1. Objectives
   1. Learn how to program a computer
   2. Learn semantics of C
      1. **Syntax**: rules of language.
      2. **Semantics**: meanings of symbols.
      3. Love and hate C
   3. Give ability to use C to solve real-world problems through computing
   4. Learn to be creative and resourceful
2. Computers
   1. First thing to remember: Computers are dumb creatures
      1. They only do exactly what you tell them to do
         1. They cannot figure out what you mean like humans do
   2. You must communicate in a way they understand what you want them to do.
      1. C: Terse, vaguely English-like language that is specific in what it states
   3. Languages with varying strengths and weaknesses
      1. Most popular
         1. C
            1. One of hardest
            2. One of most useful
            3. Provides a window into how computers work
         2. C++
         3. Java
         4. JavaScript
         5. Python
         6. HTML
      2. Pascal
      3. Fortran
      4. Lisp
      5. Prolog
      6. Many others
3. Instructions
   1. Tell the computer what to do
      1. Computers manipulate data to achieve an objective
   2. A *program* tells the computer what actions to perform…
      1. On exactly what data
      2. In what sequence
      3. What to do with results
   3. Program provides processor with instructions to execute, and the specific relevant data to be transformed.
      1. Data piece 🡪 Paint it red 🡪 Processor 🡪 Red data piece
   4. **Bits**: long sequences of simple elements that take on values of 0s and 1s.
      1. Make up…
         1. **Instructions**: indicate a pattern of specific action to be executed.
            1. Machine language
         2. **Data**: represent a number, symbol, memory location, pixel color, sound element, or other representation of data to be transformed.
      2. 0s and 1s are electrical voltage values at output of transistor
         1. Transistor outputs either…
            1. 0 voltage (0)
            2. Saturated maximum voltage (1)
         2. Value can be rapidly switched on and off by hardware as required by instructions.
            1. Helps execute programs quickly.
4. Computer Languages
   1. Humans think in terms of…
      1. Words
      2. Pictures
      3. Sounds
      4. Concepts
   2. …not in machine language (1s and 0s)
      1. This makes it difficult for humans to communicate with computers.
      2. Programs originally had to be written in 0s and 1s in the early days of computing.
   3. High-level programming languages
      1. Example: C
      2. Vaguely English-like “*statements*”
         1. Relatively easy to write by a programmer
      3. **Compilers**: translate statements into machine language instructions.
      4. **Statements**: instructions written in high-level languages.
      5. **Source code**: the program in the form of statements.
5. Modern Computer Programming
   1. **Computer programming**: process of putting together a sequence of statements that when compiled, linked, and executed, will cause the computer to act as desired and produce the desired result.
      1. Modern computing is complex
         1. Graphic programming
         2. Internet programming
         3. Writing programs that control devices in real time
      2. Fundamentally, these are simply sequences of instructions to a processor.
6. Some History about C
   1. 1972: C was developed by Dennis Ritchie at ATT Bell Labs.
      1. 2011: Dr. Ritchie passed away.
      2. C was built to work with the Unix operating system
         1. **Operating system**: overarching computer program that handles basic functionality for the computer.
      3. Derived from other languages
         1. “B”, probably derived from “A”
      4. Several low features not found on other high-level languages
         1. Pointers
         2. Direct bit manipulation
7. Stored-Program Digital Computer
   1. 1940s: older computers were…
      1. Large
      2. Special-purpose
      3. Hard-wired
      4. Could only perform tasks they were designed to do
   2. **Stored program computer**: allowed computers to have general purposes, and run any program so long as it followed the instructions defined for its processor, without changes to wiring.
      1. Major improvement
         1. Program could be stored in memory as data
         2. Hardware components could be made smaller
            1. Wiring no longer had to be manipulated
   3. Present day: computers are programmable
      1. **Random Access Memory (RAM)**: Instructions (program) and data are now stored in the same memory elements.
8. Components of the Modern Computer
   1. **Central Processing Unit (CPU)**: executes instructions.
      1. Modern computers have ≥1.
   2. **Main Memory (RAM)**: running programs and their data are stored in main memory.
      1. **Volatile**: wiped off when computer is turned off.
   3. **Graphics Processor (GPU)**: additional processor charged with processing graphics.
      1. Often used nowadays to assist CPUs with general purpose computing tasks
   4. **External Memory**: persistent, long-term memory.
      1. Examples
         1. Hard drive (HD)
         2. Solid-state drive (SSD)
         3. CD
         4. Flash drives
      2. **Non-volatile**: data persists after shutdown.
   5. **Registers**: very fast, hardware memory locations in which to place instructions about to be executed and relevant data.
      1. Volatile
   6. **Arithmetic Unit (ALU)**: additional processor for heavy “number crunching” computations.
      1. No longer common
         1. Integrated into CPU or GPU
   7. **Operating System (OS)**: performs basic functions related to operating computer, file manipulation, queuing instructions, etc.
      1. Examples
         1. Windows
         2. Mac OS
         3. Unix
      2. Allocates computer’s resources
         1. Memory
      3. Allows…
         1. Interface to choose what to run
         2. Moving files (data or programs) around in external memory
         3. Loading programs into main memory for execution
      4. Loaded into main memory by a primitive process “hardcoded” into the system.
   8. **Compiler**: translates high-level source code to machine language.
   9. **Application program**: programs intended by end-user on a computer.
9. Preliminaries
   1. How to program a computer
      1. Create instructions in C…
         1. For a compiler to convert into machine language code
         2. To be linked to other programs and/or libraries
         3. For the processor to execute and return the desired input
      2. Use Code::Blocks integrated development environment (IDE) to translate C into machine language and execute your program
10. Programming Cycle
    1. **Edit**: compose C statements that make up the program.
    2. **Compile** the C statements.
       1. If fail, go back to 1.
    3. **Link** the program to other libraries as appropriate.
       1. Done automatically
    4. **Execute** the “object code” program.
       1. **Object code**: instructions in machine language.
    5. **Obtain** results.
    6. **Examine** results for “mistakes”.
       1. If happy with results, end.
       2. Otherwise, go back to 1.
11. Editors and Compilers
    1. Any text editor can be used to write C
       1. Some editors > others
    2. Prefer to use an **Integrated Development Environment (IDE)**
       1. Code::Blocks
       2. Allows you to edit code easily, compile, and run.
       3. Provides debugging tools.
12. Basic C Programming Structure
    1. //Comment lines
    2. C Preprocessor
       1. Header files
       2. Macros
    3. Global data declarations
       1. If any
       2. None in Code::Blocks
    4. Function prototypes
       1. If any
       2. None in Code::Blocks
    5. **main()** function
       1. C/C++ statements
    6. Definition of other functions
       1. If any
       2. None in Code::Blocks
13. **Comments**: parts of the code written purely to describe things to other humans and not for the compiler to translate them.
    1. Comment indication tells computer not to compile them.
       1. Can be placed anywhere within program.
    2. Types
       1. //at the start of a line
       2. /\* ….. \*/ if inserted into active code
    3. Examples
       1. //John Smith
       2. //My first C program
    4. Good and frequent use makes it easier to understand programmer’s intent.
14. The C Preprocessor
    1. **Preprocessor directives**: special commands that indicate things to be done to the program prior to compilation.
       1. Include certain other files when compiling
       2. Replace some code by other code
       3. Denoted by…
          1. Statements beginning with #
15. Header Files
    1. Allow programmer to…
       1. Break up program across several files
          1. Modularity
          2. Reusability
       2. Include Standard C Library files that contain needed functions
    2. **#include** **directive**: tells preprocessor to include named files in compilation process.
       1. Part of linking process
       2. Causes a copy of the designated file to be included in place of the directive
          1. Shorthand for inserting code from a file in another location.
       3. Examples of header files
          1. **math.h**
          2. **stdio.h**
          3. **stdlib.h**
          4. **string.h**
       4. Format
          1. **#include “filename”**: preprocessor searches in the same directory as the file being compiled.
             1. Typically user-defined files
          2. **#include <filename>**: preprocessor searchers in predefined directories where standard C Library files are located.
             1. Normally used for standard library files, as the ones shown above.
    3. **#define directive**
       1. **Symbol replacement**: causes preprocessor to replace in the source code prior to compilation the symbol defined in the macro by another symbol.
          1. Example: #define A 29.75
             1. Pre-processor will revise source code and replace all instances of A with 29.75.
             2. Often used for constants.
             3. Case-sensitive
       2. **Macro replacement**: can replace code.
          1. Macros may be defined either with or without arguments.
             1. Replacement code text is put in place of the macro identifier.
          2. Arguments should be between parentheses.
             1. Parentheses should be used in the definition to not confuse compiler.
          3. Example: #define CIRCLE\_AREA (x) (PI \* (x) \* (x))
             1. When called:

area = CIRCLE\_AREA(4);

* + - * 1. Preprocessor replaces CIRCLE\_AREA(4) with:

(3.14159 \* (4) \* (4))

* + - 1. Scope of a macro or symbolic constant is…
         1. **Scope**: how long/where a statement is meaningful.
         2. Until the end of the file in which the directive resides.
         3. Until it is defined with #undef.
         4. Or, whichever comes first.

1. **The main() Function**: begins and ends the program.
   1. Format
      1. **int** = placed before the main() function as a declaration of an integer.
         1. Not strictly necessary
         2. Needs a **return(0)** statement at the end of the code.
      2. { = beginning of body
      3. } = close of main function
      4. () = where any arguments to the function are entered
         1. Most of the time has no arguments.
   2. There is a way to do so when called from a command line Entire C program can be written with only the main() function.
      1. At least one has to be defined by the programmer.
         1. Limiting, as it may have to be a very large function.
         2. Many disadvantages
      2. Contains essence of the program.
2. Defining Functions
   1. Other functions can also be defined to be later called in the program to execute their designed computations.
      1. Basis of C/C++ programming
   2. Used to abstract long/complex computations into simple call
3. **C Statements**: ≥1 instructions that will later be compiled into machine language.
   1. Conclude with ;
   2. Could be grouped together as block of code by using {}
   3. Example
      1. result = a + b + c \* (d + e);
      2. Meaning: compute this expression, and set the result to the variable “result”
         1. Values for a, b, c, d, and e represent data and must be known.
4. **Bugs**: errors in programming.
   1. Happen often
   2. 3 basic types
      1. **Compilation errors**: mistakes that the compiler catches.
         1. Causes
            1. Not closing strings: missing the “”s.
            2. Not closing {}s properly
            3. Not closing ()s properly
            4. Forgetting ; at the end of each statement
            5. Forgetting & before the variable when calling scan(f)
            6. Using %f instead of %lf for doubles when using…

printf()

scanf()

* + - 1. Generally easy to detect and easy to fix.
         1. Compilation fails
    1. **Execution errors**: program compiles and runs initially, but terminates abnormally.
       1. Causes
          1. ÷0
          2. Infinite loops

Will go on forever, unless compiler stops it.

RAM may run out, causing crash

* + - * 1. Improperly terminated recursion
      1. Easy to detect but difficult to understand why.
         1. Must identify C statement(s) causing the crash.
    1. **Logic errors**: happen normally when the program compiles and terminates normally, but produces incorrect output.
       1. Hardest to detect
          1. No indication of something wrong
          2. Program validation and verification used to detect them
       2. Difficulty of fix is variable once the error is found.
       3. Example
          1. Variable takes on improper values.

1. Correcting Errors
   1. Errors will happen.
   2. Key: minimize errors.
   3. When they happen…
      1. Recognize them
      2. Correct them