# Software Development

# Software engineering life-cycle

- IEEE Standard 1074 for Software Lifecycles
- Sequential models
  - waterfall model : standard phases (req., design, code, test), in order - linerar
  - V-model: Also known as Verification and Validation model, non-linear
- Iterative models
  - Unlike sequential models, step-by-step process of the development stages,
  - these are cyclical process. After an initial planning phase, a small handful of stages are repeated over and over, with each completion of the cycle incrementally improving and iterating on the software.
  - Advantage: Enhancements can quickly be recognized and implemented throughout each iteration,
  - Example - spiral model : assess risks at each step; do most critical action first

# Lifecycle stages

- The key lifecycles steps are:
  - Requirements
  - Design
  - Implementation
  - Testing
  - Maintenance

# Requirements

- Requirements are constantly changing
  - The client might not know all the requirements in advance
  - Frequent changes are difficult to manage
  - Identifying checkpoints for planning and cost estimation is difficult
  - There is more than one software system
  - New system must often be backward compatible with existing system (“legacy system”)

# Design

- **Architectural Design** –Top level: It identifies the software as a system and how its components interact with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design-** Intermediate level: The high-level design breaks down the architectural design into a less-abstracted view of sub-systems and modules and depicts their interaction with each other. It focuses on how the system with all of its components can be implemented in a modular structure.
- **Detailed Design-** Lowest Level :focused on detail of the modules and their implementation. It defines a logical structure of each module and their interfaces to communicate with other modules

# Implementation

- Building of software units to satisfy module specifications.
- Assembly, integration, and testing of software components into a software configuration item.
- Prototyping software components to resolve implementation risks or establish a fabrication proof of concept.
- Dry-run acceptance testing procedures to ensure the software product is ready for acceptance testing.

# Testing

- Unit Testing. Unit testing is the first stage of software testing levels.
- Integration Testing. Testers perform integration testing in the next phase of testing.
- System Testing.
- Acceptance Testing.
- Requirement Analysis.
- Software Testing Planning.
- Test Case Development.

# Unit Testing

- Testing of software units to satisfy module  specifications.
- Testers evaluate individual components of the system to see if these components are functioning properly on their own.
- Each of the units selected is then tested to check whether or not it’s fully functional. To perform this stage successfully, the tester needs to have knowledge about granular levels of detail.

# Integration Testing

- Test of individual components of the system  as a collective group.
- This is to identify any problems in the interface between the modules and functions.

# System Testing

- final stage of the verification process.
- Testers see whether the collective group of integrated components is performing optimally
- verifies that the application meets the technical, functional, and business requirements specified by the customer.

# Acceptance Testing.

- Evaluate if the application is ready to be released for user consumption.
- Identifies any misunderstanding of business requirements and delivers the product that your customers want.

# Requirement Analysis

- To evaluate the requirements of testing and outline which of the given requirements they can test
- Need to
  - outline the types of tests needed for testing,
  - gather information about the priorities in testing
  - configure a traceability matrix for requirements.

# Software Testing Planning

- Generate test plan for different types of software testing, select which testing tool is optimal, and evaluate effort estimation.

- Assign responsibilities and roles to their team.

# Test Case Development

- creation of test cases and their corresponding scripts.
- The test team needs to create, verify, and remake specific test cases, based on specific features and their requirements

# Environment Setup

- Test conditions, such as hardware and software specifications used during the testing procedure.
- Ideally, this should imitate the environment used by the end-user in his/her working space.

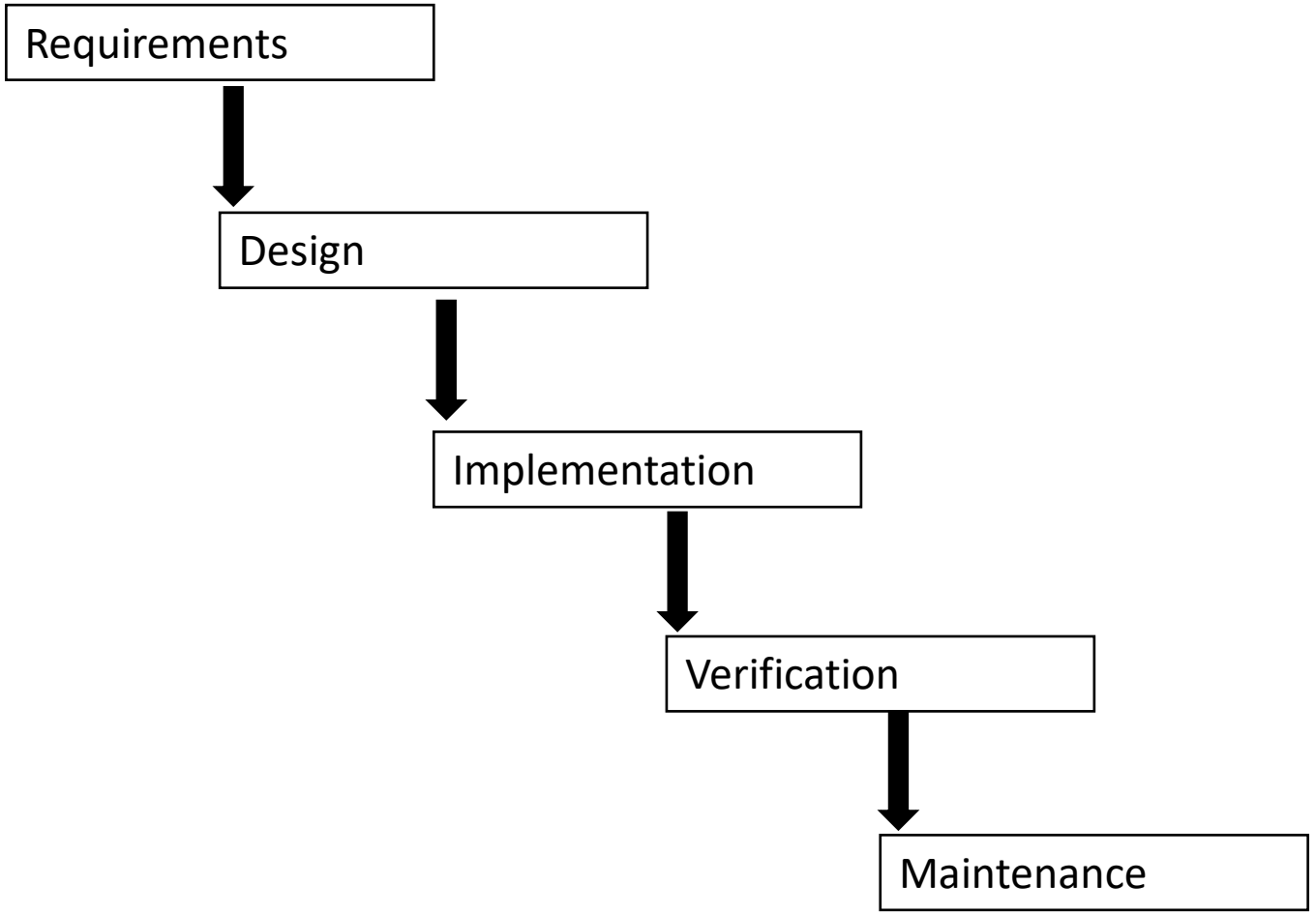
# Software Testing Execution

- Testers carry out testing according to the test plans and test cases created by the team.
- They will evaluate if all the requirements are met
- Document all test results and log any case that have failed.
- They need to map the bugs with the test cases

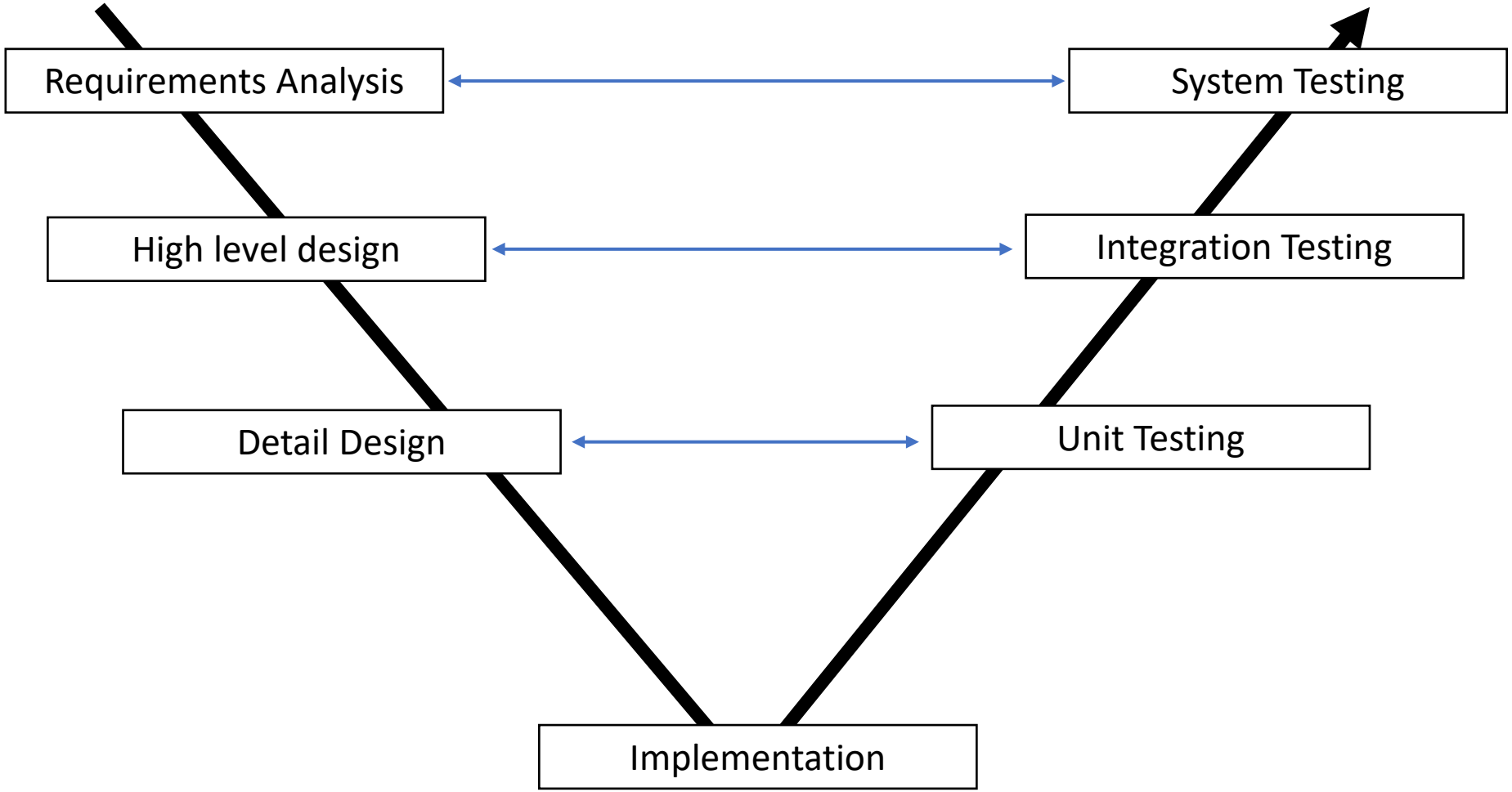
# Example of Testing Table

| Test Number | Test Type | Test input data | Testing | Expected Outcome | Outcome | Pass/Fail |
|---|---|---|---|---|---|---|
| 1 | Validation | X = 10 | Evaluate root based on starting value | Root = –0.05 | +/- 1.0056 | Fail |
| 2 | Range test | X_lower = 0<br>X_upper = 1.5 | Test if known minima found | Min_y = 0 | Min_y = 0 | Pass |
| 3 | Invalid input | X = a | Testing if exception handling picks up wrong data type | New Input request | Application terminated | Fail |

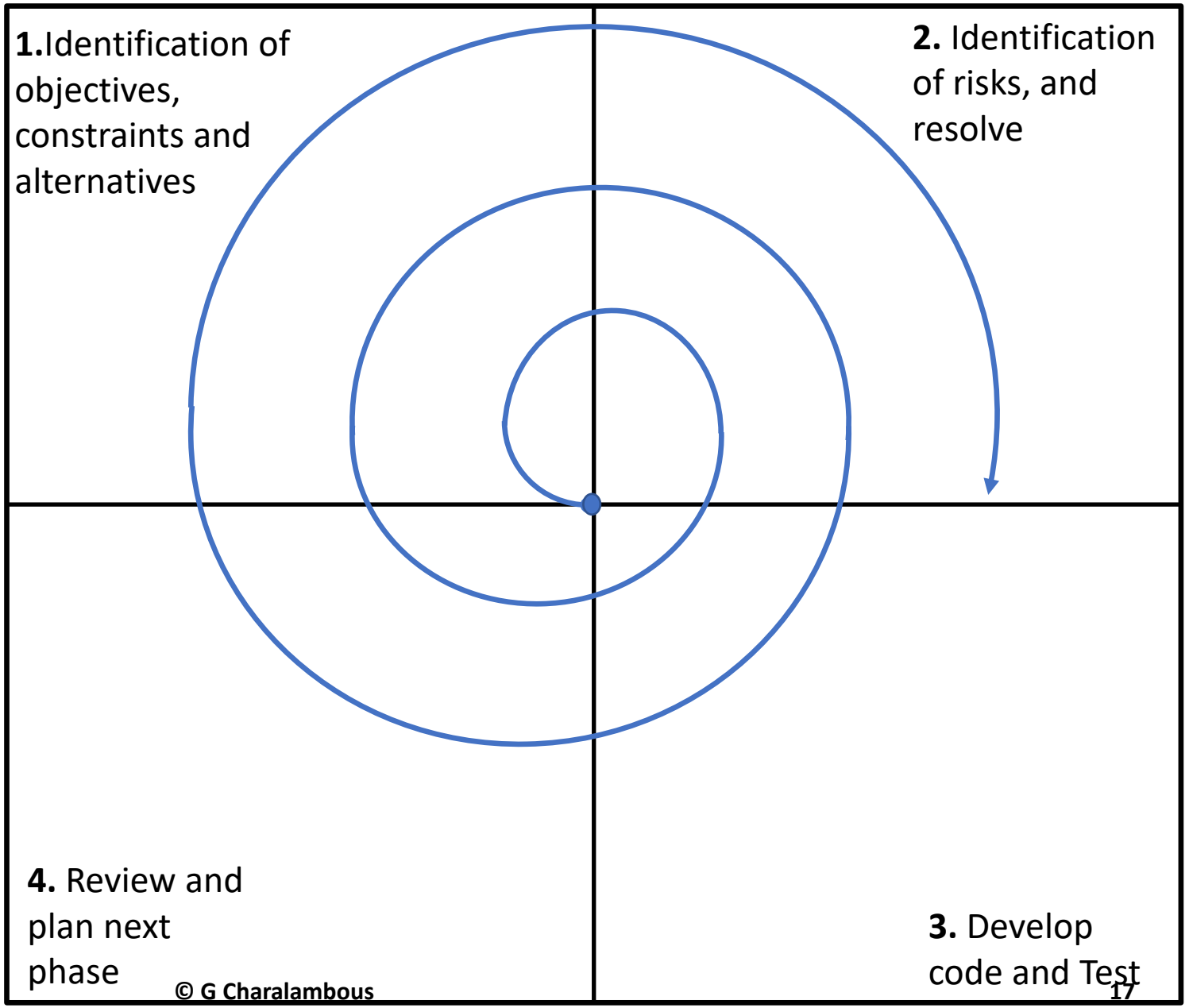
# Waterfall Model



# V-model



# Spiral model



# Need for Programming Languages

- 'Natural' languages (e.g., English, Arabic, Chinese, etc)
- have very large, dynamic vocabularies
- - English has 500,000 - 1,000,000 words depending upon which source you take, and is constantly changing.

- have complex and non-standardized grammar
- - there is constant debate about what is 'correct' English.
- This is made worse (from a machine translation point of view) in that sentences may be well-formed but meaningless.
- - For example:
- “Colourless dead green sheep dream furiously.”

- can be ambiguous
- - the following sentence has several interpretations:
- “I saw a man on a hill with a telescope.”

# Computer Languages

- Computer languages compared to natural language:
- have a very  small vocabulary compared to natural language

- - most have less than 100 reserved words  - the C language has a lexicon of just 32 reserved words

- have  a very simple syntax - usually defined using Backus-Naur Form (BNF)
  - BNF is a formal way to present the syntax of formal languages, e.g.,  programming languages.

- have rigid rules regarding context

- For example, all programming languages have strict rules concerning the scope and lifetime of variables.

# Need for Programming Languages

- can be contradictory, e.g., - No it doesn't! (?)

- have problems with anaphora
- - For example, the sentence:  “It did it.”,
- requires contextual knowledge to determine its meaning

- can be difficult to translate between natural languages, i.e. “loses something in translation”.
- - For example, most poetry loses its effect in translation

# Compilers

- Purpose
- translate a program in a 'high level language'' into something a machine can execute

- report errors if the program is not well-formed, i.e., syntax errors.

- do not correct the semantics (meaning) of a program, i.e., does not identify or report the “logical errors”

# Translation types

- Single pass compilers
- multi-pass compilers
- debuggers
- optimizers

# Compiler Tasks

- Analysis & Synthesis
- break source program into logical units
- create intermediate representation
- construct target (executable) program from intermediate representation
- tools include language-specific compilers

# Front-End (Machine-independent) Activities

- interpret pre-processor commands
- focus on source language
- lexical, syntactical & contextual analysis
- creation of  a 'Symbol Table'
- intermediate code generation
- error handling

# Back-End Activities

- focus on target language (and machine)
- code generation into an assembler
- code optimization
- linking
- loading

# C/C++ Pre-compiler tasks

- expand the text of all #include statements
- substitute the value of all #define constants and macros
- discard unwanted text for conditional #ifdefs
- error report impossible #-directives

- e.g., non-existent #include files

# Lexical (linear) Analysis

- scans a stream of characters
- discards 'white-space'  - ' ', tab, NL, ...
- comments
- converts rest into a 'token' stream
- for many languages done a character at a time from left to right (LR1 analysis)

# Tokens

- a token will represent a sequence of characters treated as a single unit by the source language

- predefined token types will represent keywords, operators, punctuation, etc  examples: ID, REAL, IF, COMMA, WHILE,

- token instances will be generated for names

- e.g., variables, classes, functions, etc

- these names are entered in a Symbol Table

# Example code extract & token stream

Source Code:

```
/* find a zero */
 bool match(char* s)
 {
    if (!strcmp(s, "0.0", 3))
    {
       ...
 }
```

Token Stream:

- BOOL ID(match)
- LPAREN
- CHAR STAR ID(s)
- RPAREN
- LBRACE
- IF
- LPAREN
- EXCLAIMATION
- ID(strcmp)
- LPAREN
- ID(s)
- COMMA
- STRING(0.0)
- COMMA
- INT(3)
- ....
- RBRACE
- EOF

# Sample Symbol Table

The following extract shows:

- the names x and y defined in main()
- the name f used in main() and defined elsewhere
- the name x defined and used in f()
- the name z used in f()

| Name | Block | Type | Info | Location |
|---|---|---|---|---|
| x | main | INT | CONST | 100000 |
| y | main | CHAR | ARRAY | 100004 |
| f | main | INT | FUNCT | 100200 |
| f | f | FUNCT | INT → INT | 200000 |
| x | f | FLOAT | VARIABLE | 200004 |
| z | f | INT | UNKNOWN | EXTERNAL |

- depending on the compiler/language more or less detail is stored
- details for z are waiting for link information
- locations are provisional and only relative to one another
- actual locations are only completed at load time

# Syntax Analysis

- also called parsing
- looks at clauses & sentences
- checks if grammar rules are followed
- if ok then creates abstract 'syntax tree'

# Contextual (semantic) Analysis

- checks if syntactic components fit
- checks if variables have been declared
- checks if types are correct and compatible
- checks if function arguments match expectations
- resolves laws of scope
- cannot check if a program does what it is meant to do!

# Example code extract & some contextual analysis tasks

```
int f (int x, int z)
 {
    return x + z + y ;
 }

 int g (int x)
 {
     int y ;
     return x +  f(x, y) ;
 }
```

- checks if f and g have the correct number of parameters
- checks if f and g have the right source and target types

- I.e. check they have the correct signature:

f : int × int → int

g : int → int

- distinguishes between the several uses of x and y
- ..etc..

# Backus-Naur Form (BNF)

- Computer languages are formally specified to remove ambiguity
- BNF specifies languages in terms of Regular Expressions (cf. The UNIX 'grep' command.)
- determines precedence of operators
- Some BNF regular expressions include:
  - a|b means a or b
  - [abcd] means a or b or c or d
  - a* means zero or many instances of a
  - "abc" means the 3-character string abc

# Backus-Naur Form (BNF)

- some token specifications:

  "if"  IF

  [0 - 9]+  INTEGER

- clauses are handled in a similar way
  - e.g., a function could be defined as:

  TYPE NAME LPAREN PARAMETERS RPAREN
  LBRACE FBODY RBRACE

- where PARAMETERS and FBODY would have already been specified.

# Code Generation and Optimization

- **Code Optimization**
- re-uses common sub-expression once only
- eliminates dead-end code (i.e., code that cannot be reached)
- uses machine registers
- converts single-assigned variables to constants
- optimizes loops
- and much more

# Code Generation and Optimization

- **Code Generation**
- resolves Symbol Table entries
- calculates relative addresses of variables
- produces target code
- and much more

# Assembler tasks

- checks assembly programs components  - e.g., labels, op codes & operands
- reserves memory based on memory directives - e.g., .BLOCK 256
- gives binary representation of other data formats - e.g., hex
- converts mnemonic op codes into machine code - e.g., SUBA d#1,i
- converts address modes to machine code - i, d, x, ????
- converts symbolic addresses into addresses - e.g., BR Loop1

# Linking and Loading

- Linking overview
- after compilation of the source code, external functions are incorporated to create the executable code

- linker attempts to handle unresolved Symbol Table entries

- e.g. if prog.cc uses a library routine then the Symbol Table entry will be resolved at link time

- missing or ill-formed (e.g., wrong type signatures) entries are reported as errors

# Linking and Loading

- Loading overview
- if everything can be linked then the program is ready to load

- loading occurs when the program is executed

- by converting virtual addresses into actual addresses

# programming languages

- Programming languages comes in many different flavors and
- They are designed to allow programmers to develop systems at different layers

- Hardware – programming use Assembler (use this as machine code is very difficult to read/ code)
- At the low level combine Assembler with C – most systems developed (kernel) using these.

- There are languages deigned for specific purposes e.g FORTRAN this is design to carry out mathematical programming

# programming languages

- At the higher levels we have object-based languages which focus on the problem, and we form definitions (classes) and build upon these definitions( inheritance) and create these structures in memory (objects) languages like VB, C#, Java, Python are object-oriented languages

- ALL codes generated in whatever programming languages used need to be translated back into machine code that the CPU can interpret and execute

# compilers, assemblers, linkers and loaders

- A popular language like Python/Java their code is translated and executed line by line – this form we describe as an interpreted languages
- These languages try to optimize, java you compile this into byte-code and then that code is executed line by line via the interpreter
- For these programs to run you need to have installed the interpreter to run a given application in that language
- These are slower in comparison to languages like C and Fortran

# compilers, assemblers, linkers and loaders

- C, Fortran, C++, VB are compiled
- These
- 1. test code for any syntax errors
- 2. translate the code in stages into object code
- 3. the object code are linked together to form the executable code
- Programs compiled on a machine into executable can run without the need of another program

# Debuggers and IDEs

- Most programmers do not like writing code directly just using a text file
- We have IDE's (Integrated Development Environments) which help the programmer develop code
- Visual Studio – is an IDE for multiple languages from C, C++, VB, C# etc..
- It will allow you to set up the libraries you need, it gives suggestions for different functions to use, it also pick ups the form of defined functions and highlights its parameters, their type and the functions return type.
- Python – has a shell interface as its IDE this is called IDLE; it's a little more basic than VS but requires less space/time

# Debuggers and IDEs

- Debuggers are built into many IDEs and this allows you to track logical problems in code

- IDES will have a debug mode so that you can put in markers for the execution to pause and display values

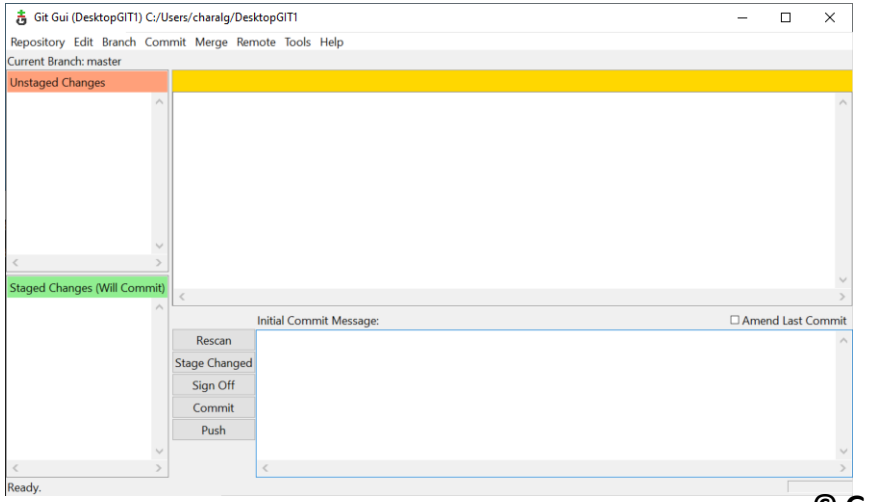
- Languages without IDEs like the gcc compiler here the C language allows you to build into the code print statements that will be compiled and executed based on flag settings. This is called conditional compilation
- And so the developer can build in testing and set flags that when the compiled with will print out specific variables  within the application.

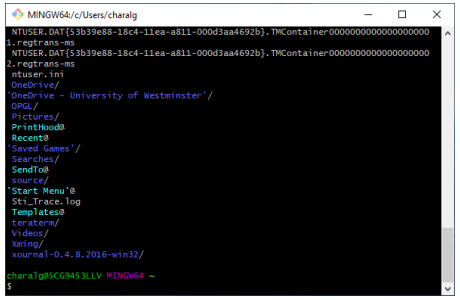
# version control

- When you develop a large application there will be many programmers developing the system
- It is important to maintain version control and track changes
- For example, Alice Bob, and Fred are working on a system
- Alice is the team leader and has assigned Bob and Fred modules to work on
- Bob is working on Module_34 and Fred is working on Module _98
- Both these modules are dependent on Module_22
- Say Fred decides to make changes to Module_22 without informing anyone
- Alice has an older version of Module_22 and she makes changes unaware of the adaptions made by Fred
- This can lead to serious problems for the team; so tracking changes is very important and managing these is what version control does.

# Git

- Allows if working on your own to go back to earlier versions of your code
- When working with others, simplifies changes, merging
- Good tool to know
- You can download and install from http://git-scm.com/downloads or use obtain it from https://appsanywhere.westminster.ac.uk/. This is installed on Linux
- When run in widows it will create a GUI Interface as well as a BASH shell to enter cmds



# git

- You will need to create a project directory
- Set up users
- Inside the project directory create a project repository – will have the extension .git
- Add files in project directory into the repository
- You can add comments
- You can close repositories ; all repositories are equal but you choose a master
- The team will each have their own repository; they can be merged with other

# git

- Users can commit files from project directory to repository
- The commit object contains
- (1) a set of files,
- (2) references to the “parents” of the commit object,
- (3) a unique “SHA1” value
- work on files in the working directory, the repository isn’t updated until commit file(s) is applied
- Two commit objects can be merged – so new commit has 2 parents
- This will create a directed graph which is used to manage the versions of the modules
- To get started: read http://git-scm.com/book/en/v2/Getting-Started-About-Version-Control