C++ Programming (Malik, 2018)

January 3, 2024

Contents

1 An Overview of Computers and Programming Languages (ch 1)

In this lecture, you will:

- Learn about the history of computers
- Learn about different types of computers
- Explore hardware and software components of a computer system
- Learn about the language of a computer
- Learn about the evolution of programming languages
- Examine high-level programming languages
- Discover what a compiler is and what it does
- Examine a C++ program
- Explore how a C++ program is processed
- Learn what an algorithm is and explore problem-solving techniques
- Become aware of structured / object-oriented design methods
- Become aware of Standard C++, ANSI/ISO C++, C++11, and C++14

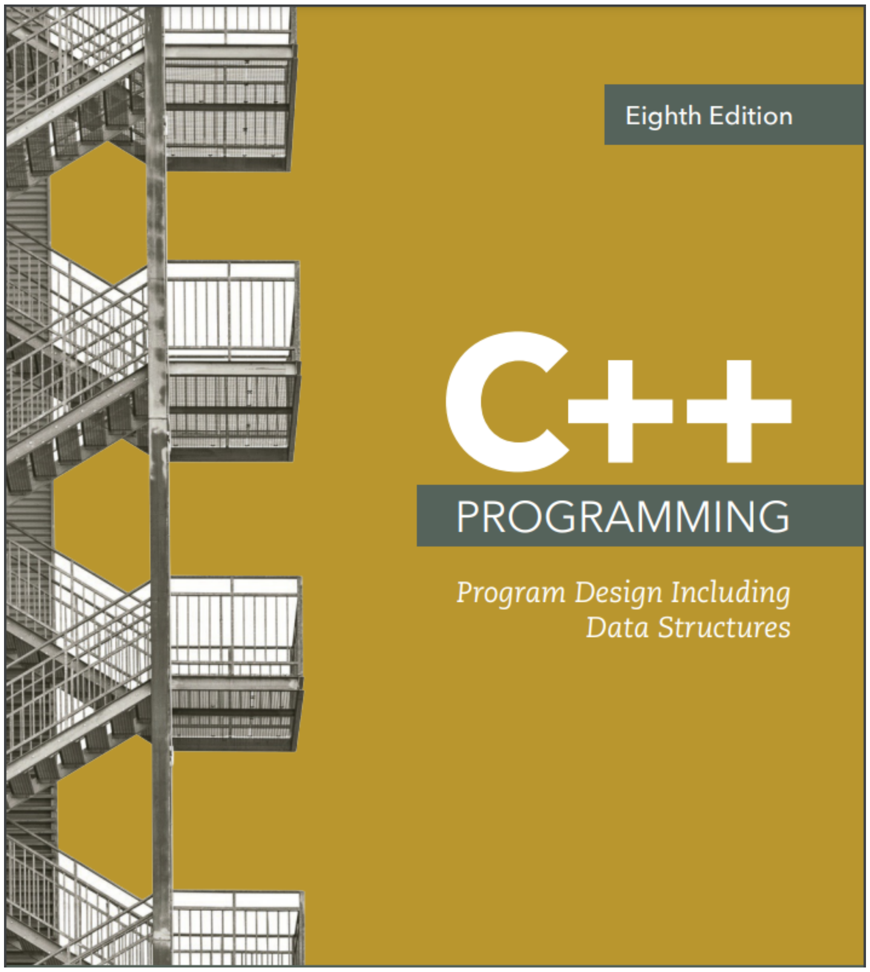


Figure 1: Cover for Malik, C++ Programming (8e), Cengage 2018.

There will be an online test with multiple choice, multiple answer and true/false questions. We will begin each session with a review of a selection of topics from the previous session. The lecture follows chapter 1 of the textbook by Malik (2018).

2 A Brief Overview of the History of Computers

The history of computing with special devices ("computers") goes back ca. 5,000 years. Most of the progress was made in the past 130 years.

TECHNOLOGY	YEAR
Abacus	3000 BC
Pascaline	1642 AD
Jacquard's Loom	1804 AD
Difference Engine	1822 AD
Analytical Engine	1837 AD
Tabulating Machine	1890 AD
Mark I	1944 AD
ENIAC	1945 AD
UNIVAC	1951 AD
Microprocessors	1971 AD
Personal Computers	1977 AD
Modern Computers	1990 AD

Interesting questions:

- What are punch cards for? Punch cards store data, instructions for the computer, or software.
- Do computers replace humans? This video (The World of Science, 2022) claims that "one ENIAC could replace 2400 humans". However, if the general statement is true depends on the task, of course.
- UNIVAC "predicted the outcome of the 1952 US Presidential election." How did it do that? What do you think happened? (Martin, 1993).
- Which was the first PC with a mouse and a graphical user interface? Apple's Macintosh in the early 1980s.

Watch: multi-part BBC documentary (BBC, 1991).

3 Elements of a Computer System: Hardware

Computing hardware:

- The **Central Processing Unit (CPU)** is the computer's "brain"
- The CPU carries out arithmetic and logical operations (ALU)
- Main or **Random Access Memory** (RAM) is directly connected to the CPU
- All programs must be loaded into RAM to be executed by the CPU
- RAM is organized in memory cells whose addresses are binary values
- Content in memory (numbers, letters etc.) are also binary values
- Secondary storage or **Non Volatile Memory** (NVM) is connected to RAM
- Speed: CPU is super fast, RAM is fast, and NVM is very slow.
- **Input devices** take in data and feed it to the computer
- Examples: keyboard, mouse, scanner, camera, NVM
- **Output devices** display results from the computer
- Examples: monitor, printer, NVM

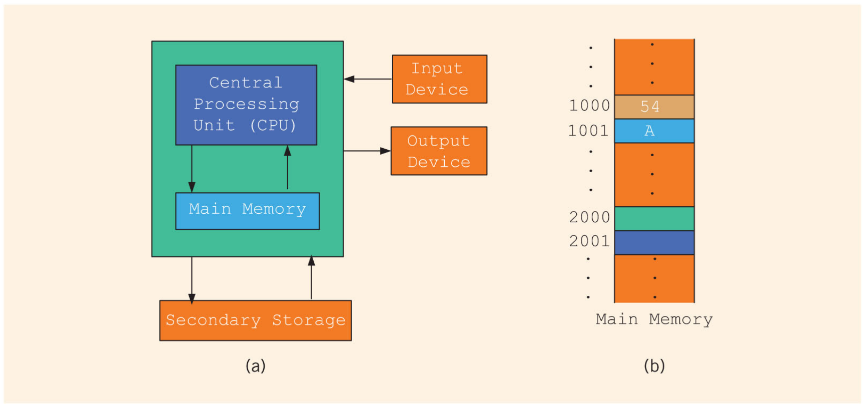
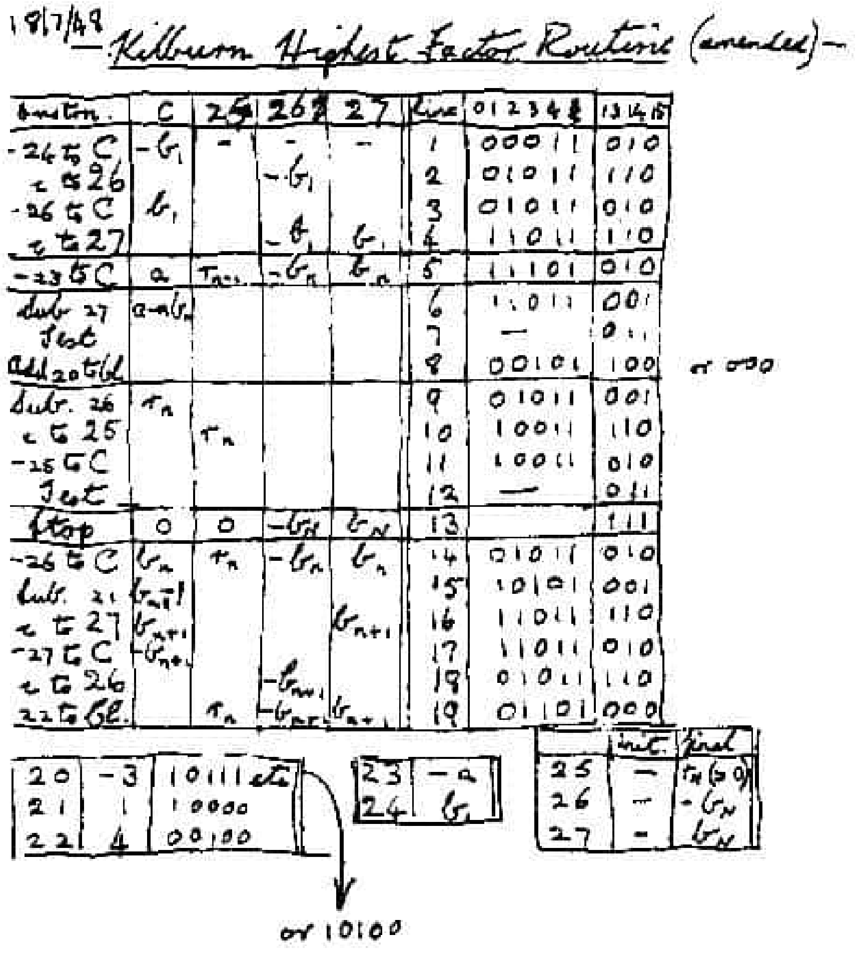


FIGURE 1-1 Hardware components of a computer and main memory

This kind of computer architecture is called von Neumann architecture after the Hungarian mathematician John von Neumann (1947).

4 Elements of a Computer System: Software



18/7/48 – Kilburn Highest Factor Routine (amended) –

Instrn.	C	25	26	27	Line	01234	13 14 15
-24 to C	$-b_1$	–	–	–	1	00011	010
c to 26			$-b_1$		2	01011	110
-26 to C	b_1				3	01011	010
c to 27			$-b_1$	b_1	4	11011	110
-23 to C	a	r_{n-1}	$-b_n$	b_n	5	11101	010
Subr 27	$a-nb_n$				6	11011	001
Test					7	—	011
Add 20 to Cl					8	00101	100
Subr. 26	r_n				9	01011	001
c to 25		r_n			10	10011	110
-25 to C					11	10011	010
Test					12	—	011
Stop	0	0	$-b_N$	b_N	13		111
-26 to C	b_n	r_n	$-b_n$	b_n	14	01011	010
Subr. 21	b_n-1				15	10101	001
c to 27	b_{n+1}			b_{n+1}	16	11011	110
-27 to C	$-b_{n+1}$				17	11011	010
c to 26			$-b_{n+1}$		18	01011	110
22 to Cl.		r_n	$-b_{n+1}$	b_{n+1}	19	01101	000

or 000

20	-3	10111 etc
21	1	10000
22	4	00100

or 10100

23	-a
24	b_1

	init.	final
25	–	r_N (=0)
26	–	$-b_N$
27	–	b_N

Figure 2: First stored program on the Manchester Baby (SSEM)

Image: The first stored program ever (on the "Manchester Baby", 21st June 1948) - to find the highest factor of any integer number a. This was done by trying every integer from a-1 until one was found that divided a without reminder.

5 What happens when you press 'A' on the keyboard?

Using a computer relies on the conversion of 'analog' (continuous) signals to digital (binary) signals.

1. Key Press (Mechanical Action):
   - You press the "A" key. This is a mechanical action.
   - The keypress pushes down a switch underneath the keycap.
2. Switch Closure (Analog Signal):
   - The closure of the switch under the "A" key creates an electrical circuit. This is still an analog action as it involves physical contact to close the circuit.
3. Key Scan (Digital Signal Processing):
   - The keyboard's controller (microcontroller) scans the keyboard's key matrix—a grid of circuits corresponding to different keys—to detect which circuit has been closed.
   - When it detects a closed circuit, it converts that detection into a digital signal. This is the conversion from analog to digital: the physical action has been translated into binary data.
4. Keycode Generation:
   - The keyboard's controller assigns a keycode to the "A" keypress. This keycode is a digital representation of the "A" key in binary form.
5. Data Transmission Over USB/Bluetooth (Digital Signal):
   - The digital keycode is sent via the USB or Bluetooth interface to the computer's central processing unit (CPU). This is done through a serial data transmission, where the binary data is sent bit by bit.
6. Interrupt and Processing by the CPU:
   - The CPU receives an interrupt signal from the keyboard device, indicating that it needs to process new input data.

- The CPU reads the keycode from the keyboard's device driver.

7. Software Interpretation:
    - The operating system (OS) interprets the keycode based on the current keyboard layout and other settings (like whether CAPS LOCK is on).
    - The OS determines that the keycode corresponds to the "A" character.
8. Application Response:
    - If a text editor or another application is open and in focus, the OS sends the character input to that application.
9. Character Rendering:
    - The application instructs the graphics card to render the "A" character on the screen.
    - The graphics card uses the font and rendering engine to create a visual representation of the character "A".
10. Display on Screen:
    - The graphics card sends the digital signal to the monitor.
    - The monitor's controller converts the digital signal into the proper format for the monitor's display technology (such as LCD or OLED).
    - The pixels on the screen light up to form the shape of the "A" character, which is now visible to you.

Throughout this process, the analog part is mostly in the initial mechanical action and electrical circuit closure. The digital signal processing dominates the latter stages, where binary data is processed and interpreted by various components of the computer's hardware and software.

6 The Language of a Computer

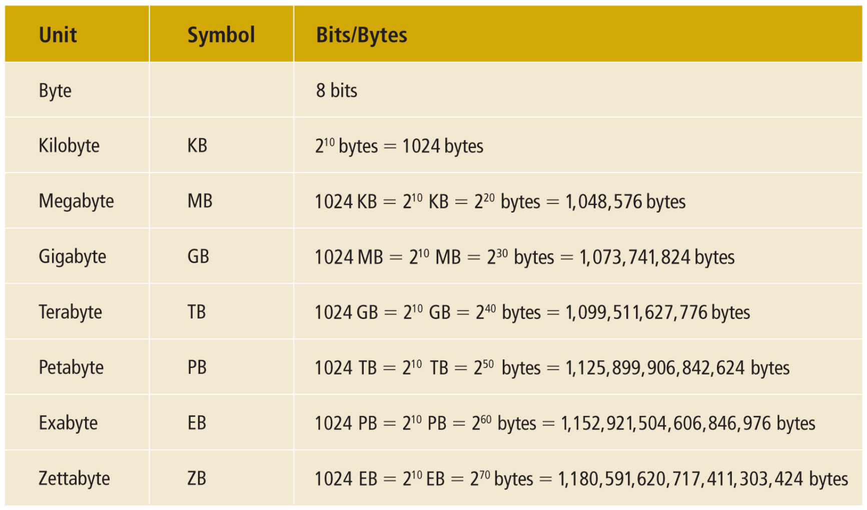
- Analog signals represent continously changing stuff like sound.
- Digital signals represent information with a sequence of 0s and 1s.

- 0 represents low, 1 high voltage.
- The computers native language, machine language, is binary code.

To measure the amount of information, we use binary units:

- A binary digit 0 or 1 is also called a 'bit' state.
- A byte ('word') is a sequence of eight bits.
- 2^{10} bytes = 1024 bytes is a kilobyte (KB)
- 2^{20} bytes = 1024 KB = 2048 bytes is a megabyte (MB)
- etc.

TABLE 1-1 Binary Units



Unit	Symbol	Bits/Bytes
Byte		8 bits
Kilobyte	KB	2^{10} bytes = 1024 bytes
Megabyte	MB	1024 KB = 2^{10} KB = 2^{20} bytes = 1,048,576 bytes
Gigabyte	GB	1024 MB = 2^{10} MB = 2^{30} bytes = 1,073,741,824 bytes
Terabyte	TB	1024 GB = 2^{10} GB = 2^{40} bytes = 1,099,511,627,776 bytes
Petabyte	PB	1024 TB = 2^{10} TB = 2^{50} bytes = 1,125,899,906,842,624 bytes
Exabyte	EB	1024 PB = 2^{10} PB = 2^{60} bytes = 1,152,921,504,606,846,976 bytes
Zettabyte	ZB	1024 EB = 2^{10} EB = 2^{70} bytes = 1,180,591,620,717,411,303,424 bytes

Some comparisons to keep these units in perspective:

- 1 KB: a paragraph of text.
- 1 MB: a short novel.
- 1 GB: Beethoven's 5th symphony.
- 1 TB: all X-rays in a large hospital.
- 1 PB: half the contents of all 123,000 US academic research libraries.
- 1 EB: a video call that started 237,823 years ago.

- 1 ZB: as much information as there are grains of sand on all the world's beaches.
- 1 Yottabyte is as much information as there are atoms in 7,000 human bodies, more than the amount of data currently (2023) stored in all of the worlds PCs, more than 2 billion. (Source).

7 Representation inside your computer

- Symbols on your keyboard are encoded in a unique binary code of seven bits. With that memory, you can encode $2^7 = 128$ symbols.
- The ASCII (American Standard Code for Information Interchange) data set consists of 128 characters numbered 0 to 127.
- The infamous 'A' is encoded as the binary number `1000001`. In decimal, i.e. base=10, this is 65, or the 66th ASCII value.

  In Python, you can use the `int` function for the conversion:

  ```
  A = int("1000001", base=2)
  print(A)
  ```

  ```
  65
  ```

  This also works the other way around using the `bin` function:

  ```
  # remove the prefix '0b' for binary numbers
  print(bin(65)[2:])
  ```

  ```
  1000001
  ```

  In C++, you'd use the `stoi` ('string-to-integer') function:

  ```
  std::cout << std::stoi("1000001",nullptr,2);
  ```

  ```
  65
  ```

So far, so good. But characters in a computer are represented in byte, or eight-bit sequences. To do this, the seven-bit ASCII code is prefixed by a 0.

There are other encoding schemes. For example, Emacs will ask you which scheme you want to use, and it will suggest Unicode, which consists of 10000000000000000 (in binary) characters - every character in Unicode is stored in 16 bits or 2 bytes.

8 The Evolution of Programming Languages

Programming languages have evolved over the past 100 years or so alongside the machines that use them to carry out simple jobs.

Here is one of many rankings of the top programming languages (IEEE, 2023):

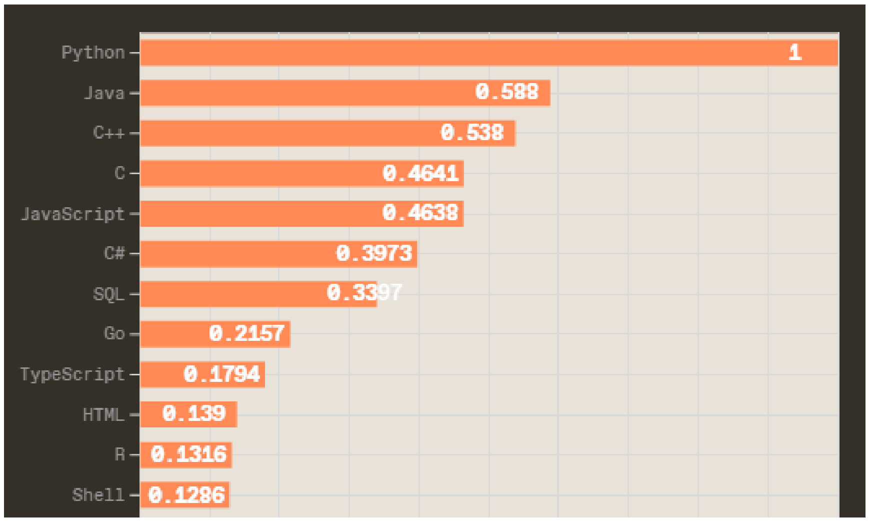


Figure 3: Top programming languages 2023

9 Machine, assembly, and higher languages

- Early computers (like the "Manchester Baby" of 1948) were programmed in machine language.
- An equation like `wages = rate * hours` for example implies the steps:
    1. Load the value of `rate`
    2. Multiply the values for `rate` and `hours`
    3. Store the result in `wages`
- In machine language, this command sequence might look like:

```
100100 010001
100110 010010
100010 010011
```

- An intermediate step is Assembly language, which encodes instructions as mnemonic, easy to remember words:

  ```
  LOAD rate
  MULT hours
  STOR wages
  ```

- The *assembler* program translates this instruction set into machine language before execution.
- Machine and assembly languages are not portable because their instructions must be specific to the computing architecture used.
- In different languages, the instruction sequence looks simpler:

  ```
  // in C and C++
  wages = rate * hours;
  // in Python
  waves = rate * hours
  // in R
  waves <- rate * hours
  ```

- Programs in these languages must be compiled into machine code to run, using a program called *compiler*.

10 Processing a C++ Program

We'll now learn how to

1. create a first program (without too much explanation)
2. compile the (readable) source code into (executable) machine code
3. run the machine code on the command line

Once we succeed, we'll analyze the process.

1. To create a program, open the command line terminal.
2. Type `nano hello.cpp`.
3. If the Nano editor opens, you're good otherwise install it:

- Download `nano` for Windows from sourceforge
- Unpack the Zip file ("Extract all")
- Go into the extracted folder (~'GNU-Nano$_{\mathrm{Win32}}$(static)')
- Inside the folder, run `nano`.

4. Inside the editor, enter your first C++ program:

```
#include <iostream>
int main(){
  std::cout << "My first C++ program." << std::endl;
  return 0;
}
```

```
My first C++ program.
```

5. Exit Nano and save the file with `CTRL-x`

6. Compile the file with the command `g++ -o test test.cpp`. If this does not work, you don't have an operational compiler and you need to download and install it from here.

7. Run the machine code with the command `test`. The line `My first C++ program.` shold be displayed.

8. Check the existing programs with the command `DIR *test*`: there should be two programs.

9. Close the shell with the command `exit` or, if that is not recognized, by closing the windows with the mouse.

Analysis:

1. The command line terminal is a program, too, called "shell" or `Command Line Interface` (CLI), or specifically `bash(1)`. The other way of interacting with the OS is a graphical user interface (GUI).

2. The `nano` editor is a simple `text editor` without syntax highlighting. The Emacs editor that use here, has it.

3. `Sourceforge` is one of many software repos[itories]. In class, we use the most popular of these, `GitHub`, where you can get all course materials. The `Extract all` routine, carried out inside the `Windows Explorer` program uses another program called `zip`. You can also extract packaged or compressed content manually without a GUI.

4. The program you entered is called `source code`. It has three parts:

    - A `preprocessor` directive `#include`
    - A `main` program, and
    - A command inside the `main` program: to print a `string`, an ordered sequence of characters, to the screen using `cout` and `endl`. These two commands are contained in a `namespace` (a place for commands) called `std` (for "standard").

5. When you save the file, it has the file extension `.cpp`. The C++ compiler will only accept files with that ending.

6. The compiler command itself is `g++` which stands for GNU C++ compiler. The `-o test` is a `flag` with an argument that names the compilation result `test`.

7. `test` is an executable file. On Windows, its full name is `test.exe` and you can run it by simply entering its name.

8. The `DIR` command is a command line terminal command in Windows that lists file directories. `*test*` returns all files that have `test` in the name of the file. The `*` is a so-called wildcard character.

9. The shell has many commands like `DIR` (Windows) or `exit`, or `cd` to change directory to manipulate files and navigate around. The Windows shell is an amputated version of `bash`, which is a Linux program. There is also a so-called "PowerShell" on Windows, but it's a nightmare, and you don't have to know about it unless someone (like your boss) forces you to.

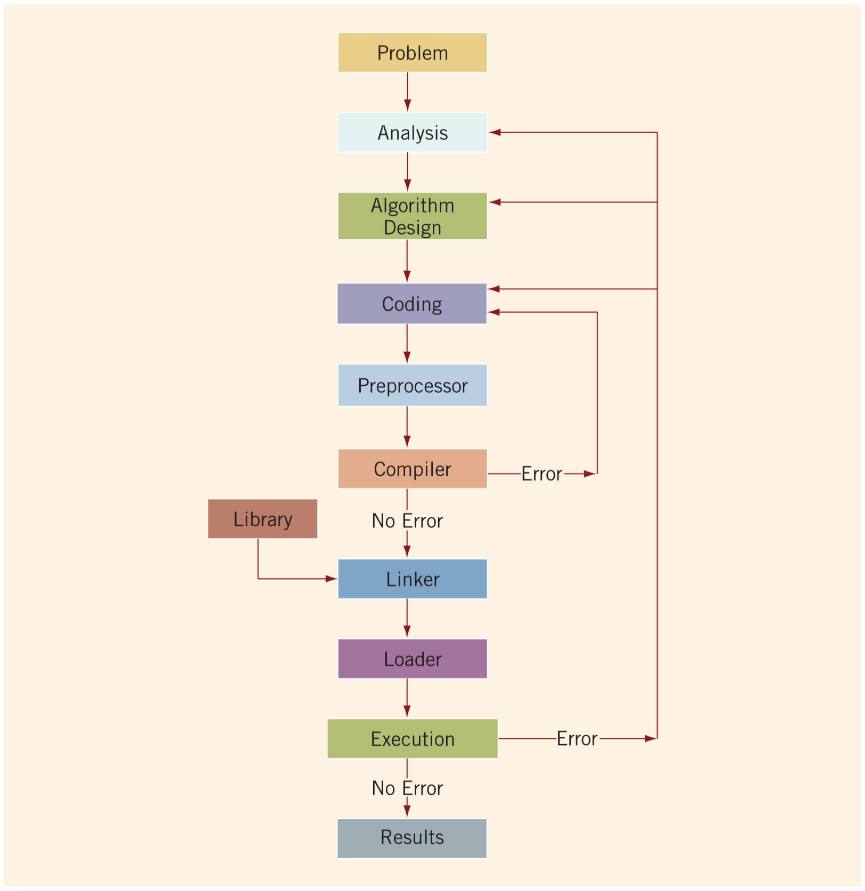


FIGURE 1-3 Problem analysis–coding–execution cycle

The figure shows what really went down when you ran the compiler:

- after editing the source program with an `editor` (1)
- the `preprocessor` includes libraries like `iostream` in source (2)
- the `compiler` checks for Syntax errors producing object code (3)
- the `linker` links the libraries to the object code (4)
- the `loader` loads the executable into main memory for execution (5)
- the `shell` runs the executable program (6)

11 Integrated Development Environments (IDEs)

Some people will tell you that "programmers" only need to be concerned with the editor and the source code development, which is done inside an integrated development environment (IDE).

The IDE, they say, takes care of the rest. It is a grave mistake to believe this. The "user friendliness" of the IDE comes at the expense of transparency. All IDEs except Emacs, the IDE we'll be using later in the course, are not (easily) extensible and customizable.

The reason why disregard for the coding development infrastructure is detrimental to your career and your mastery is that there are no "programmers". All professional coding is done as part of software development and involves intimate knowledge of many parts (and jobs).

There are different professional tools for every part of the software development lifecycle, which involves more than one party - like customers, suppliers, other developers, etc.

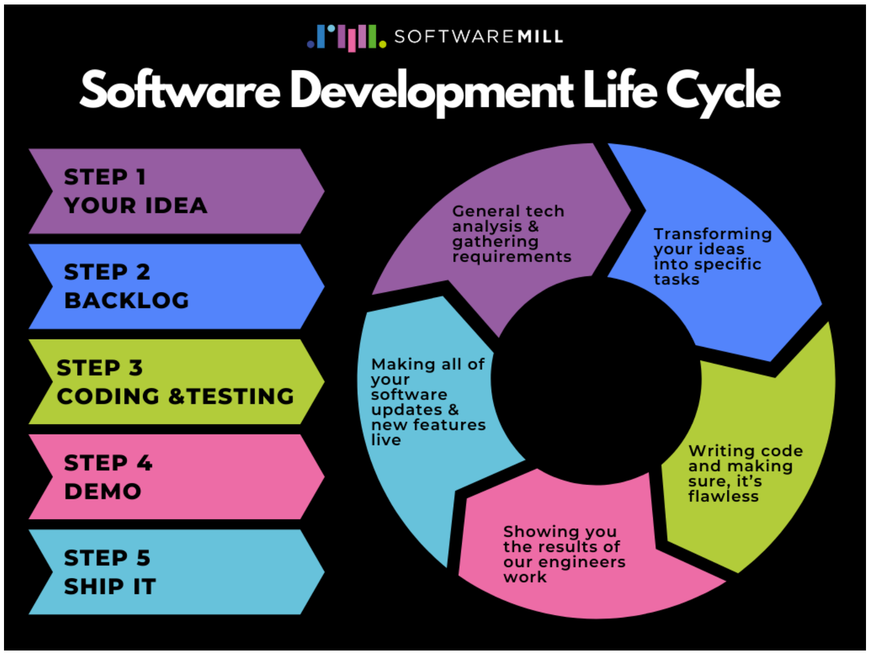

SOFTWAREMILL
Software Development Life Cycle
STEP 1 YOUR IDEA
STEP 2 BACKLOG
STEP 3 CODING &TESTING
STEP 4 DEMO
STEP 5 SHIP IT
General tech analysis & gathering requirements
Transforming your ideas into specific tasks
Writing code and making sure, it's flawless
Showing you the results of our engineers work
Making all of your software updates & new features live

Image source: softwaremill.com

12 Your turn: Different platforms

IDEs are increasingly available online, too, and since 2021, AI coding assistance is integrated though often only available to premium users.

Here is a selection of IDEs that you can peruse at your own peril.

- Code::Blocks - Project-based app (usually installed on Lyon PCs)
- CodeLite - Project-based app
- replit.com - online REPL (with AI) - must register
- Visual Studio Code (VSCode) - plugin based (online and PC)
- GitHub Codespaces - must register and use student ID to use

The last two are probably the most popular modern IDEs, and they both come with AI coding assistance in the form of GitHub Copilot.

The first two are PC apps and require download and installation. They come with an option to also install a compiler (which you need).

You'll be busy enough learning C++ but if you have spare time and interest, take a look at these and/or ask me for a demonstration or help when installing them on your PC.

13 Programming with the Analysis-Coding-Execution Cycle

Programming is a process of problem-solving. Some developers despise the term "coding" because they think it reduces programming to writing code - I use both terms interchangeably (this is not linguistics).

There are many different problem-solving strategies. There cannot be one, because problems are not neatly organized, and often a problem is not fully understood until it is attacked, requiring a strategy change.

Solving computational problems is special because it involves (at least) three unusual companions: data, math, and machines. Each of them have their own nature and approach, and you must satisfy them all! Because that's hard to do, developers in practice specialize: some are better at data engineering, some are better at math, some at machines.

In computing, all work boils down to developing, understanding, and improving detailed, precise sets of instructions for the machine - algorithms.

Data engineers rule at using and feeding, mathematicians at developing, and machine engineers excel at optimizing algorithms.

In a programming environment, the process involves at least:

1. Analysis: define problem and constraints and devise algorithm.
2. Implementation: code algorithm in any language and verify it.
3. Maintenance: use and modify program for different scenarios.

Analysis should occupy you the most though it is perhaps the hardest, because you know nothing at first. It requires:

1. Thoroughly understanding the problem. For example by calculating small examples and limit cases, or by looking at other people's solutions and similar problem settings (i.e. reading).
2. Understand the constraints and requirements, like
    - User interaction (e.g. should numbers be entered?).
    - Data manipulation (e.g. calculating a formula).
3. Problem modularization, like
    - Subdivision of the main problem.
    - Solution of sub-problems separately.
    - Synthesis of the separate solutions.

Here, 1. requires experimentation, 2. requires knowledge of data types and data structures, and 3. requires being able to write functions, and object-orientation.

14 Against time: literate programming

Code is notoriously badly commented and documented. This hinders the development process because it affects readability and understanding of code. Even worse, documentation, when kept separate from the code, goes out of date easily because every programmer likes to code but few like to write and maintain.

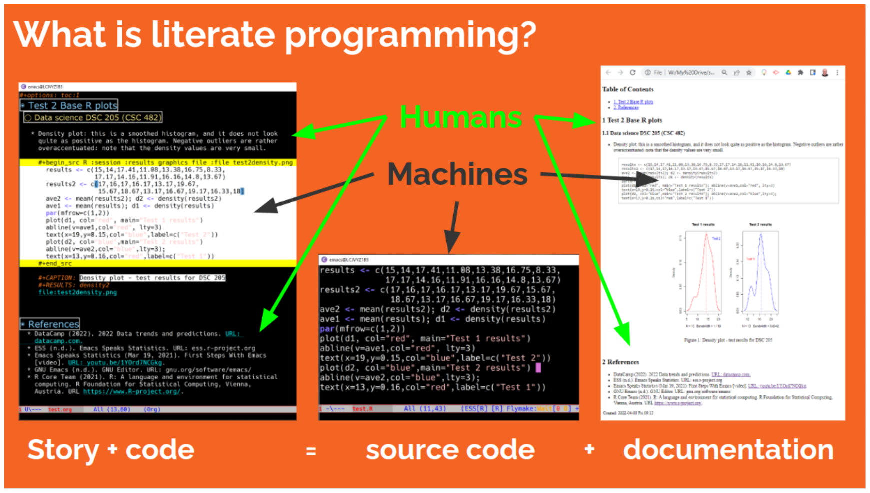
In 1984, Donald Knuth of Stanford U. devised a technique against the decay of documentation and time, called "Literate Programming" (Knuth, 1984). Literate programming tools let authors interleave source code, and descriptive text in one and the same document.

The document can be "tangled" into source code for compilation, and "woven" into a document for human reading and understanding.

In data science, this approach is implemented in the form of interactive notebooks that combine documentation, code and output, allowing developers to address data, math and coding issues at once.

I have written at length about it (Birkenkrahe, 2023), and I was even once (in my graduate student days) maintainer of a virtual library at Stanford U. for literate programming.

You will yourself graduate to "literate programmers" in time when you learn to use the hacker's editor, Emacs, and its notebook plugin, Org-mode, which allows both tangle and weave operations via the `noweb` header argument, and interleaving of documentation, code blocks and output in one file.



```
#+options: toc:1
* Test 2 Base R plots
  ○ Data science DSC 205 (CSC 482)

    * Density plot: this is a smoothed histogram, and it does not look
      quite as positive as the histogram. Negative outliers are rather
      overaccentuated: note that the density values are very small.

    #+begin_src R :session :results graphics file :file test2density.png
      results <- c(15,14,17.41,11.08,13.38,16.75,8.33,
                   17.17,14.16,11.91,16.16,14.8,13.67)
      results2 <- c(17,16,17,16.17,13.17,19.67,
                    15.67,18.67,13.17,16.67,19.17,16.33,18)
      ave2 <- mean(results2); d2 <- density(results2)
      ave1 <- mean(results); d1 <- density(results)
      par(mfrow=c(1,2))
      plot(d1, col="red", main="Test 1 results")
      abline(v=ave1,col="red", lty=3)
      text(x=19,y=0.15,col="blue",label=c("Test 2"))
      plot(d2, col="blue",main="Test 2 results")
      abline(v=ave2,col="blue",lty=3);
      text(x=13,y=0.16,col="red",label=c("Test 1"))
    #+end_src

    #+CAPTION: Density plot - test results for DSC 205
    #+RESULTS: density2
    file:test2density.png

* References
  * DataCamp (2022). 2022 Data trends and predictions. URL:
    datacamp.com.
  * ESS (n.d.). Emacs Speaks Statistics. URL: ess.r-project.org
  * Emacs Speaks Statistics (Mar 19, 2021). First Steps With Emacs
    [video]. URL: youtu.be/1YOrd7NCGkg.
  * GNU Emacs (n.d.). GNU Editor. URL: gnu.org/software/emacs/
  * R Core Team (2021). R: A language and environment for statistical
    computing. R Foundation for Statistical Computing, Vienna,
    Austria. URL https://www.R-project.org/.
1 U\---  test.org      All (13,60)   (Org)
```

```
results <- c(15,14,17.41,11.08,13.38,16.75,8.33,
             17.17,14.16,11.91,16.16,14.8,13.67)
results2 <- c(17,16,17,16.17,13.17,19.67,15.67,
              18.67,13.17,16.67,19.17,16.33,18)
ave2 <- mean(results2); d2 <- density(results2)
ave1 <- mean(results); d1 <- density(results)
par(mfrow=c(1,2))
plot(d1, col="red", main="Test 1 results")
abline(v=ave1,col="red", lty=3)
text(x=19,y=0.15,col="blue",label=c("Test 2"))
plot(d2, col="blue",main="Test 2 results")
abline(v=ave2,col="blue",lty=3);
text(x=13,y=0.16,col="red",label=c("Test 1"))

1 -\---  test.R      All (11,43)   (ESS[R] [R] Flymake:Wait[0 0] +
```

Figure 4: What is literate programming? (2022)

See also: Separate lecture and practice file (next week).

15 Problem analysis and algorithm design examples

Five sample problems to practice problem analysis and algorithm design in the form of pseudocode and/or BPMN diagrams.

You find the solution to each problem in chapter 1 of the textbook (pp. 14-19). We will cover C++ implementations later in this course - they involve material up to and including chapter 9 (pp. 610-650).

16 Finding the perimeter and area of a rectangle

Problem: design an algorithm to find the perimeter and area of a rectangle.

1. Analysis:
   - The `area` of a rectangle can be obtained by multiplying its `length` and its `width` with one another.
   - The `perimeter` of a rectangle is twice the sum of `length` and `width`.
   - We need `length` and `width` as input in the same units of measurement so that we can compute `area` as their product.
   - We need to print the result on the screen.
2. Algorithm:
   (a) Get length of rectangle.
   (b) Get width of rectangle.
   (c) Compute area as `area = length * width`.
   (d) Compute perimeter as `perimeter = 2 * (length + width)`.
3. Pseudocode:

```
// Algorithm CalculateRectangleProperties
Input: length, width (both numbers)
Output: perimeter, area (both numbers)

Begin
    // Calculate the perimeter of the rectangle
    perimeter = 2 * (length + width)

    // Calculate the area of the rectangle
    area = length * width

    // Return or print the perimeter and area
    return perimeter, area
End
```

4. Process diagram (BPMN):

[width=.9]../img/rectangle

17 Calculate sales price with tax

Problem: In this example, we design an algorithm that calculates the sales tax and the price of an item sold in a particular state.

The sales tax is calculated as follows: The state's portion of the sales tax is 4%, and the city's portion of the sales tax is 1.5%. If the item is a luxury item, such as a car more than $50,000, then there is a 10% luxury tax.

1. Analysis

   - Input constants:
     (a) `price` of an item.
     (b) `salesCityTax` is 1.5%.
     (c) `salesStateTax` is 4%: or `price * 0.04`
     (d) `luxuryTax` is 10%: `price * 0.10`
   - Output variables:
     (a) `totalTax`: the sum of `salesCityTax`, `salesStateTax`, and `luxuryTax`.
     (b) `salesPrice`: `price` + `totalTax`.
   - However the luxury tax is only applied if the item `price` is above $50,000. This requires a conditional statement that is only executed if the condition is true.

2. Algorithm:

   (a) Set tax rate constants.
   (b) Get item `price`.
   (c) Compute total tax as `price * (city tax + sales tax).`
   (d) If `price > 50000`, add luxury tax.

3. Pseudocode:

```
//Algorithm CalculateSalesPrice
  Input: basePrice
  Output: salesPrice

  Begin:
      // define tax rates
      STATE_TAX_RATE = 0.04 // 4%
      CITY_TAX_RATE = 0.015 // 1.5%
      LUXURY_TAX_RATE = 0.10 // 10%

      // initialize luxuryTax
      luxuryTax = 0

      // if item is a luxury item
      if base_Price >= 50000 then
         luxuryTax = basePrice * LUXURY_TAX_RATE

      // calculate state and city sales tax
      stateTax = basePrice * STATE_TAX_RATE
      cityTax = basePrice * CITY_TAX_RATE

      // calculate total tax
      totalTax = stateTax + cityTax + luxuryTax

      // calculate total sales price including tax
      salesPrice = basePrice + totalTax

      // return or print result
      return salesPrice
  End
```

4. Process (BPMN diagram)

   [width=.9]../img/tax_1

18 Calculate monthly paycheck of a store clerk

Problem description: design an algorithm to compute the monthly salary based on a base salary, sales and service bonus payments.

Every salesperson has a base salary. The salesperson also receives a bonus at the end of each month, based on the following criteria: If the salesperson has been with the store for five years or less, the bonus is \$10 for each year that he or she has worked there. If the salesperson has been with the store for more than five years, the bonus is \$20 for each year that he or she has worked there. The salesperson can earn an additional bonus as follows: If the total sales made by the salesperson for the month are at least \$5,000 but less than \$10,000, he or she receives a 3% commission on the sale. If the total sales made by the salesperson for the month are at least \$10,000, he or she receives a 6% commission on the sale.

1. Analysis: get years of service, sales per month to compute bonus payments, and base salary to compute total salary:

```
// Algorithm: Monthly Paycheck
Input: base_salary (float),
       service_years (int),
       sales_month (float)
Output: service_bonus,
        sales_bonus,
        total_salary

Begin
   // Set constants
   LOW_SERVICE (const float = 10),
   HIGH_SERVICE (const float = 20),
   LOW_SALES (const float 0.3),
   HIGH_SALES (const float 0.6)

   // get input variables
   base_salary
   service_years
   sales_month

   // calculate service bonus
   if service_years less or equal than 5:
      service_bonus = service_years * LOW_SERVICE
   if service_years greater than 5:
      service_bonus = service_years * HIGH_SERVICE
```

```
    // calculate sales bonus
    if sales_month less or equal than 5,000 and smaller than 10,000
       sales_bonus = sales_month * LOW_SALES
    if sales_month greater or equal than 10,000
       sales_bonus = sales_month * HIGH_SALES

    // calculate total salary
    total_salary = base_salary + service_bonus + sales_bonus

    // return or print results
    return total_salary
End
```

2. Process diagram (BPMN):

   [width=.9]../img/salary

19 Design a number-guessing game

Problem: In this example, we design an algorithm to play a number-guessing game. The objective is to randomly generate an integer greater than or equal to 0 and less than 100. Then prompt the player (user) to guess the number. If the player guesses the number correctly, output an appropriate message. Otherwise, check whether the guessed number is less than the random number. If the guessed number is less than the random number generated, output the message, “Your guess is lower than the number. Guess again!”; otherwise, output the message, “Your guess is higher than the number. Guess again!”. Then prompt the player to enter another number. The player is prompted to guess the random number until the player enters the correct number.

1. Analysis: need to be able to generate (pseudo) random numbers. A winning strategy here is the "binary search": half the (wrong) guess each time (this is not required for the algorithm).

   Strategy example: the (random) number is 40.

   (a) First guess: 50 - Response: too high.
   (b) Second guess: 50/2 = 25 - Response: too low.
   (c) Third guess: mean of (25,50): 37[.5] - Reponse: too low.
   (d) Fourth guess: mean of (37,50): 43[.5] - Response: too high.

   (e) Fifth guess: mean of (37,43): 40.

2. Algorithm:

```
// Algorithm: number-guessing game
Input: random_number in [1,100] (integer)
       my_guess (user)
Output: feedback

Begin:
   // Get random integer
   random_number in [1,100]

   Repeat:

       // Ask user to guess a number
       get my_guess from the keyboard

       // Compare user guess and random number
       if my_guess is equal to random_number
             Output: "You guessed right! Game over!"
             Quit the program
       otherwise, if my_guess is smaller than random_number
             Output: "Your guess is lower than the number. Guess again!"
             Go back to Repeat:
       otherwise
             Output: "Your guess is higher than the number. Guess again!"
             Go back to Repeat:

   End Repeat
End
```

3. Process model (BPMN):

[width=.9]../img/guessNumberCpp

20 Calculate the grade for different students

Problem: There are 10 students in a class. Each student has taken five tests, and each test is worth 100 points. We want to design an algorithm to calculate the grade for each student, as well as the class average. The grade is assigned as follows: If the average test score is greater than or equal to 90, the grade is A; if the average test score is greater than or equal to 80 and less than 90, the grade is B; if the average test score is greater than or equal to 70 and less than 80, the grade is C; if the average test score is greater than or equal to 60 and less than 70, the grade is D; otherwise, the grade is F. Note that the data consists of students' names and their test scores.

1. Analysis: "The data consists of students' names and their test scores". We can divide this problem into subproblems:

    - Find the average test score over five tests for each student.
    - Determine grade according to the average test score for each student.
    - Join subproblems by solving both problems for each student in turn and calculate class average.
    - Calculate average score:

    ```
    // Algorithm to find the average test score for each student
    Input: student name or ‘ID‘ (int)
           student ‘score‘ test 1 through 5 (float)
    Output: student’s ‘average‘ score (float)

    Begin
       // Input
       Get student ‘id‘ and test ‘score‘ 1 through 5

       // For each student, calculate ‘average‘ test score
       For each student:
           average = sum of scores / 5

       // Return result
       Return average test score for each student
    End
    ```

    - Determine grade:

    ```
    // Algorithm to determine the grade for each student
    Input: Average grade for each student
           Constant: cutoff for letter grades A-F
    Output: Letter grade for each student A-F (char)
    ```

```
Begin
   // Determine letter grade
   If average test score is greater than or equal to 90
      grade = A
   Otherwise:
      If average test score is greater than or equal to 80
         grade = B
      Otherwise:
         If average test score is greater than or equal to 70
            grade = C
         Otherwise:
            If average test score is greater than or equal to 60
               grade = D
            Otherwise:
            grade = F
End
```

- Both problems together:

```
// Algorithm to determine letter grade based on average test scores
Input: student name or ‘ID‘ (int)
       student ‘score‘ test 1 through 5 (float)
       letter grades (const char)
Output: students’ average score (float)
        students’ letter grades (char)
        class average (float)

Begin
   // Initialize class average
   class_average = 0

   // Iterate over list of students
   Repeat
      // Calculate student’s average test score
      average = sum of scores / 5
      // Determine student’s letter grade
      If average test score is greater than or equal to 90
         grade = A
      Otherwise:
         If average test score is greater than or equal to 80
```

```
            grade = B
         Otherwise:
            If average test score is greater than or equal to 70
               grade = C
            Otherwise:
               If average test score is greater than or equal to 60
                  grade = D
               Otherwise:
                  grade = F
   End

   // calculate class average
   class_average = sum of average scores / 10

   // return results
   print student's 'id', test 'score', letter 'grade'
   print 'class_average'
End
```

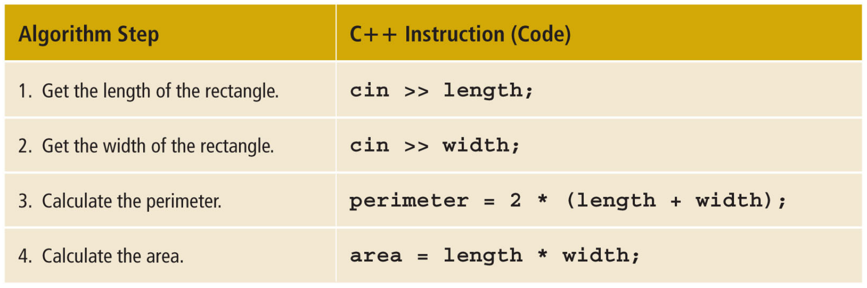
2. Process model:

[width=.9]../img/gradesCpp

21 C++ code example

- Problem analysis is the most important part of the solution process.
- The pseudocode is sufficient to code in any language.
- The process model helps to keep an overview and debug logic errors.

The table gives a preview of C++ syntax for the rectangle problem:



Algorithm Step	C++ Instruction (Code)
1. Get the length of the rectangle.	`cin >> length;`
2. Get the width of the rectangle.	`cin >> width;`
3. Calculate the perimeter.	`perimeter = 2 * (length + width);`
4. Calculate the area.	`area = length * width;`

- Four steps of the algorithm from the computers point of view:
  1. Accept a common input value for the variable `length`.
  2. Accept a common input value for the variable `width`.
  3. Compute an expression and assign it to the variable `perimeter`.
  4. Compute an expression and assign it to the variable `area`.

22 Programming Methodologies

1. Structured programming (also: modular programming, stepwise refinement, top-down or bottom-up design): dividing a problem into sub-problems, solving each and combining the sub-solutions to solve the main problem.

   Example: finding the highest proper factor of an integer with subtraction (rather than division) only - see src/factorSSEM.cpp.

   (a) Load I/O library (for the preprocessor)
   (b) Declare variables and data types.
   (c) Iterate over range of values to find the factor (loop).
   (d) Print the results.

2. Object-oriented design (OOD or OO programming): identify problem components (objects), and their interactions with one another. Each object consists of data (or attributes) and operations (or methods).

   Example: write a program that automates a library.

   Objects include: books and customers.

   - Book data include: title, author, year of publication, number of copies in library, etc.
   - Customer data include: library number, name, address, date of membership, type of membership, etc.

   Operations on books include: checking if book is available, ordering more books, retiring books, etc.

   Operations on customers include: issuing membership, taking out a book, returning a book, paying a fine, etc.

C++ is an OO extension of the C programming language. For some problems, structured programming will suffice (e.g. for scientific computing), for others, OOD will be useful (e.g. for developing library software).

23 ANSI/ISO Standard C++

The table indicates the evolution of C++ from C:

1970s	C	Dennis Ritchie
1980s	C++	Bjarne Stroustrup
1989		First release
1998-2023		Standard updates

Since its standardization in 1989, C++ has undergone several significant updates, each introducing key innovations:

1. C++98: This was the first standardized version of C++, solidifying the features that existed in the language since its creation by Bjarne Stroustrup in the 1980s. It included major features like Standard Template Library (STL), templates, exceptions, namespaces, and the bool type.

2. C++03: This release was more of a bug fix version of C++98 rather than introducing new features. It focused on fixing issues and ambiguities in the 1998 standard.

3. C++11: This was a major update, often referred to as "C++0x" during its development. It introduced several significant features, such as:

   - Auto type declarations
   - Range-based for loops
   - Smart pointers ($\text{unique}_{\text{ptr}}$, $\text{shared}_{\text{ptr}}$)
   - Lambda expressions
   - Concurrency support (thread, mutex, etc.)
   - nullptr keyword
   - Move semantics and rvalue references
   - Initializer lists
   - Static assertions

4. C++14: This release included improvements and refinements to features introduced in C++11, such as:
   - Auto return type for functions
   - Improved lambda expressions
   - Digit separators
   - Binary literals
   - Deprecated attributes
5. C++17: This standard further enhanced the language with features like:
   - std::optional, std::variant, and std::any
   - Parallel algorithms in the STL
   - Structured bindings
   - If and switch with initializers
   - Inline variables
   - Fold expressions (for variadic templates)
6. C++20: This is one of the most significant updates since C++11, introducing features like:
   - Concepts and ranges
   - Coroutines
   - Modules
   - Three-way comparison operator (spaceship operator)
   - Designated initializers
   - Ranges library
   - constexpr and consteval improvements
   - std::span Source: ChatGPT [12/02/2023]

Modern C++ feels like a new programming language compared to the original version (which I learnt in the early 1990s!).

What are all the changes about? They introduce different levels of "abstraction" into the language, making maintenance and development easier through:

1. simplifying code and increasing readability

2. enhancing safety and correctness
3. improving performance
4. supporting modern programming paradigms
5. encouraging good programming practices

Since 2002, the C++ Standard Library has undergone significant changes and enhancements, particularly with the introduction of new standards such as C++11, C++14, C++17, and later versions. Some of the key changes include:

- Smart Pointers (C++11): Smart pointers like std::unique$_{\text{ptr}}$, std::shared$_{\text{ptr}}$, and std::weak$_{\text{ptr}}$ were introduced for better memory management, reducing the risks of memory leaks and dangling pointers.
- Lambda Expressions (C++11): Support for lambda expressions, enabling the writing of inline anonymous functions, which is particularly useful for functional-style programming and in conjunction with the Standard Library's algorithms.
- Concurrency Support (C++11): Introduction of a threading library including std::thread, std::mutex, and other utilities for multithreading, making concurrent programming more accessible and safer.
- Move Semantics (C++11): Move constructors and move assignment operators for more efficient object transfers, especially important for managing resources in container classes.
- Range-based for Loops (C++11): Simplification of iterating over containers with range-based for loops, improving code readability and maintainability.
- Regular Expressions (C++11): The <regex> library for regular expression processing, providing powerful pattern matching and text processing capabilities.
- Random Number Generation (C++11): Improved random number generation facilities in the <random> library, offering a wide range of generators and distributions.
- Type Traits and Compile-Time Utilities (C++11 and later): Enhanced type introspection and compile-time programming support through type traits and constexpr functions.

- The Chrono Library (C++11): A time library (<chrono>) for dealing with durations, time points, and clocks, facilitating time-related operations with greater precision.

- Unordered Containers (C++11): Introduction of unordered (hash-based) containers like unordered$_{\text{map}}$ and unordered$_{\text{set}}$ for efficient lookup operations.

- Tuple Types (C++11): The std::tuple class for handling fixed-size collections of heterogeneous values.

- Container Improvements (C++11 and later): Enhancements to existing containers (e.g., std::vector, std::array) and new functions for improved usability and performance.

- Optional, Variant, and Any (C++17): Introduction of std::optional, std::variant, and std::any for more flexible handling of values and type-safe unions.

- Filesystem Library (C++17): The <filesystem> library for portable and intuitive file system operations.

- Parallel Algorithms (C++17): Parallel execution support for many of the algorithms in the Standard Library, enabling better utilization of multi-core processors.

- String and Character Utilities (C++11 and later): Enhanced support for Unicode and new utilities for string manipulation.

These changes reflect the ongoing evolution of C++ to meet modern programming needs, focusing on memory safety, performance, convenience, and support for modern programming paradigms like functional programming and concurrency.

24 Glossary (Concepts and Code)

55 concepts and commands to remember:

	TERM	DEFINITION
Historical overview	Punch card	Storage medium
Elements of computers	CPU	Central Processing Unit
Hardware	RAM	Random Access Memory
	NVM	Non Volatile Memory
	Input device	Keyboard, mouse, NVM etc.
	Output device	Monitor, printer, NVM etc.
	von Neumann architecture	CPU + RAM + NVM + I/O
Software	Manchester Baby/SSEM	Small Scale Exp Machine (1948)
		with the first stored program
	Pythontutor.com	Online program visualization
	Analog vs. digital signal	Continuous vs. binary values
Language of computers	Machine code	Binary encoded commands
	Bit	Binary digit 0 or 1
	Byte	Sequence of 8 bits
	KB, MB, GB, TB, PB, EB, ZB	Powers of 2
		1 KB $= 2^{10}$ bytes $= 1024$ bytes
Representation	ASCII	American Standard Code for
		$128 = 2^{7\text{symbols}}$ (7 bits)
	Unicode	$65536 = 2^{16}$ symbols
Machine languages	Assembly language	Encode commands as mnemonic
Processing C++ Pgms	CLI	Command line interface
	Syntax highlighting	Text editor feature
	GitHub	Software development platform
	`#include`	Preprocessor directive
	`main`	Main program function
	String	Ordered character sequence
	`std`	Namespace (command repository)
	`.cpp`	C++ file extension
	Compiler	Program to produce executable
	`.exe`	Windows OS executable file
	Flag	Compiler option like `-o`
	Shell	Program for OS commands
	Linker	Add libraries
	Loader	Loads machine code into RAM
IDEs	IDE	Integrated dev. environment
	Emacs	Self-extensible text editor
Literate programming	Org-mode	Emacs mode for Lit Prog
	tangle	Program to source code
	weave	Program to documentation
	noweb	Lit Prog preprocessor

Problem analysis and algorithm design	Pseudocode	Program design without syntax
	Algorithm	Instruction set
	Analysis	Dissect and understand
	Implementation	Code
	Maintenance	Keep code alive
	BPMN diagram	Business Process Model & Notation
C++ code example	`cin`	Common input
	`cout`	Common output
	Variable	Accepts value, memory cell
	`=`	Assignment operator
	`>>`	Input stream operator
	`<<`	Output stream operator
	`;`	Command delimiter
	Expression	Statement that leads to a value
Pgm Methodologies	Structured programming	Divide problem in sub-problems,
	Modular programming	solve each and combine solutions
	Step-wise refinement	(e.g. factoring an integer)
	top-down/bottom-up design	
	Object-oriented design	Identify objects and their interactions (e.g. library)
ANSI/ISO Std C++	Abstraction	Increase high level concepts and remove details

25 Quick Review

1. The first man-made computing devide was the Abacus for addition and subtraction.
2. Early computing devices and mainframe computers used punchcards to store data (including programming instructions).
3. A computer is an electronic device capable of performing arithmetic and logical operations.
4. A computer system has two components: hardware and software.
5. The central processing unit (CPU) and the main memory are examples of hardware components.
6. All programs must be brought into main memory before they can be executed.

7. When the power is switched off, everything in the main memory is lost.
8. Secondary storage provides permanent storage for information. Hard disks, flash drives, and CD-ROMs are examples of secondary storage.
9. Input to the computer is done via an input device. Two common input devices are the keyboard and the mouse.
10. The computer sends its output to an output device, such as the computer screen or a printer. Software are programs run by the computer.
11. The operating system handles the overall activity of the computer and provides services.
12. The most basic language of a computer is a sequence of 0s and 1s called machine language. Every computer directly understands its own machine language.
13. A bit is a binary digit, 0 or 1.
14. A byte is a sequence of eight bits.
15. A sequence of 0s and 1s is referred to as a binary code or a binary number.
16. One kilobyte (KB) is 2^{10} = 1024 bytes; one megabyte (MB) is 2^{20} = 1,048,576 20 bytes; one gigabyte (GB) is 2^{30} = 1,073,741,824 bytes; one terabyte (TB) is 2^{40} = 1,099,511,627,776 bytes; one petabyte (PB) is 2^{50} = 1,125,899,906,842,624 50 bytes; one exabyte (EB) is 2^{60} = 1,152,921,504,606,846,976 60 bytes; one zettabyte (ZB) is 2^{70} = 1,180,591,620,717,411,303,424 70 bytes; and one Yottabyte (YB) is 2^{80} = 1,000 ZB.
17. 1 KB: a paragraph of text. 1 MB: a short novel. 1 GB: Beethoven's 5th symphony. 1 TB: all X-rays in a large hospital. 1 PB: half the contents of all 123,000 US academic research libraries. 1 EB: a video call that started 237,823 years ago. 1 ZB: as much information as there are grains of sand on all the world's beaches. 1 Yottabyte is as much information as there are atoms in 7,000 human bodies, more than the amount of data currently (2023) stored in all of the worlds PCs, more than 2 billion (Source).
18. Assembly language uses easy-to-remember instructions called mnemonics.

19. Assemblers are programs that translate a program written in assembly language into machine language.

20. Compilers are programs that translate a program written in a high-level language into machine code, called object code.

21. A linker links the object code with other programs provided by the integrated development environment (IDE) and used in the program to produce executable code.

22. Typically, six steps are needed to execute a C11 program: edit, preprocess, compile, link, load, and execute.

23. A loader transfers executable code into main memory.

24. An algorithm is a step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

25. The problem-solving process has three steps: analyze the problem and design an algorithm, implement the algorithm in a programming language, and maintain the program.

26. In structured design, a problem is divided into smaller subproblems. Each subproblem is solved, and the solutions to all of the subproblems are then combined to solve the problem.

27. In object-oriented design (OOD), a program is a collection of interacting objects.

28. An object consists of data and operations on that data.

29. The ANSI/ISO Standard C11 syntax was approved in mid-1998.

30. The second standard of C++, C++11, was approved in 2011. C++14 was approved in 2014.

26 Review Questions with answers (11)

1. Which hardware component performs arithmetic and logical operations? (2) In short: The Central Processing Unit or CPU.

   The hardware component that performs arithmetic and logical operations in a computer is the Arithmetic Logic Unit (ALU). The ALU is a critical part of the central processing unit (CPU) and is responsible for carrying out all arithmetic and logic operations.

In more detail:

- Arithmetic Operations: These include basic mathematical operations like addition, subtraction, multiplication, and division.
- Logical Operations: These involve boolean logic operations such as AND, OR, NOT, XOR, and comparison operations like equal-to, less-than, and greater-than.

The ALU can perform these operations on binary data (bits). The results of these operations are used by the CPU to carry out tasks and process instructions as part of executing computer programs. The sophistication and performance of the ALU play a significant role in determining the overall speed and efficiency of a computer.

2. Which number system is used by a computer? (3) The computer's architecture uses the binary number system, which is based on the number 2: in this system, every number is expressed in powers of 2, e.g. the number 8 in the decimal system, is written in binary as `1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0` or 1000.

   In C++ you can use the `std::stoi` (string-to-integer) function to convert binary to decimal:

   ```
   std::cout << std::stoi("1000",nullptr,2);
   ```

   ```
   8
   ```

3. What is an object program? (5) Object code, or an object file, is the output generated by a compiler after it processes the source code. When you write a program in a high-level language like C or C++, the compiler translates this code into object code, which is a machine-readable format (binary format). However, this object code is not yet executable. It often needs to be linked with other object files and libraries to create a final executable program.

4. What is linking? (8) Linking in the context of computer programming and compiling is a process that combines various pieces of code and data into a single file that can be executed by a computer. This process

occurs after individual pieces of a program have been compiled into object files, which are binary files containing machine code.

Here are key points about linking:

(a) Combining Object Files: A program often consists of multiple source code files, each compiled into separate object files. The linker combines these object files into a single executable.

(b) Resolving References: Programs typically contain references to functions and variables defined in different files or libraries. During compilation, these references are left unresolved. The linker connects these references to the appropriate addresses.

(c) Static Linking: This involves including all the necessary library functions and resources within the executable file. The final executable is larger but self-contained, meaning it doesn't depend on external library files at runtime.

(d) Dynamic Linking: In contrast, dynamic linking involves keeping the code of libraries separate from the executable. The links to these libraries are resolved either at the time the program is loaded (load-time dynamic linking) or at runtime (runtime dynamic linking). This can save space and allow for the updating of libraries without recompiling the program.

(e) Symbol Resolution: The linker matches function calls and variable references in the object files with the correct function definitions and variable addresses.

(f) Relocation: Adjusts the object code and data to be loaded at a specific memory address. This process includes adjusting pointers and references to absolute addresses.

(g) Error Checking: The linker checks for errors like undefined symbols (functions or variables that are declared but never defined) or multiple definitions of the same symbol.

After linking, the output is an executable file that can be run on the computer. The specific actions performed by the linker can vary based on the operating system and the programming language used.

5. Which program loads the executable code from the main memory into the CPU for execution? (8) The program that loads the executable code from main memory into the CPU for execution is typically part

of the operating system and is known as the "loader." The loader is a critical component of an operating system's process management.

Here's a brief overview of what the loader does:

(a) Loading Executable into Memory: When a program is executed, the loader loads the executable file (which contains the compiled code of the program) from the disk into the main memory. This includes both the code and data segments of the executable.

(b) Relocation: If necessary, the loader relocates code and data addresses to the actual addresses allocated in memory. This step is crucial for programs that are not position-independent.

(c) Resolving Dynamic Links: For executables that use dynamic linking (linking to shared libraries), the loader is responsible for loading the required shared libraries into memory and linking them to the executable. This might involve resolving symbols and addresses at runtime.

(d) Setting Up Program Execution: The loader sets up the initial program stack and initializes the necessary CPU registers, including the instruction pointer, which is set to the beginning of the program's entry point (often the main function in C and C++ programs).

(e) Handing Control to the Program: Once the program is loaded and set up, the loader transfers control to the program, enabling it to start executing on the CPU.

The loader's operations are typically transparent to the user and are handled automatically by the operating system whenever a user or another program initiates the execution of a program. This process is a part of the broader set of functionalities that an operating system provides for managing processes and resources.

6. In a C++ program, preprocessor directives begin with which symbol? (8) They begin with the hash or pound symbol (#).

7. In a C++ program, which program processes statements that begin with the symbol #? (8) In a C++ program, statements that begin with the symbol # are processed by the preprocessor. The preprocessor is a program that processes directives in the source code before it is compiled by the compiler.

   Here are some key points about the C++ preprocessor:

(a) Preprocessing Directives: The lines that begin with # are called preprocessing directives. These directives instruct the preprocessor to perform specific operations before the actual compilation process begins.

(b) Common Directives: Some common preprocessing directives include:

- `#include`: Used to include the contents of a file in the source code, typically header files.
- `#define`: Used to define macros or symbolic constants.
- `#ifdef`, `#ifndef`, `#if`, `#endif`, `#else`, `#elif`: Used for conditional compilation.
- `#undef`: Used to undefine previously defined macros.
- `#pragma`: Used to provide additional information to the compiler, often platform-specific.

(c) Macro Expansion: The preprocessor replaces macros with their corresponding values or code snippets. This happens before the compiler sees the code.

(d) File Inclusion: When the preprocessor encounters an `#include` directive, it replaces it with the entire content of the included file. This is commonly used for including header files that contain function declarations, class definitions, or other macros.

(e) Conditional Compilation: Preprocessor conditional directives allow parts of the code to be compiled or omitted based on certain conditions. This is useful for enabling or disabling specific sections of code, perhaps for different operating systems or build configurations.

(f) No Compilation: The preprocessor does not compile or execute code; it only processes these directives to prepare the code for compilation.

After preprocessing, the modified source code is passed to the actual compiler, which then compiles the C++ code into object code. The preprocessor is an integral part of the C++ compilation process, and its directives provide a powerful way to influence the compilation process.

8. What is programming? Programming is the process of designing and building an executable computer program to accomplish a specific computing task or to solve a particular problem. It involves tasks such

as analysis, generating algorithms, profiling algorithms' accuracy and resource consumption, and the implementation of algorithms (usually in a chosen programming language, also known as coding).

Key aspects of programming include:

(a) Writing Source Code: This involves using a programming language to write instructions that a computer can execute. These instructions are written in a form that is understandable by humans and then translated into a form that a computer can process.

(b) Problem-Solving and Logical Thinking: Programming often requires breaking down a problem into smaller, manageable parts and then solving each part logically and systematically.

(c) Algorithm Development: A significant part of programming is developing algorithms, which are step-by-step procedures or formulas for solving a specific problem.

(d) Debugging: This involves identifying and fixing errors or bugs in the code to ensure that the program works correctly and efficiently.

(e) Testing and Validation: Programmers must test their code to make sure it performs as expected in all anticipated scenarios. This can involve various types of testing methodologies.

(f) Maintenance and Updating: Once a program is in use, it may require maintenance to fix unforeseen bugs, enhance its performance, or adapt it to new requirements.

Programming can be done using various programming languages, each suited for different types of tasks. It plays a crucial role in virtually every aspect of modern technology and is foundational to fields like software development, data analysis, artificial intelligence, and more.

9. What is an algorithm? An algorithm is a set of step-by-step instructions or rules designed to solve a specific problem or accomplish a particular task. Think of it as a recipe in a cookbook: just as a recipe provides detailed directions on how to cook a dish, an algorithm provides clear instructions on how to perform a computation, process data, or automate tasks.

   Algorithms are used in all areas of computing and are fundamental to programming. They can range from simple procedures, like a recipe for making tea, to complex operations, like algorithms for internet search engines or for encrypting sensitive information. The key characteristics of an algorithm are that it must be clear, unambiguous, effective, and finite – meaning it has a clear starting and ending point.

10. Describe the steps required by the problem-solving process.

    (a) Analyze and outline the problem and its solution requirements, and design an algorithm to solve the problem.
    (b) Implement the algorithm in a programming language, such as C++, and verify that the algorithm works.
    (c) Maintain the program by using and modifying it if the problem domain changes.

11. Describe the steps required in the analysis phase of programming.

    (a) Thoroughly understand the problem.
    (b) Understand the problem requirements.
    (c) If the problem is complex, divide the problem into subproblems and repeat step 1 and 2.

27 References

- BBC (1991). History of Computers. url: youtube.
- Birkenkrahe M (2023). Teaching Data Science with Literate Programming Tools. Digital 3(3):232-250. url: mdpi.com.
- Knuth D (1984). Literate Programming, Comput. J. 27:97-111. url: literateprogramming.com
- Malik DS (2018). C++ Programming - Program Design to Data Structures, Cengage. pdf: drive.google.com.
- Martin CD (1993). The Myth of the Awesome Thinking Machine. ACM Comm. 36(4):120-133. doi: 10.1145/255950.153587. (PDF)
- The World of Science (2022). How Computers Evolved? History of Computers From 1642 to 2022. url: youtube.com.