# Application Generation

**Types of Application and Utilities**

**Applications**

- Applications software is software that allows a user to perform a task or produce something. Could be:

- Word Processors.

- Spreadsheet Packages.

- Presentation Software

- Desktop Publishing Software.

- Image Editors.

- Web Browsers.

**Utilities**

- A utility is a relatively small program that has one purpose usually concerned with the maintenance of the city.

**Anti-Virus Software:** Viruses are malicious programs, anti-virus software detects and removes them from a computer.

**Disk Defragmentation:** Over time data can get split over lots of sections of the hard drive making it slow to access this information. Disk defragmentation groups files back together so that they can be read faster.

**Compression:** Reduces the amount of space data takes up in storage.

**File Manager:** Used to move, copy, delete and rename files.

**Backup Utilities:** Allow backups of certain data to be made automatically.

**Off-The-Shelf Software (General Purpose Application)**

- Off the shelf software means software that is purchased as a complete application from a commercial source as a finished product.

- Provides many features that the majority of users will use.

- Applications often quite large as they have a lot of stuff that’s unnecessary for an individual user.

- Cheap.

- Easily available.

- Thoroughly tested should be free of serious problems or bugs.

- Lots of user support.

**Custom Software**

- Software with a high amount of customisability, may involve an on-site visit by an expert for set-up.

- Usually more expensive than an off-the-shelf solution but more suitable for fields such as accountancy.

**Bespoke Software (Specific Application)**

- Some tasks are so specific that only a specifically written application will do. For example a production process for a factory or scientific equipment.

- Bespoke applications are written in-house by professional programmers or they can be sub-contracted to a software house.

- Most expensive option.

- Company will get exact software they need.

- Software will work exactly how they want it to work.

- Software will only have features they specifically need.

- Takes a lot of time to develop such a solution.

- Costs loads.

- New employees may be necessary, business analysts, programmers and testers.

- No external support for application.

**Open vs Closed Source Software**

**Open Source**

- Source code publicly available.

- Software is free.

- License allows code to be copied and modified.

- Less polished interface normally.

- Written by expert volunteers.

- Documentation and technical support limited.

- Arguably less secure as the source code can be deliberately examined for weaknesses and taken advantage of.

- Counter argument – popular open source applications have hundreds of individuals looking for security flaws in the source code and are quickly patched, but this relies on the fact people keep their software up to date.

- No enforced deadlines as code is written by volunteers.

- Lower cost of development.

**Open Source in Business**

- Business do this when it makes commercial sense to do so.

- Usually means open source applications are mature, stable and well supported.

- Non-technical businesses in general may not want the burden of updating and maintaining open source software so will pay for technical support from the company that makes the software or other companies.

**Closed Source**

- Source code is a trade secret.

- Application is not free.

- License restricts copying and modifying.

- Quite polished to attract new customers.

- Written by paid software programmers.

- Documentation usually good and support is formally available.

- Arguably more secure as the source code is not available and weaknesses are not so apparent.

- Many applications update automatically so fixes are propagated rapidly to customers without any technical knowledge.

- Company will have strict deadlines for programmers to make new releases.

- Higher development cost as programmers are paid.

**Applications Generation**

**Machine Code**

- Computers only understand machine code, consisting of just binary sequences split into instructions and data (opcodes and operands).

- For the first computers there was no choice other than to write in machine code.

- Would be error prone. Different processors had different **instruction sets** so the code was not portable. Programs need rewriting for each computer system.

**Assembler**

- Consists of mnemonics that are easier to remember than opcodes but which have a 1:1 relationship with the opcodes in an instruction set.

- An assembler converts assembly code into machine code.

- Essentially translates directly into binary word by word.

- Assembly isn’t very portable either.

**Compilers and Interpreters**

- A higher level language consists of more human readable code.

- Requires converting into machine code via either an interpreter or compiler.

**Interpreter**

- An interpreter takes each line of a high-level language program, converts it to machine code and runs it.

- This is useful when debugging as the program can start right away and stop when it encounters an error.

- The downside is that the program runs slower. The user has to wait for the translation of each line as well as the translation.

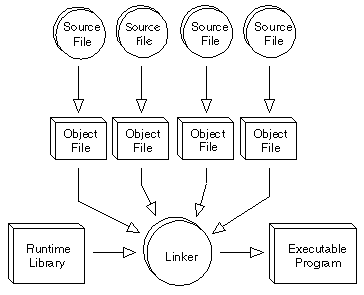
- The interpreter may have to translate the same line many times in a loop which is inefficient.

**Compiler**

- A compiler is a program that takes a program written in a high-level language and converts it to **object code.**

- The object code can then be distributed to anyone with a compatible system without the need for additional programs.

- Once the program has been compiled it can be run as often as needed faster than its interpreted equivalent.

- The object code is not human readable which strengthens anti piracy measures for commercial code.

- Compilation may take a while for large programs.

**Object Code**

- Often used interchangeably with machine code but object code is an intermediary step sometimes taken before pure machine code is produced.

- The object code contains place-holders where library code needs to go. Once a linker has been sued machine code that can be run is produced.

**How a Compiler Works**

- A compiler works by going through a sequence of stages, each moving closer to machine code. While the exact process varies between compilers. Most will include the stages:  
 - Lexical Analysis.

- Syntax Analysis.

- Code Generation.

- Optimisation.

**Lexical Analysis**

**Reserved Word**

A word that has special meaning in a programming language and as such cannot be used as a variable name. E.g: if, else, while and for.

- All comments and white-space are removed from the program.

- High-level code is turned into a series of **tokens** which represent **reserved words**, operators, variables and constants. Tokens are specific strings of characters.

**Syntax Analysis**

- The syntax of a language is the set of rules that govern its structure.

- Syntax analysis checks the code that has been written uses a valid syntax. Where code does not follow the rules of the language the compiler will generate a list of syntax errors to alert the programmer.

- Syntax analysis will produce an Abstract Syntax Tree (**AST**) that will represent the program. If the tokens do not fit into this structure then this means there is a syntax error.

**Code Generation**

- At this stage code is represented by an Abstract Syntax Tree. Code generation is when the compiler converts this into object code.

**Optimisation**

- Optimisation makes sure code runs as quickly as possible.

- There are a number of tricks the compiler can use to achieve this:

- Remove lines of code that have no effect on the program.

- It will look at instructions and see if they can be replaced by a more useful alternative.

- Optimisation happens during and after **code generation.**

**Modules**

- Many software projects produce large, complex programs. Because of this, it is often useful to break programs down into smaller tasks called modules.

- Using modules allows smaller chunks of code to be written and tested separately to the rest.

- Each module should be as independent as possible from the other modules. Helps to reduce chances of an error in one module adversely affecting another.

- Only when each individual module passes inspection is the program tested as a whole.

- Modules that are very independent are **loosely couples** whilst those that are dependent on one another are **tightly coupled.**

- When the modules have been written and checked the compiler translates each module’s source code into machine code.

- Machine code versions of modules are called **object files.**

**Libraries**

- Often code to perform complex tasks has already been written. This code can be reused by other programmers.

- Contains a number of useful procedures.

- It is best to do use libraries where possible. They are often designed to tackle a complex task such as graphics or cryptography which may otherwise be time consuming to write yourself.

- Programmers use libraries through an **API (Application Programming Interface)** a library may be written in one language and the have APIs designed to work in other languages.

- Specialist libraries can be bought as products in their own right.

**Pros and Cons of Libraries**

- No need to write some procedures from scratch which saves time.

- Procedures have already been tested, they are unlikely to contain errors.

- The application itself remains small and compact.

- Allows code to be shared with other applications which make use of the same procedures.

- An external library procedure can be updated without needing to re-compile the application.

- Library functions can be written in the most efficient language for the job.

- Library routines are connected to the program using a linker.

- The addresses of library routines are handled by the loader when the program is run.

- The library has to be well-written and robust or it will impair all applications making use of it.

- Specialist libraries for engineering, science and finance can be very expensive.

- For run-time loading the library has to be present.

**Compiling Libraries**

- When software is compiled, you have two choices. Integrate the libraries used into the final program or you can keep the libraries as separate files.

- Integrating libraries results in a standalone executable which can be run without access to libraries all the time.

- However the libraries will have to be loaded into memory with the rest of the code which slows down the process and uses more memory.

- Alternative is **run-time loading** in which the libraries remain as separate files and the core of the software is kept in its own executable.

- When the application is opened, it loads into memory but the library procedures are only loaded when they are called. This allows code to be loaded quicker and use less memory.

- Downside is that if libraries are missing, the code will not run.

**Linkers**

- At this stage in compilation, we have a group of object files and libraries. These need to be combined together to create a functioning program.

- The program that does this is called the **linker.**

- The linker is provided with a text file listing all of the object files and libraries that need to be connected together. The text file is called a **make file** and is generally produced by the compiler automatically.

- The result is an executable file – a machine code translation of the full program that can now be understood and carried out by the CPU.

**Loaders**

- After an executable file has been created, it has to be loaded into main memory in order to run it. This is done by a utility program within the OS called the **loader.**

- The loader is part of the memory segmentation role of the OS. It first creates a code segment to hold the program itself, then creates a data segment for the constants and variables and also a stack segment for handling procedure calls.

- The machine code of the executable file defines the memory addresses of each line of code relative to one another, allowing it to jump from one point to another.

- One of the duties of the loader is to translate these **relative addresses** into **absolute addresses.**