

Introduction

July 16, 2021

1 Introduction

1.1 Topics

- computer science fundamentals
- problems, algorithms and programs
- programming languages
- C++ language and C++ program development steps
- setting up development and learning environments
- the first program and its anatomy
- errors and debugging

1.2 Computer science (CS) fundamentals

- the core is a disciplined ability to be logical and creative in a pragmatic way to solve problems in varieties of disciplines
- CS is a newer discipline that burrows from Mathematics, Engineering, and Natural Science
- like mathematicians computer scientist use formal languages to denote ideas (esp. computation)
- like engineers, they design things, assemble components into systems and evaluate tradeoffs among alternatives
- like scientists, they observe the behavior of complex systems, form hypothesis, and test predictions
- **the single most important skill for a computer scientist is problem-solving mostly writing computer programs**
- the goal of this course is to teach you how to think like a computer scientist
- computer scientists primary job revolves around problems, algorithms and programs

1.3 Problems, Algorithms and Programs

1.3.1 Problem

- we deal with and solve a lot of problems in every walk of lives
- problem is a question raised for inquiry that someone needs to answer or find solution to
- computer scientists typically deal with computational problems
- can be as simple as:

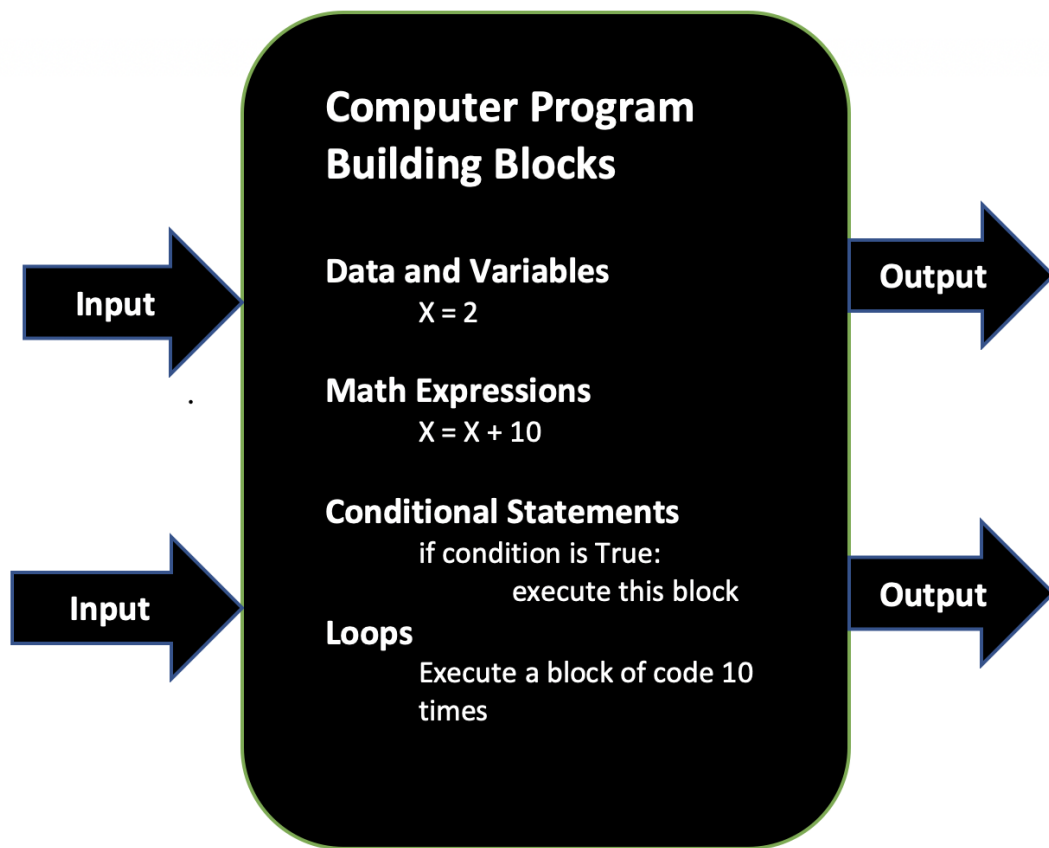
- what is the sum of 9 and 999?
- or can be as complicated as:
 - what is the shortest path from San Francisco, California to New York City, New York?
- one must understand the problem, analyze the requirements, constraints and assumptions in order to correctly solve the problem

1.3.2 Algorithm

- once the problem is formulated and well analyzed, computer scientists work on algorithm
- step by step process/task to solve a given problem
 - like a recipe for a food menu
- typically written in human language or pseudo-code (in between)
- e.g. problem: How can your martian friend buy grocery on earth?
- you should be able to help solve this problem given you live on earth and shopped groceries many times
 - in other words, you're the domain expert
- algorithm steps:
 1. Make a shopping list
 2. Drive to a grocery store
 3. Park your car
 4. Find items in the list
 5. Checkout
 6. Load grocery
 7. Drive home
- there's a lot of details missing from these steps
 - it's a good start and can be refined by drilling each step further down

1.3.3 Program

- once the algorithm steps are finalized, programmers can convert them into computer instructions
- sequence of instructions that specifies how to perform a computation using computers
 - computation can be mathematical (solving system of equations), symbolic computation (searching and replacing text in a document, scientific simulations), etc.
- the instructions (or commands or statements) look different in different programming languages, but the basic fundamental concepts are the same
- some fundamental concepts that make up a computer program regardless of the language are:
 1. data/values and variables
 2. input
 3. output
 4. math
 5. conditional (logical) execution
 6. repetition



1.3.4 data and variables

- program works with data (called values) which must be stored in computer memory; variables are names given to memory locations to store values

1.3.5 input

- get data from keyboard, a file, or some device

1.3.6 output

- display data/answer on screen, or save it to file or to a device

1.3.7 math

- basic mathematical operations such as addition, subtraction, multiplication, etc.

1.3.8 conditionals

- test for certain conditions or logics and execute appropriate sequence of statements

1.3.9 loops

- perform some action repeatedly, usually with some variation every time

1.4 Programming languages

- programming language is a formal language used to create computer program
- there are dozens of programming languages

1.4.1 Types of programming languages

1.4.2 High-level languages

- languages that are designed to be programmer friendly hiding all the details
 - C++, Java, C, FORTRAN, Python, PHP, JavaScript, Rust, etc.
- advantages:
 - simpler; easier to learn and write
 - shorter and easier to read
 - programs are portable; can run in different machines with a few or no modifications
- disadvantages:
 - translation to machine code can take some time
 - slower to run if the translation is not optimal

1.4.3 Low-level languages

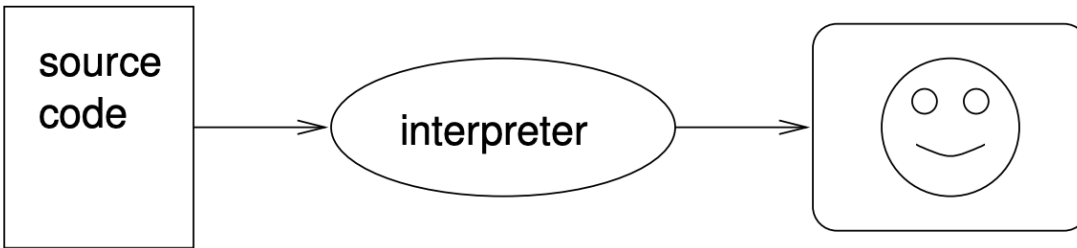
- machine language e.g., Assembly language
- loosely speaking: computers can only execute programs written in low-level languages
- programs written in a high-level languages must be translated before they can run
- advantage:
 - programs run faster
- disadvantages:
 - harder to learn and write code (need to know very low level details about computers)
 - programs are not portable; usually need to rewrite for each kind of machine architecture

1.5 Ways to translate high level programs

- there are two ways to translate high level programs: **interpreting** and **compiling**

1.5.1 interpreting

- an interpreter reads a high-level program and does what it says
- it translates the program line-by-line alternately reading lines and carrying out commands
- Python, PHP, JavaScript are interpreted languages

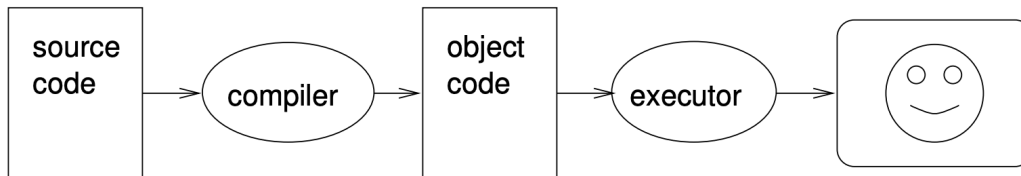


The interpreter
reads the
source code...

... and the result
appears on
the screen.

1.5.2 compiling

- a compiler reads a high-level program and translates it all at once into byte code before executing any of the commands
- compilers check for syntax/grammers of languages
- the byte code or binary program must be then loaded into memory to execute
- C++, C, Rust, Java, FORTRAN are compiled programming languages



The compiler
reads the
source code...

... and generates
object code.

You execute the
program (one way
or another)...

... and the result
appears on
the screen.

1.6 C++ Programming language

- C++ is one of the most popular general pupurpose programming languages - see [tiobe index](#)
- high level, compiled language
- extension of C programming language
 - same syntax; burrows all C libraries and supports class (object oriented programming, OOP)
- you can use all the C libraries and features in C++
- designed for system programming and embedded, resource-constrained software and large systems with performance, efficiency, and flexibility
- see Wikipedia entry for history and other details: - <https://en.wikipedia.org/wiki/C%2B%2B>
- official C++ reference site <https://en.cppreference.com/w/>

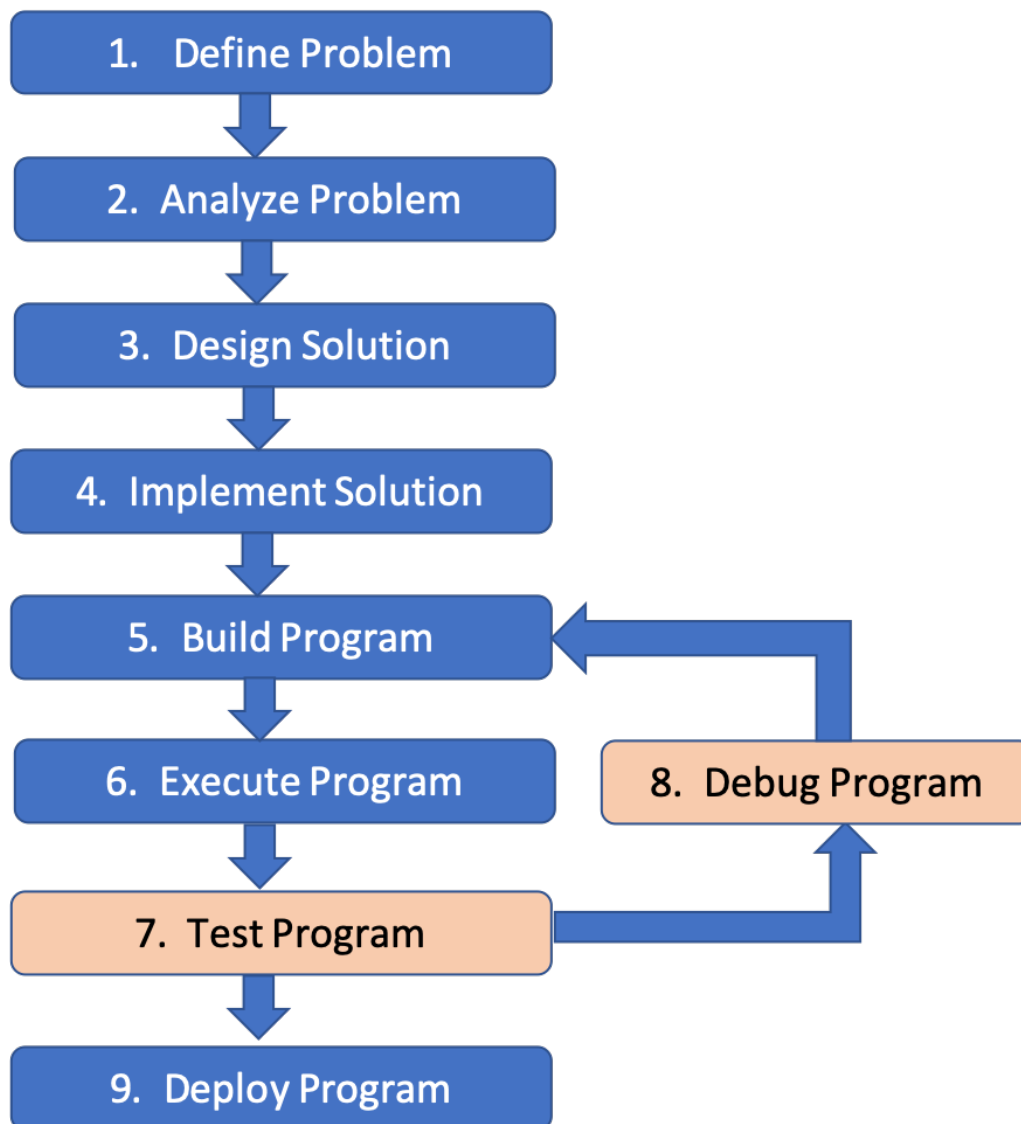
1.7 Problem solving using C++

- C++ is one of many tools computer scientists use to solve problems
- learning and being proficient using your tools are important
- ultimate goal is to learn problem solving like computer scientists
 - writing code, though important, is a small part of problem solving
 - programmers spend about 20% of their time developing code
- program design and development requires many steps
 - a major part of software engineering process

1.7.1 C++ development steps

- a simple high-level approach to developing C++ programs is depicted in the following figure

C++ Development Approach



Step 1: Define problem

- figure out what problem you are trying to solve
- it may be just an idea or a fully researched problem statement
 1. write a program to find the average temperatures in the USA over the last decade
 - write a program that checks if a given string is a palindrome
 - write a program that finds the shortest path from New York City to San Francisco
 - design and develop a system for Mars rover

Step 2: Analyze problem

- really understand the scope and all the parameters of the problem
- gather all the requirements, outline any assumptions, input and output constraints, etc.
 - let's say you want to solve the temperature problem #1 defined in step 1
 - how'd you get the temperatures data? would you include all 50 states?
 - what if there's no temperature data for some states? are you going to average out the states and then find the average of the average?
 - are there any outliers, how'd you handle those?

Step 3: Design solution

- determine "how" you'd actually solve the problem
- many ways to solve a problem; look into tradeoffs e.g. efficiency vs cost, etc.
 - often simple and straight forward solutions are better ones
- break the problem into smaller modules or sub problems
 - write algorithm steps for each sub problem
 - modular solution is easier to update, expand, and reuse without affecting other parts
- design mockups; draw system design
 - explain how various modules and components interact with each other
 - helps address any assumptions made or limitations

Step 4: Implement solution

- write the program using a programming language
 - use C++ in this course
- programmers spend only about 20% of their time writing code
- need a computer with a text editor or an Integrated Development Environment (IDE)
 - depends on the system: Windows, Linux, Mac, etc.
- a good code editor or IDE typically provides
 - way to organize project with multiple files and resources
 - syntax highlighting, color coding, line numbers, easy way to compile, run and debug code

Step 5: Build program

- this is typically a two-step process:
 1. compile c++ code in object or byte code
 - a project/program may contain many c++ files and header files (.cpp, .cc, .h file extensions)

- each C++ source file gets converted into object file (typically have .o extension)
- 2. link object files and libraries
 - program called linker combines or links together all the object files and C++ standard library and any other libraries used into one single executable program
- modern compilers (e.g. g++) can do both the steps at once
- **Makefile** is a better way to build C/C++ programs
 - a bash like script that simplifies a lot of step for building programs over and again
 - learn about makefile from these tutorials:
 - * <https://makefiletutorial.com/>
 - * <https://www.cs.bu.edu/teaching/cpp/writing-makefiles/>

Step 6, 7, and 8: Execute, Test, and Debug Program

- you must execute or run the program to test it
- a program called loader loads the executable into main memory RAM (Random Access Memory)
 - CPU (Central Processing Unit) does the actual computation
- testing ensures you're getting right results under all the assumptions
- programmers may spend a lot of time testing their own programs or others'
- counter intuitively, you try to break your own program!
- if bug or error exists, you need to pin point it and correct it
 - build the program again repeating from step 5 as many times as required
- two common ways to test your program for correctness; learn both in this course!
 1. manually run and feed input data and compare the results with the expected answers
 - write test cases and test your program automatically using code

Step 9: Deploy program

- deliver or deploy your program in production environment
 - given your program meets all the requirements, passes rigorous testing, etc.

1.8 The first program

- traditionally, "hello world" is the first program one writes to learn coding in any given language

```
[1]: /*=====
    Hello World program
    Author: Ram Basnet
    Date: June 24, 2020
    Copyright: MIT License

    The program prints "Hello World!" on the console
    =====*/

    // include required libraries/header files
    #include <iostream>
```



```
// one main function is always required in a C++ program
int main() // main entry to the program
{
    // output Hello World!
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

1.9 Structure of a C++ program

- C++ program may constitute one or many text files (typically header and source) files
- each C++ file contains various C++ statements, instructions and codes
- C++ source file typically has these extensions: filename.cpp or filename.cc
 - avoid file and folder names with spaces
- C++ program must have one file with the **main()** function
 - **int main()** is the main entry of the program
 - computer starts executing instructions top to bottom starting from **main()**
- C++ file typically contains:
 - **program description**
 - * brief information about the program and programmer, copyright info
 - * these are comments meant for programmers/readers
 - **libraries**
 - * include the libraries (header files) that are only required
 - * libraries provide built-in codes that programmers can use
 - * programmers don't have to write all the basic, details and common tasks
 - so, they can focus on solving problems
 - * libraries are mandatory for many common tasks such as input and output
 - **comments**
 - * comments are ignored by the C++ compiler
 - * comments are for programmers to explain the thought process, subtle code blocks
 - * it's best practice to write adequate notes as comments, esp. when learning
 - * makes it easy to read and understand code without actually having to run and decode the code
 - * *write code for others to read*
 - * `//` - double forward slashes is used for single line comment
 - * `/*` everything within are comments; used for multi-line comments `*/`
 - **instruction codes**
 - * tells computer what to do!
 - * code composed of keywords, identifiers, symbols, literal values, etc.
 - when put together following the language's grammar solves the problem
 - * block of codes appear within squiggly-braces `{ }`
 - * statements end with a semi-colon `(;)`
 - **white spaces**
 - * indentations, extra spaces and blank lines are typically ignored by the compiler unless necessary
 - * adequate white spaces are required as a best practice for readability of code

1.10 Setting up C++ development environment

- setting up a good development environment and getting familiar with it can make you an efficient learner and problem solver
- dev environment may depend on various factors: personal preference, available resources, project, etc.
- see [this note](#) for recommended C++ Dev Environment

1.10.1 Compiling C++ programs using g++ and Makefile

- modern integrated development environment (IDE) makes it very easy to compile, build and deploy C++ programs
- however, the abstraction and easyness they provide may hide some of the important steps beginner programmer should know
- the compilation process using Makefile is highly recommended for both beginners and the professionals in the field
- see the basic tutorial and examples on how to create and use Makefile [here](#)

1.10.2 Compile and run hello world program

- write a program that prints "Hello World!"
- compile and run the program on your system
- see the example `helloworld.cpp` program found in [demos/intro/](#)
- run the provided Makefile using a terminal assuming you're using g++ and Makefile
 - must cd into the folder that contains the Makefile

`make`

`make run`

`make clean`

1.10.3 Using Jupyter Notebook as Teaching and Learning Environment

- setting up Jupyter Notebook program is only required if you're teaching/learning C++ interactively
- these notes/pdfs are created using Jupyter Notebook program
- see [this page](#) for setting up and using Jupyter Notebook

1.11 Testing and debugging

- programs often contains many types of errors called bugs
- programmers spend majority of their time in testing their programs, finding bugs and getting rid of them
- the process of finding and correcting bugs is called debugging
- many IDEs provide a way to step through the code and examine memory as the program executes
- the key to finding and fixing bugs is testing
 - exhaustive testing makes sure program provides correct output for all sets of input

1.11.1 Types of errors

- there are three major types of bugs: syntax, run-time and semantic

Compile-time errors

- also called syntax errors or grammatical errors
- computer languages are formal languages with strict grammar to a semicolon
 - Natural languages (English, e.g.) are full of ambiguity, redundancy and literalness (idioms and metaphors)
- compiler parses the C++ code; provides a list of errors if any
- fails to compile a program to byte code if program has compiler error

Run-time errors

- also called run-time exceptions
- these errors appear while program is running
- can be handled to certain extent

Semantic errors

- also called logical errors
 - errors in thought process, may arise due to misunderstanding of problem, solution, language quirks
- program runs fine but gives wrong answer
 - e.g., adding instead of multiplying to solve an equation (e.g., $2 + 2$ is same as 2×2)
- can be identified and removed by doing adequate testing

1.12 Labs

1. Standard Output Lab

- write a C++ program that produces the following output on console
- use the partial solution provided in [labs/intro/main.cpp](#)
- observe and note how the special symbols such as single quote, double quotes and back slashes
- run the program as it is using the provided make file in the stdio folder
- complete the rest of the ASCII Art by fixing all the FIXMEs
- write #FIXED next to each fixme

```
| \_ / |      *****      ( \_ / )
/ @ @ \      *   ASCII Art   *   ( = ' . ' = )
( > 0 < )     *   Author: <Your Name>   *   ( " ) _ ( " )
>>x<<        *   CS Foundation Course   *
/ 0 \        *****
```

1.13 Exercises

1. Setup Development Environment

- setup your personal development environment

- download and install tools that are typically used by programmers: C++ Editor, C++ Compiler, git client, etc.
 - you can follow the steps provided [here](#)
2. Hello World
 - write a C++ program that prints “Hello World!” as an output to the console
 - see complete solution in [exercises/intro/helloworld/main.cpp](#)
 - write Makefile and use make to build the program
 - see Makefile solution here: [exercises/intro/helloworld/Makefile](#)
 3. ASCII Art
 - google images made using ASCII arts
 - print some ASCII arts, texts and pictures of your choice
 - can use ASCII Art generator: <http://patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20>
 4. The Game of Hangman
 - write a C++ program that prints various stages of the hangman game
 - game description: [https://en.wikipedia.org/wiki/Hangman_\(game\)](https://en.wikipedia.org/wiki/Hangman_(game))
 - produce the output seen in Example game section of the Wikipedia page
 - game will not have any logic to actually play, unless you know how to implement it!

1.13.1 Kattis problems

1. Hello World!
 - login and solve the Hello World! problem: <https://open.kattis.com/problems/hello>

1.14 Summary

- this chapter covered:
- the basics of Computer Science and programming
- different types of programming languages
- C++ basics, the first program and the basic structure of a C++ program
 - how to print data to standard output
- C++ editor and compiler
- exercises and sample solutions

[]: