History of Computing

Computer: A device that can be programmed or a description of a device that can be programmed (e.g. Turing machines).

Computers were originally people who carried out mathematical operations.

Analogue Computers

Parameters are **precisely** represented and manipulated by **continuously variable analogues**. e.g. voltage, current.

The Difference Engine

a.k.a The Babbage Engine

- First automatic mechanical calculator.
- Programmed using punch cards and based on weaving looms.

Digital Computers

Parameters represented and manipulated using **discrete values.**

1st Generation – Valves & Vacuum Tubes (40s)

Valves & Vacuum Tubes: Amplify and switch on/off electronic signals.

Colossus

- First programmable digital computer.
- Broke the Enigma code.

The Manchester Baby

a.k.a Small-Scale Experimental Machine (SSEM)

- First electronic stored-program computer.
- Built to show it was possible and practical to store data electronically.

Problems with 1st Generation Computers

- Expensive.
- Large.

2nd Generation – Transistors (50s)

Transistors: Same function but smaller and more energy-efficient.

3rd Generation – Integrated Circuits (60s)

Integrated Circuits: A set of circuits on a chip of semiconductor material called a silicon wafer.

Properties

- 10s-100s of transistors per chip.
- More compact.

Other Machines

- **Eliza:** First ever chatbot invented by Joseph Weizenbaum.
- **Sketchpad:** Computer program which revolutionised interactive graphical computing.

Theoretical Foundations

The Universal Language

The idea of **a language that can express all human thought and maths problems** through logical symbols.

- Envisioned by Leibniz.

The Decision Method

The concept of **a formal system that can solve any maths problem and logical reasoning** algorithmically and mechanically.

- Envisioned by Leibniz.
- Built on the idea of a universal language.
- Laid the groundwork for formal systems and logic. (e.g. set theory)

The Entscheidungsproblem

Asks if there exists a systematic method (an algorithm) to **determine the truth or falsehood of any mathematical statement**.

- Created by Hilbert and Ackermann.
- Spawned from Leibniz' ideas.

Church and Turing independently disproved it, inventing Lambda calculus and Turing Machines respectively in the process. This showed that there are **limits to computation**, laying down the **foundation for theoretical computer science**.

Principa Mathematica

Russel and Whitehead **attempted to derive all maths from logic**. They hoped to prove maths is:

- **Complete:** Every statement can be proven true or false.
- **Consistent:** No contradictions.

The Incompleteness Theorem

Proved that **any formal system cannot be both complete and consistent**, undermining Principa Mathematica.

- Created by Gödel.

Lambda Calculus

Formally defines a computable function and provides a framework for analysing them and their applications.

- Created by Church.
- A **theoretical basis for functional programming languages**.

The Basic Concept

- **Lambda (λ) Symbol:** Introduces the definition of a function.
- **Parameters:** Listed after the lambda symbol.
- **Body:** Listed after the last dot.
- Numbers following this expression are an argument being applied to the function.

Examples

1. $\lambda x.(2x + 1)3$ means $2(3) + 1 = 7$
2. $\lambda f.\lambda x.fx$ Is a function which takes two parameters, a function and a variable, and applies the function to the variable once.

This is a **Church numeral** and represents 1.

Turing Machines

Abstract devices designed to **model the logic of any computer algorithm by reading and writing to a tape**. They establish a baseline for "computable", helping us to understand the limits of computation.

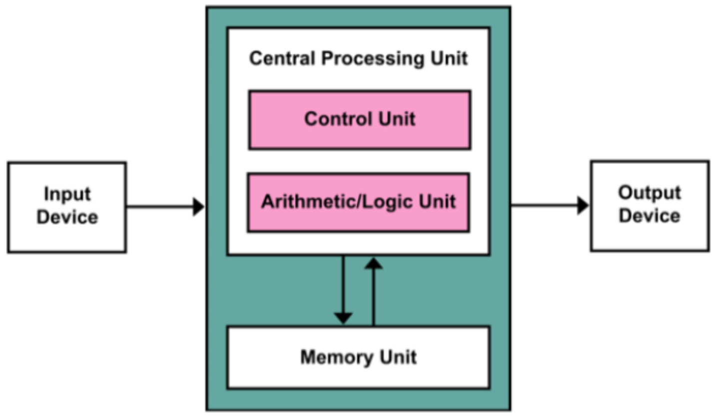
- **Turing Complete:** Can be used to simulate a Turing machine.
- To be a computer, a device must be expressible as a Turing machine.
- Lambda calculus and Turing machines are computationally equivalent.
- Turing machines have 7 components.

Church-Turing Thesis

Any function which can be calculated by an "effective method" or algorithm can be computed by a Turing machine. So, a function on the natural numbers can be calculated if and only if it is computable by a Turing machine.

Key Concepts

Von Neumann Architecture



Arithmetic Logic Unit (ALU)

Carries out computational operations. It contains registers called **accumulators/data registers** for holding data being processed.

Control Unit (CU)

Supplies operations and operands in a sequence to other components which **enables autonomy.**

Random Access Memory (RAM)

Often called the **store or DRAM.** Contains data & instructions currently in use.

I/O Devices

Enable communication between the machine and the outside world.

The Type of a Variable

Without the context of the programmer's intentions and the programming language, an operation can mean lots of different things.

e.g. z=x/y can be a web address or a calculation depending on the programmer's intent and the language.

The State of Machine

The current system configuration, including:

- Stored data. (e.g. variable values)
- Memory contents.
- Active processes.

Computing as String Processing

At its core, computing is the manipulation of symbols, often in the form of strings.

- **Data Representation:** Text and binary data are strings of characters or bits.
- **Programming:** Source code and algorithms often operate on strings, which are stored as bits.

Standards

Guidelines and best practices that ensure quality, compatibility, and consistency in the creation and maintenance of software. There are two types:

- **Proprietary:** Controlled by a single organisation.
- **Consensus:** Developed collaboratively by multiple stakeholders.

Properties of Good Standards

- **Unambiguous:** Avoid misunderstandings and errors.
- **Flexibility:** Can optimise programs without violating the standard.
- **Machine Independent:** Not tied to specific hardware.
- **Relevant:** Developed and adopted quickly to keep up with technological advancements.

Assemblers

Translates assembly code into machine code, enabling programmers to write human-readable code while still having low-level control over hardware.

Advantages of Assemblers

Enables the use of assembly, so useful in scenarios where:

- Maximum performance is crucial.
- Need to directly manipulate hardware resources.
- Resources are limited.

Disassemblers

Translates machine code into assembly code, enabling programmers to read and understand low-level operations.

- Useful for debugging and reverse engineering.

Macros

Instructions which can be invoked using a single command. They **replace one text string with another** and enable automation of repetitive tasks.

e.g. Lisp macros & C preprocessor directives.

Uses

- **Text Editors:** Automate complex text manipulation.
- **Spreadsheets:** Automate calculations, formatting and data manipulation.
- **Software Development:** Repetitive code patterns.

Properties of Programming Languages

Properties of Effective Languages

- **Clear and simple syntax.**
- **Support for abstraction:** Enables function creation.
- **High portability:** Programs can be used on different systems.
- Facilitates **easy development and maintenance**.

Influences on Design

- The hardware and software that will use them.
- **Standardisation:** Functions the same across devices.
- Programming and implementation methods.

Orthogonality

The degree to which we can combine a language's built-in components without side effects. High orthogonality means components don't interfere with one another.

- Lack of orthogonality limits the language's expressive power.
- Independent parts are easier to maintain.

Language Differences

Low-level	***High-level***
Minimal abstraction from hardware, closer to machine code. • **Direct Hardware Control:** Programmers can access memory and CPU registers. • **Performance Optimisation:** Enables highly efficient code fine-tuned to architecture. • Requires **understanding of the specific architecture.** • **Smaller,** so takes less time to translate to machine code.	Greater abstraction from hardware, further away from machine code. • **Less hardware control.** • **Less performance optimisation.** • Requires **little understanding of architecture.** • **Easier to learn,** as code is more alike to English.

Compiled	***Interpreted***
The entire program is translated by the compiler at once.	Code is translated and executed line-by-line.

• Code can **only run once it has all been compiled**. • **Run faster.** • **Harder to write and debug.**	• Code must be **interpreted** every time it's run. • **Better for writing and debugging programs.**

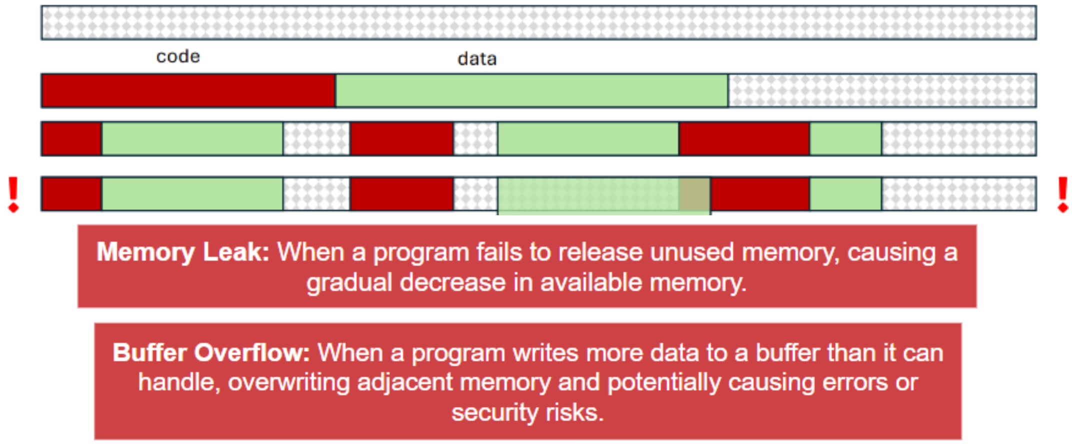
Strongly Typed	***Weakly Typed***
Variable types are strictly enforced so cannot be changed without explicit conversion. • **Reduces errors.** • **Less flexibility.**	Variable types are not strictly enforced so may change without explicit conversion.

Call-by-Value	***Call-by-Reference***
C always uses pass-by-value. When passing-by-value to a function: • The **value of the variable** is passed. • A **copy of this value is created at a different memory address**. • Any modifications will be applied to this copy. • When the function returns, the copy will pop meaning that the old value remains.	When passing-by-reference to a function: • The **address of the variable** is passed. • Any modifications will be **applied to the real value**. • When the function returns, the original value will have changed. This can be achieved in C **using pointers.** You must: • Pass the address of the variables to the function. • Initialise these addresses as pointers, meaning that C interprets them as the contents of these addresses.

Memory Management

Effective memory management means allocating and freeing it at the right time, **without overwriting the program or data, while keeping them separate.** Incorrect memory management can lead to:

- Crashes.
- Security vulnerabilities.
- Unpredictable behaviour.



Static Storage Allocation

Memory is **assigned at compile time** and remains **fixed** throughout the program's execution.

Dynamic Storage Allocation

Memory is **assigned at runtime** as needed and can be **freed or resized** during the program's execution.

Automatic Memory Management

Garbage collection automatically allocates memory by tracking objects no longer in use and reclaiming their memory.

- Reduces the risk of memory leaks and buffer overflows but can impact performance.

e.g. Python and Java

Manual Memory Management

Programmers must allocate and deallocate memory themselves.

- Increases control but also the risk of memory leaks and buffer overflows.

e.g. C and C++

Parallelism and Concurrency

- **Parallelism:** Executing multiple processes or threads simultaneously to increase performance and efficiency.
- **Concurrency:** Switching between tasks to give the appearance of simultaneous execution.

Used in high performance computing, web servers and volume rendering.

Programming Paradigms

A paradigm is **a style of programming or design aspect of a programming language**, defined by certain concepts and practices. They provide **different ways to organise code**, enabling different approaches to problem solving.

- Facilitate large systems development.
- Some languages support multiple paradigms but may discourage you from using one by making it difficult to use.

Imperative

Describe the steps needed to achieve a goal. Execution flow is controlled using loops, if statements, etc.

- Not mutually exclusive with other paradigms. *E.g. a language can be object-oriented and imperative.*
- Examples: C & Java.

Procedural

Carry out actions and calculations used in a **logical step-by-step process** for solving a problem. Code is **structured around functions/procedures** which perform tasks by operating on data.

- **Imperative.**
- Encourages a linear, structured approach.
- Typically **used in large, complex systems** where procedures are run at various stages of execution.
- Examples: C & Python.

Object-Oriented

Centered around the concept of classes and objects. They aim to organise software design around data and objects, rather than functions and logic.

- **Imperative.**
- **Supports parallel development** through multi-threading.
- **Classes:** A blueprint for objects defining their attributes and methods.
- **Objects:** An instance of a class that brings it to life with specific data.
- **Encapsulation:** Data and the operations on it are bundled into one unit, a class, to protect its integrity.
- **Inheritance:** Subclasses can inherit attributes and methods from superclasses, promoting code reuse.

- **Polymorphism:** A method can work differently depending on the class that runs it or the data it is given to work with.
- **Abstraction:** Complex implementation details are hidden, exposing only necessary parts. Reduces complexity, allowing for efficient design and implementation.

Unstructured

Don’t enforce or support a structured approach so lack organisation.

- **Imperative.**
- More common in early computing, so often found in legacy code.
- Typically used in small, simple programs.
- Code is **harder to read**, maintain and debug.
- **Unrestricted Flow Control:** Programs can jump to any part of the code, making them complex and difficult to follow.
- **Lack of Modularisation:** No functions.
- Uses *goto* statements.

Declarative

Describe what they want to achieve rather than how to achieve it, which is determined by the system itself.

- Not mutually exclusive with other paradigms. *E.g. a language can be functional and declarative.*
- Example: SQL.

Functional

Treat computation as the **evaluation of mathematical functions**, emphasising immutability and pure functions. They structure code recursively using higher-order and first-class functions, meaning functions can be passed as arguments, returned and assigned to variables.

- **Declarative.**
- **Supports parallelism and concurrency.**
- Code is **more predictable** and **easier to test**.
- Typically **used in applications that require parallel processing and immutability**, such as financial services, data analytics and real-time processing.

Logical

Define facts and rules about problems. Instead of detailing steps to prove them, the programmer specifies relationships and conditions. Then, the system figures out how to achieve the desired outcomes.

- **Declarative.**
- Typically **used for reasoning and pattern matching**, such as NLP and exepert systems.

Language Types

These languages are categorised by their use cases and purpose, rather than style.

Scripting

Designed for **integrating and communicating with other languages**.

- Can be imperative or declarative.
- Often interpreted.
- Typically used to automate tasks, manipulate data, or control and glue together other software components.
- Examples: Python, JavaScript and Bash.

Markup

Define the structure, presentation, and semantics of text through commands.

- **Declarative.**
- Often interpreted.
- Typically used in web development or documents.
- Examples: HTML, LaTeX.

Programming Languages

Assembly

A low-level programming language **specific to a particular computer architecture**. It allows programmers to write instructions that directly control hardware.

- **Difficult to Read:** One assembly instruction corresponds to one machine code instruction.
- **See advantages of low-level languages.**

Uses

Embedded systems and where performance is critical, such as operating systems and drivers.

Example

```
        LDR R0, 150
        LDR R1, 151
        ADD R2, R1, R0
        CMP R2, #10
        BLT end
        MOV R2, #10
end:
        STR R2, 152
        HALT
```

Fortran (Formula Translator)

Designed by IBM in the 50s to perform scientific calculations. It was the first high-level language to become widely used and is still being updated and used today.

- **Procedural.**
- Only pass-by-reference.
- **Efficient:** Optimised for IBM's computer.
- **Portable.**
- Early versions had static storage allocation and no recrusion.
- **COMMON data areas** allowed different program units to share variables.
- **EQUIVALENCE statement** allows different variables to share the same memory location.

Uses

Scientific computing and engineering.

Example

```
SUBROUTINE FACTOR ( N )
Factor N
Test for factors of 2,3,5,... using NEXTPRIME function to get next prime.
INTEGER N, NUMBER, PRIME
NUMBER = N
PRIME = 2 !Start with 2
CONTINUE
          IF (NUMBER .EQ. 0) RETURN
          IF ( MOD(NUMBER,PRIME) .EQ. 0 ) THEN !prime is a factor
```

Lisp (List Processing)

Developed in the 50s for list processing. It was influenced by the λ-calculus.

- **Functional.**
- **Minimalistic.**
- Automatic memory management.
- Treats data structures as a whole.

Uses

- Artificial intelligence.
- Symbolic computation & list processing.

Example

```
(defun prime(n)
 (and
   (> n 1) ;;picky case so we don't say numbers <=1 are prime
   (forall (nums 2 (floor (sqrt n)))
       #'(lambda (divisor) (not (= (mod n divisor) 0)))
       ))))
```

COBOL (Common Business Oriented Language)

Released in 1959 for business, finance and admin systems. Still being updated and used.

- **Procedural.**
- **Easily to Read & Understand:** Uses nouns and verbs.
- **Efficient Data Handling:** Can handle large data values and batch processing.
- Complete separation of data descriptions from commands.

Uses

- Banking & finance.
- Government Agencies: public records and admin.
- Legacy systems.

Example

```
SOURCE-COMPUTER. SUN.
OBJECT-COMPUTER. SUN.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT INP-DATA ASSIGN TO INPUT.
        SELECT RESULT-FILE ASSIGN TO OUTPUT.
DATA DIVISION.
```

ALGOL (Algorithmic Language)

Developed in the 50s for scientific computation and **introduced many key concepts which had a massive impact on language design**. But it was missing several features so was overtaken by Fortran.

- **Procedural.**
- Recursion.
- **Portable.**
- **Structured Programming:** Blocks of code and structured control flow.

Uses

- Algorithm description.
- Scientific computing.

Example

```
begin
           comment Algol program print the primes less than 1000
                      using the sieve method.;
           Boolean array sieve[2:1000];
           integer p, count;

           comment Eliminate the multiples of the argument prime number;
           procedure eliminate(p);
```

BASIC (Beginner's All-Purpose Symbolic Instruction Code)

Designed in the 60s to be an easy-to-learn programming language for beginners.

- Early versions were **unstructured.**
- **Easy to Use:** Simple syntax.
- Line numbers and *goto* statements.

Uses

- Teaching programming.
- Prototyping.

Example

```
10 HOME
20 PRINT "WHAT IS YOUR NAME?"
30 INPUT N$
40 PRINT "HELLO, " N$
50 END
```

C

A **general-purpose programming language** developed in the 70s known for its efficiency and performance. One of the most influential languages but it’s challenging to use as it **highlights programming difficulties, hence promoting good practise**.

- **Procedural.**
- **Low-level Access:** Memory can be manipulated using pointers.
- **Modularity:** Supports header files and libraries for modular programming.
- Statically typed.

Uses

- Embedded systems.
- Operating systems.
- High-performance computing.
- App development.

Prolog (Programming in Logic)

Developed in the 70s and focused on defining problems using logical relationships, facts and rules rather than step-by-step instructions.

- **Logical.**
- Computations expressed as queries.
- **Pattern matching.**
- **Logical inference.**

Uses

- Theorem proving.
- Expert systems.
- Natural Language Processing (NLP).

Example

```
% Facts:
man(socrates).

% Rules:
mortal(X) :- man(X).

% Queries and Answers:
?- mortal(socrates). % True, Socrates is mortal.
?- mortal(X). % X = socrates, meaning Socrates is mortal.
```

Ada

Designed in the 80s.

- Started as **procedural** but now multi-paradigm.
- Built-in support for **parallel computing**.
- **Exception handling.**

Uses

Long-lived applications where safety, reliability, and maintainability are critical, such as real-time and embedded systems

Example

```
with Ada.Text IO;           use Ada.Text IO;
with Ada.Integer Text IO;   use Ada.Integer Text IO;

procedure Prime is
        Number : Integer;
begin
        Put("Enter an integer : ");
        Get(Number);
        if Number < 2 then
                Put("The value "); Put(Number, 0); Put Line(" is bad.");
        else
                Put("The value "); Put(Number, 0);
                for I in 2 .. (Number-1) loop
                        if Number rem I = 0 then
```

Haskell

Developed in the 90s **based on the λ-calculus**.

- **Functional.**
- **No Statements, Only Expressions:** Every piece of code evaluates to a value.
- **No Side-Effects:** Functions do not alter the state, making predictable and reliable code.

Uses

- Academic research.
- Finance.

- Cryptography.

Example

```
primes :: [Int]
primes = sieve [2..]
 where
  sieve (x:xs) = x : sieve (filter ((/=0) . (`mod` x)) xs)
```

Python

A high-level language developed in the 90s known for its simplicity and readability.

- **Multi-paradigm.**
- **Interpreted.**
- Dynamically typed.
- Extensive libraries.

Uses

- Web & software development.
- Data science & machine learning.
- Automation.

Java

A high-level language developed in the 90s known for its portability and robustness.

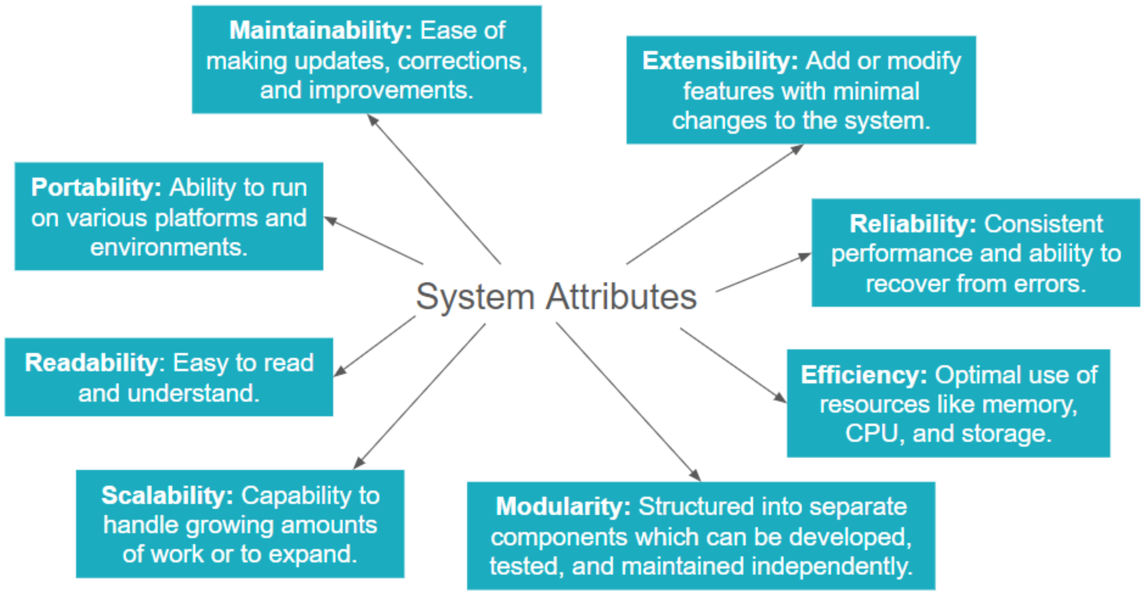
- **Multi-paradigm.**
- **Platform-Independent:** Runs on any device with the Java Virtual Machine (JVM).
- **Parallel Processing & Concurrency:** Multi-threading.
- **Automatic memory management.**

Uses

- Web & software development.
- Enterprise applications.
- Scientific computing.

Coding to High Standards

Coding professionally means developing a system which prioritises certain attributes, depending on the purpose. These include:



Coding professionally also means maintaining good practise. You can uphold these attributes to a high standard by:

- Employing **safe coding practises**.
- Creating thorough and clear **documentation**.
- Designing well-structured **functions and libraries**.
- Handling **input and output** processes effectively.
- Managing **dynamic memory allocation** responsibly.
- Implementing robust **error handling**.
- Conducting **rigorous debugging**.
- Developing **code incrementally**.
- Writing **simple code**.

Software Maintenance: The **largest contributor to development costs**. So, writing robust code reduces costs by minimising the need to debug in the future.

Procedure Method Hierarchy: A concept used to **structure procedures in a hierarchical manner** across programming languages. It enhances readability, maintainability, and reusability.

Dependencies: External libraries or packages that a program relies on to function correctly.

Functions

a.k.a subroutines or procedures, are **reusable blocks of code designed to perform one specific task**. They can be called with inputs (parameters) and may return an output.

- **Interface Design:** Carefully consider the parameters and return values of your functions. How it handles these defines its interface with the rest of the code.

Library Design

Good library design prioritises:

- Readability & maintenance.
- **Flexibility:** Can be used in a wide range of circumstances.
- **Futureproofing:** Will remain useful over time.

Environments

The settings and context in which code runs, including:

- Variables.
- Dependencies.
- Libraries.

Installing multiple packages in one environment can lead to conflicts between them. You may also want to use different versions of the same package for different projects. **Virtual environments** allow you to create custom environments for each project.

Coupling

The **degree to which modules are inter-related**.

- **Loosely coupled if modules are mostly independent** and interact through interfaces.
- Loose coupling **makes the codebase easier to work** with because changes in one module are less likely to impact others.

Cohesion

The **degree to which the elements of a module belong together.**

- **High cohesion means elements are strongly related** and have one purpose.
- High cohesion leads to better readability, maintenance and reusability.

Aim for **low coupling** and **high cohesion!**

Minimum Viable Product (MVP)

A stripped-down version of a **product with core features necessary to satisfy users**. Allows developers to **gather feedback** and validate the product's potential before investing more resources.

Proof of Concept (POC)

A prototype designed to **test if an idea is feasible**. Confirms the **validity of the concept** before committing to full-scale development.

Developing GraphBox

GraphBox should allow users to draw pictures by creating coloured outlines of multiple polygons in a pixel array, called a frame buffer.

Steps

1. Produce a **requirements specification** detailing all functions which the system will achieve.
2. Design the system as an **MVP** or **POC**, including a list of the components which need to be developed.
3. **Build the Absolute Minimum:** Develop the most essential features first.
4. **Incremental Development:** Incrementally add features until the MVP is complete.
5. **Test & Use:** Ensure it functions correctly, meets the requirements and gather user feedback.
6. **Continue incrementing** to improve the system based on feedback.

Following an **incremental development process** allows us to regularly **test our code** before it gets too complicated, **reducing** the amount of **debugging** needed.

The Starting Point

Start every project with the same code, to test the compiler is working:

```
#include <stdio.h>

int main(int argc, const char * argv[]) {
    printf("hello world\n");
    return(0);
}
```

After this, you should add **header comments** to the top of the file briefly describing;

- What the code does.
- How to run it.

Scaffolding Code

Scaffolding code is **temporary code that helps in the development and debugging process**. It helps to track the program's execution flow and diagnose issues.

fprintf statements can be used as debug statements and should be accompanied by comments. They should outline:

- The start and end of the code.
- The key sections.

Example

```
int main( int argc, const char * argv[] )
{
   fprintf( stderr, "GraphBox: begin\n"); // begin, a debug statement

  // shapes
  fprintf( stderr, "\tmake shapes\n"); // debug code
```

Creating the Framebuffer

The frame buffer is a 2D array of doubles. We want to allocate this dynamically, so we use *malloc*:

```
double **fbuff_malloc( int hgt, int wid ) {
    double *array = (double *)malloc( hgt*wid*sizeof(double) );
    double **fbuff = (double **)malloc( hgt*sizeof(double *) );
    int i;

    fbuff[0] = array;
    for (i = 1; i < hgt; i++ ){
      fbuff[i] = fbuff[i-1] + wid;
    }

    return(fbuff);
}
```

- *array* is a 1D array holding all the doubles.
- *fbuff* is a 1D array containing pointers to each row in the frame buffer.

Users may also want to deallocate the memory used for the frame buffer using *free*:

```
int fbuff_free( double **fbuff )
{
  free(fbuff[0]);
  free(fbuff);
  return(1);
}
```

Creating the Framebuffer Library

1. Develop the framebuffer code in the main file.
2. Move it to a separate file.
3. Compile it into an object file:
   gcc –c fbuff.c
4. Create a header file for the framebuffer code, specifying:
   a. Function names, return types and parameters.
   b. Macros & types.
5. Import the header file at the top of the main file using a preprocessor directive:
   #include "fbuff.h"

6. Compile the main file:
   gcc –c main.c
7. Link the main file and the library file.
   gcc fbuff.o main.o -o myprogram

Defensive Programming

Prevents your program from crashing by anticipating and handling errors or unexpected inputs.

- Minor errors, which can easily be dealt with, can cause interruptions resulting in lost data.
- Crucial when dealing with unsecure inputs.
- You should proactively anticipate exceptions in the development stage.

Advantages

- Enhances **system integrity** & user experience.
- Simplifies troubleshooting.
- Prevents data loss.

Upstream: Earlier data, actions, or function calls leading to a point.

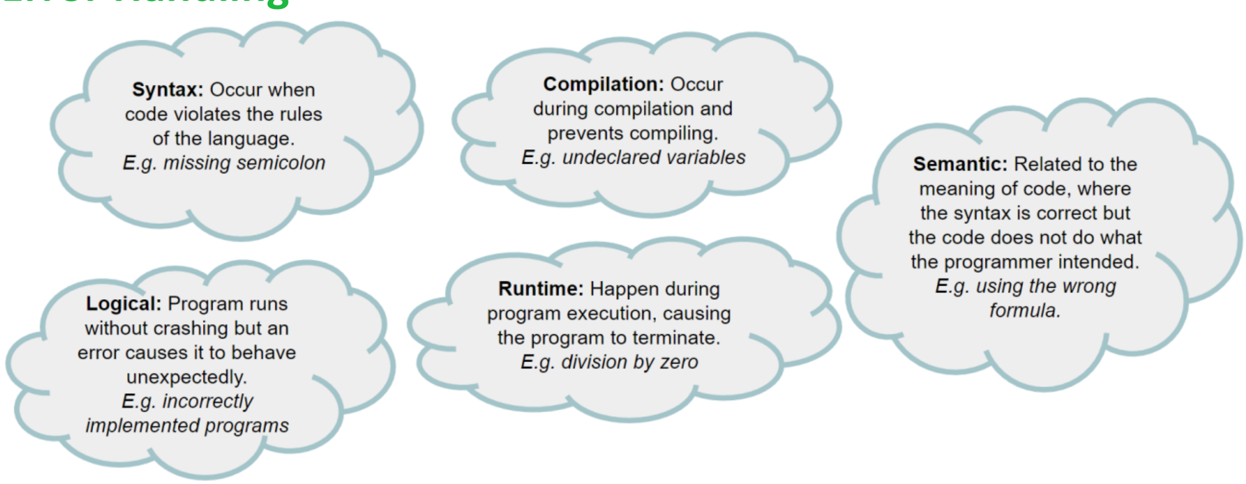
Downstream: Subsequent stages or results affected by a point.

Tracebacks

Follow the **call stack** to find the origin of an error, including:

- **Function Calls:** Sequence of calls that led to it, starting from first (upstream) to where the error occurred (downstream).
- **File names, line numbers,** and **code snippets** for each call in the stack.
- **Traceback Chain:** Illustrates the sequence of exceptions which caused the crashing exception.

Error Handling



1. **Write & Throw** an Exception: Can be done automatically.
2. **Catch & Handle** the Exception.
3. (Optional) **Log** to a file.

The GPU Cloud

Provides **access to GPUs over the internet**, enabling tasks which require **high-performance parallel processing**, such as machine learning, 3D rendering, and scientific simulations.

When using a GPU cloud, you should install all dependencies.

Key Terms

- **GPU Cluster:** A network of multiple GPU-equipped machines parallel processing.
- **Diffusion Model:** Generate high-quality images using a reverse noise-adding process which starts with noise and refines until it creates a realistic image.

Sessions and Jobs in Linux

- **Session:** Collections of processes controlled together, often started when a user logs in. Allow grouping related processes for easier management and control, all share the same session ID and close when a user logs out.
- **Job:** Processes initiated by a user from the shell. All jobs have a session.

You can prevent jobs closing when their session does using the *nohup* command or by **screening a virtual session**.

C Programming

See Obsidian notes.