicated programs are required. To speed up the programming process, **High-level languages** were developed where single statements can perform many of the assembly language instructions, as well as provide a structure that is similar to the types of operations (such as mathematical expressions) that are being described in the computer program. The following is an example:

```
Average = TotalGrades / NumGrades;
```

The Programming Steps

1. Define the problem to solve.

2. Devise an **algorithm** (logical steps) to solve the problem.
   Example: Filling your Car with Gas

3. Write a program in a **programming language** that describes the algorithm for the computer. You use a **text editor** (in the Unix environment, common editors are **`joe`**, **`emacs`**, or **`vi`**) to type-in the program, and save this in a file normally referred to as the ***source file***). Details of the use of these editors will be discussed in your labs.

4. Then translate the program into machine language: Compile (translate) with a software application called a **compiler**.

5. **Link** (combine) the various modules that make up the program with a **linking program** to create an *executable image* of the program.

6. Finally, you **Execute** (run) the program, testing the program to see if it solves the problem. The executable image of your program is loaded into memory for execution using a **loader** program.

7. Repeat the steps above until the program does what you want it to.

Please note that all of these steps and the commands you will use to do them will be discussed in detail in your lab.

Common Cautions about Programming

- Programming in any programming language requires extreme attention to detail; the syntax rules of programming languages are very specific and cannot be ignored; if you do, you will not be able to compile your programs and frustration will be your only reward.

- Although you will be able to continuously reuse your skills and techniques that you've learned from writing other programs, new programs will always require something that you no doubt have never done before.

- Programming requires a thorough understanding of problem solving skills and a logical mind; if you can't define the steps necessary to solve problems or think logically, programming will be mere smoke and mirrors to you.

Programming Rules and Meaning

Syntax

The grammatical rules related to a programming langauge.

English: **`subject predicate.`**

C: **`variable = expr op expr;`**

Semantics

The meaning of the statements in a programming language.

```
Result = A - B;
```

"*Subtract the value of variable B from variable A, and assign the difference to the variable Result.*"

History of C

C is a general-purpose programming language designed by Dennis Ritchie of AT&T Bell Laboratories.

It was first implemented on a **PDP-11** (a mini-computer system) in 1972.

Ken Thompson (another researcher at AT&T) had used C's predecessor (a language called B) to do initial development of the UNIX operating system.

C's original claim to fame was that UNIX was written in C, and therefore UNIX was the first operating system that was written completely in a high-level programming language. To that point in time, operating systems had been written in Assembly Language, a programming language that is very close to machine language).

In 1980, **ANSI** (the American National Standards Institute) defined "ANSI C" or "Standard C", which standardized the language, making it so implementations of the language by different programming language developers would all support the same language definition.

This course will utilize a UNIX-based version of the C compiler called **`gcc`**.

Overview of the C Programming Language

C is a **Structured Language** that supports the three basic **programming constructs**:

- simple sequence
- if-then-else (conditional test)
- loop (repeating instructions)

C supports these structured language constructs with extensions.

Every program written can be written with these three, standard language constructs.

C is a **Stream-oriented** language:

- Operations can continue automatically to a new line.
- More than one command can appear on a line.
- Commands are separated by semicolons.

```
double num1,
       totalSalary,
       shares;

x = y +
  numValues - 3
  * totalCount;

printf ("The value is %d\n",x * 12);

x = 5; shares = 7.25;
```

Commands in the C language are **Case-sensitive** (lowercase letters and capital letters are considered to be different):

`if` is different than `IF`

Keywords (also sometimes referred to as **Reserved Words**), are words that have specific meaning in the language and can only be used for that one purpose.

```
auto      double    int       struct
break     else      long      switch
case      enum      register  typedef
char      extern    return    union
const     float     short     unsigned
continue  for       signed    void
default   goto      sizeof    volatile
do        if        static    while
```

Others keywords may be included in specific compiler implementations.

There is no reason to memorize what the keywords are, as the compiler will immediately let you know you are using a keyword incorrectly.

C Supports "Subroutines"

Subroutines are named modules of code that are designed to perform a specific operation in your program. When we write our programs, we can group statements and give them a name so we can easily refer to them again elsewhere in the program. This is an extremely powerful and useful feature in programming, saving many hours of repetitious coding.

Subroutine Call Sequence

When a subroutine is "***called***," control is transferred to the subroutine and its instructions are then executed. When the subroutine is completed, control is automatically returned to the calling set of instructions and continues where it left off.

You reference a subroutine by specifying its name (note the parenthesis):

```
displayResults();
```

Values (commonly referred to as **parameters** or **arguments**) can be passed to subroutines to modify or alter what the subroutine does, or they modify the data that the subroutine operates on:

```
displayMessage ("Hi there!");

answer1 = sqrt(x);
answer2 = sqrt(y);
```

In C, subroutines are often referred to as **functions**.

C Comments

Comments (programmer remarks) allow the programmer to describe (in English terminology rather than C-code) the algorithms and approaches used in a program.

Comments should be included in the program liberally, but don't go overboard. The examples in these notes show reasonable uses of programmer comments.

The syntax of comments have two forms:

- A pair of delimiters:

  `/*` and `*/`

  are used to surround the comments. This form is commonly used for multiline comments.

- A single delimiter:

  `//`

  is placed before the comment. The rest of the current source line is all considered to be part of the comment.

Comment examples:

```
/* Anything inside these delimiter characters */

/* Comments can also be continued
       onto multiple lines, if you wish */

// Single line comments
```

Basic Structure of a C Program

A typical book consists of a series of sections:

- Title Page
- Author Acknowledgements
- Table of Contents
- Several Content Chapters
- Appendices
- Index

A C program also consists of a series of sections that must appear in a specific order:

- Introductory Comments that identify the program, its author, and the primary purpose of the program.
- "'Included files" (descriptions of other code modules that will be used by your program).
- Global variable definitions (values that are used throughout the entire program).
- Function (Subroutine) definitions.
- Main (primary) function definition (where the program will begin its execution):

```
int main() {

  Local variable definitions

  Instructions

  }
```

Example of a Simple C Program

01simple.c

```
/* C Program Example - Course: CSE 1030 - C Programming
   programmer: Don Retzlaff  donr@unt.edu

   C program to accept two numbers from user and
   add the numbers together.
*/

#include <stdio.h> // include system information

int main() {
  int num1, num2, sum;

  // display initial program identification
  printf ("Simple C Program - Don Retzlaff donr@unt.edu\n\n");

  // prompt the user for input
  printf ("Please enter first number: ");
  scanf ("%d",&num1);
  printf ("Please enter second number: ");
  scanf ("%d",&num2);

  sum = num1 + num2;
  printf ("The sum is %d\n",sum);

  return 0; // return control to Operating System
}
```

Possible modifications to this sample program that you can try as separate exercises:

- Change the text in the comment section and in the code section so the program includes your name and email address rather than your instructor's.
- Change the input values to different numbers and verify that the program continues to produce the proper output for the numbers entered.
- Change the variable names in the program to "value1," "value2," and "result" (remember to change all references to the variables in the program).
- Change one of the lines in the program so it no longer matches the syntax that we've discussed and compile the program; explain the syntax error message; put the program back to its original, compilable state, and verify that it will compile successfully and produce the proper output.
- Change the calculation in the program so it finds the difference between the two values, rather than the sum (remember to also change the output message so it indicates the result is the difference rather than the sum, and change the variable name from sum to difference).
- Change the calculation in the program so it finds the product (the multiplication of the two values), rather than the sum (remember to also change the output message so it indicates the result is the product rather than the sum, and change the variable name from sum to product).
- Change the program so it prompts the user to enter three values rather than two, and have the program find the sum of the three entered values.

Common C Characteristics Related to this Simple Example