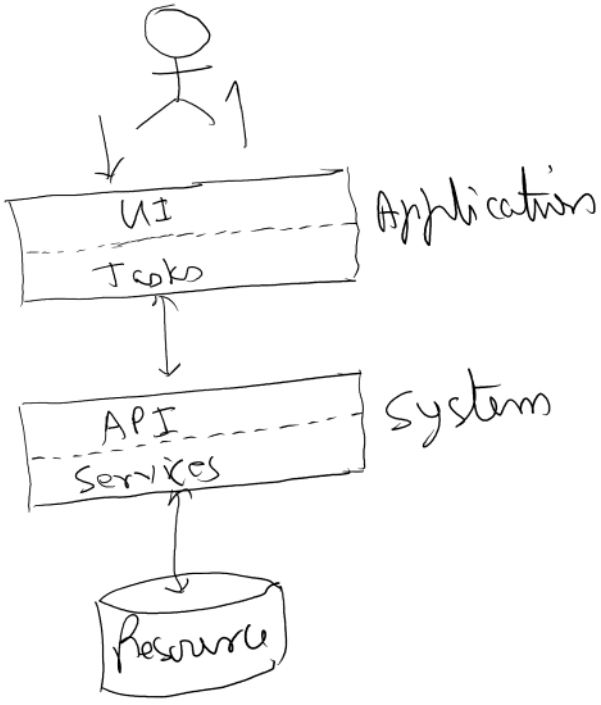
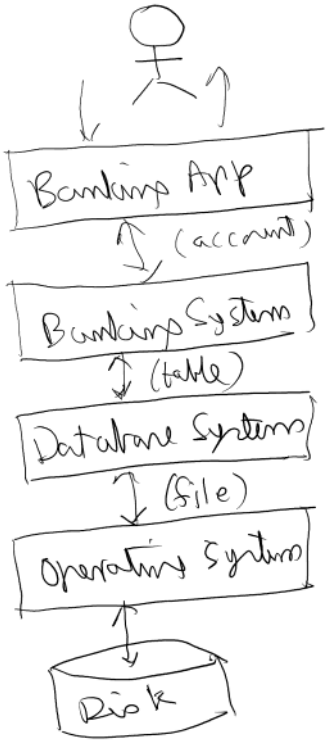
Basics

04 March 2022 08:55

Software: It is a series of *binary coded instructions* known as a *program* which can be loaded (from storage device into the memory) and executed by a *machine* to exchange and process some data according to a set of logical steps known as an *algorithm*. There are two types of software

1. **Application** - It implements *tasks* for processing data and provides a *user interface* (UI) allowing a human to execute those tasks.
2. **System** - It implements *services* for managing exchange of data with a *resource* (which supports reading/writing of data) and provides an *application programming interface* (API) allowing other software to consume those services.



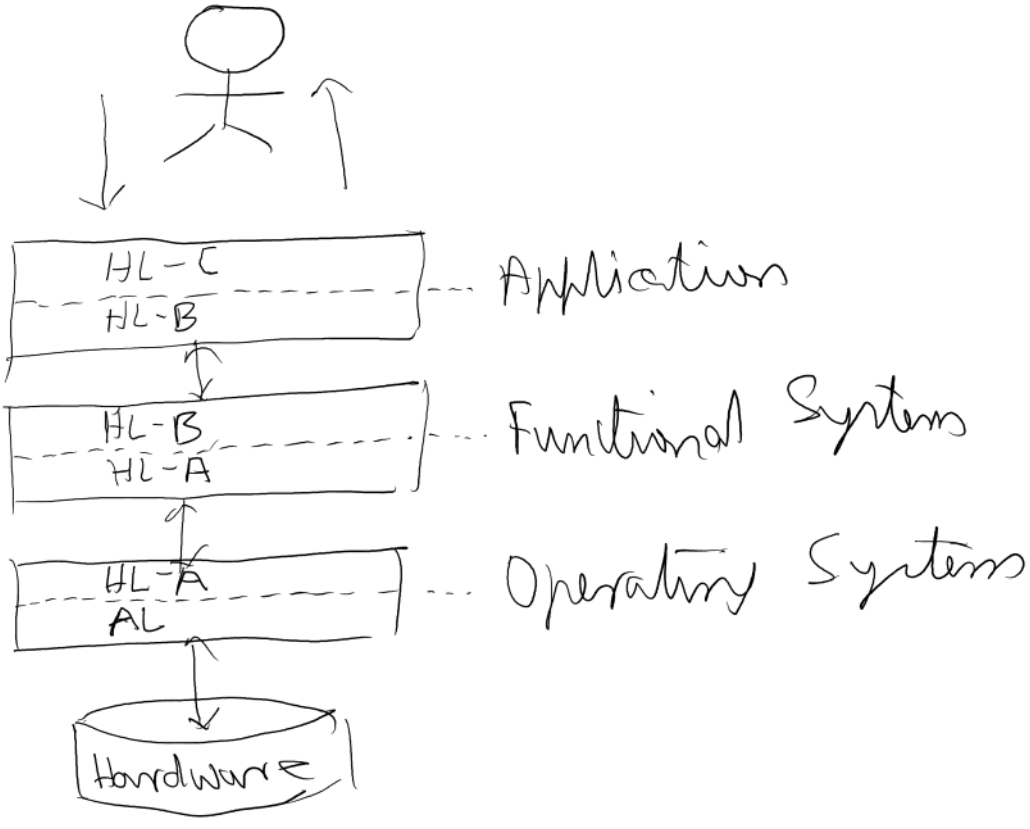


Programming Language: It is a *tool* that can be used for simplifying the process of writing the instructions of a program. It specifies a *syntax* consisting of a set of *symbols* and a set of *rules* for combining those symbols into a human readable form known as the *source code* of a program. Programming language is classified into two main types

1. **Assembly Language** - The symbols are machine specific *mnemonics* (friendly short names) for *opcodes* (binary instructions) which can be executed by that type of machine. Each file containing source code of a program is translated by the *assembler* into the corresponding *object code* file containing its binary equivalent and all such object code files required by the program are combined together by *linker* into the machine executable file of that program.
2. **High Level Language** - The symbols are machine independent *keywords* derived from a natural (spoken) language such as English. A high-level language is further classified into three categories:
   (A) The source code is translated by the *compiler* into the corresponding assembly language code which is then assembled and the resulting object code is linked with the runtime library of the language to produce the program executable file.
   Example: C and C++
   (B) The source code is translated by the *compiler* into machine

neutral (independent) binary instructions which are executed by the *runtime environment* of the language over any machine on which this environment is pre-installed.
Example: Java and C#
(C) The source code is executed one statement at a time by the *interpreter* on any machine on which the runtime support required by this language is pre-installed.
Example: Python and JavaScript

Feature	**Category A**	**Category B**	**Category C**
Checked typing	Yes	Yes	No
Dynamic typing	No	Yes	Yes (max)
Binary distribution	Yes	Yes	No
Portable distribution	No	Yes	Yes
Fast execution	Yes (max)	Yes	No
Safe execution	No	Yes	Yes



Variable: It enables a program to identify some data of known size and to access or update its current state. The value of each variable is stored in the *memory* at its own location and this location is identified by its *logical address* (page number and byte offset) which is determined using

1. **Direct Addressing** - the logical address of the value is linked to a *symbolic name* (label) assigned to it at the time of program implementation (coding). It provides a more efficient but less flexible mechanism for referring to data in the memory and is commonly used for supporting a *global variable* (which is visible to the entire program code).
2. **Indirection** - the logical address of the value is evaluated from an expression at the time of program execution (runtime). It provides a less efficient but more flexible mechanism for referring to data in the memory and is commonly used for supporting an *array* (which identifies address of first value in a packed sequence of multiple values of same size).

General Purpose Registers: These are fixed size data-pockets hard-wired within the machine and identified by names specific to the machine architecture such as

1. **x64** (Intel CORE, AMD Ryzen)
   RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15
2. **a64** (Qualcomm SnapDragon, Samsung Exynos, Apple M1)
   X0-X30, SP

Data Type: It indicates the size of the value identified by a variable and the restrictions on operations which can be performed on that variable. In a typed language a variable must be declared with a proper data-type before it can appear in an expression. In C language a variable is declared using following syntax

<qualifier> <data-type> <identifier> = <initializer>;

Data Type	Qualifier	Bit Size	Identified Value
char		8	single byte or a character (c)
int (default)		32	integer (d)
	short	16	integer (hd)

	long	native	integer (ld)
	long long	64	integer (lld)
	unsigned	above	non-negative integer (u)
float		32	single (7 digit) precision real number (f)
double		64	double (14 digit) precision real number (lf)

Control

11 March 2022 16:21

Control: It enable a program to make a decision by selecting an execution path other than the default sequential path. Machines provides *conditional jump* instructions to support

1. **Branching** - the execution control is transferred to an instruction other than the immediate next instruction if the program enters a particular state as indicated by the values of its variables.
2. **Looping** - the execution of a block of instructions known as an *iteration* is repeated while the program remains in a particular state as indicated by the values of its variables.

Control Structure: It is a block of statements containing a statement which indicates the condition for execution of that block. Control structures provide a more readable syntax for branching and looping in high-level programming languages.

Software Development Process

1. **Analysis** - The *functional analyst* identifies the set of requirements of a software and constructs algorithms for handling those requirements.
2. **Design**
   (a) The *solution architect* divides the above set of requirements into loosely coupled subsets so that change in one of them does not affect the others.
   (b) The *software engineer* expresses each of the above subset as a unit of data and code.
3. **Implementation**
   (a) The *senior programmer* writes code for each of the above unit according to its specific algorithm.
   (b) The *junior programmer* writes code to test whether the code for above unit works as per its requirements.

Procedures

15 March 2022 16:46

Procedure: It enables a program to *reuse* a block of code which handles a *routine* (commonly required) data-processing task from multiple points where such a processing has to be performed. The execution of the code of a procedure is called its *invocation* and it involves following two steps:

1. **Call** - the address of the next instruction and elements of data to be processed known as *arguments* are transferred along with the execution control to the first instruction of the procedure to begin its invocation.
2. **Return** - the result of data processing is transferred along with the execution control to the instruction whose address was passed at the beginning of the invocation of the procedure to end this invocation.

Function: It is a procedure whose definition includes identifiers of its arguments known as *parameters*. A high-level expression can include call to a function and when this expression is evaluated the included function is invoked and its call is replaced by the result of that invocation.

Pointer: It is an identifier of the logical address of an element in the memory. A C pointer has following characteristics:

1. A pointer of a specific data type (T*) is initialized by applying *address of* (unary &) operator to the identifier of compatible data type (T).
2. The data addressed by a pointer can be reference by applying *indirection* (unary *) operator to that pointer.

A safer form of pointer type (T*) which supports automatic addressing in initializer (without &) and automatic indirection in expressions (without *) is called *reference type* (T& in C++).

Exception: It is a fault raised by a function when it fails to complete its designated operation so that its caller can take appropriate action.

When a function *throws* an exception of particular type its invocation ends and the execution control is transferred into the *handler* of that exception type.

Modularization

23 March 2022 17:12

Code Modularization: It is a technique of dividing the implementation of a large program into separate sets of functions which can also be reused by other programs if required. A *module* is a linkable set of functions which can publish a subset of these functions and only these published function can be called from external code (not belonging to the module).
Modularization can allow multiple programmers to simultaneously implement, test and maintain different modules of a program if each module is designed according to following rules:

1. Functions with inter-dependent implementations (change in one requires change in other) should be defined in the same module.
2. The information shared by module about its published functions with other modules should be immutable (does not change with time).

Calling Conventions: It is a *machine-specific binary standard* which functions published by a module must follow so that they can be called from other modules. Calling conventions commonly supported by multiple programming language on UNIX systems provide following guide-lines for using registers in a procedure on

1. **x64 Architecture** - The procedure will accept first six integer type arguments in RDI, RSI, RDX, RCX, R8, R9 registers and will return integer type result in RAX register while preserving values in RBX, RBP, R12-R15 registers.
2. **a64 Architecture** - The procedure will accept first eight integer type arguments in X0-X7 registers and will return integer type result in X0 register while preserving values in X16-X30 registers.

Dynamic Linking: It is a linking mechanism in which the object code from a module is not copied into the executable file of its dependent program but is loaded from its own file into the memory when that program is executed. Object code modules can be linked into a *dynamic linkable* library (known as a shared-object on UNIX) so that its

dependent program can

1. Discover the path of the library and functions exported (published) by its modules at runtime.
2. Share the object code of modules of the library with other programs which depend on that library at runtime.

Aggregate Data Type: It is *a user-defined type* (UDT) implemented by the program for identifying some non-scalar data which contains multiple elements each with its own data-type. It allows a function to

1. Accept a complex form of data with multiple items through a single argument.
2. Adapt to any change in the internal structure of data it accepts through an argument without requiring a corresponding change in its list of parameter types.

Abstraction

27 March 2022 10:49

Data Abstraction: It is a technique of dividing the data processed by a program into smaller structures known as objects each controlling its own *state* (content) and exhibiting its own *behavior* (actions). A *class* defines a set of *members* containing variables known as *fields* whose values indicate the state of a particular type of object and functions known as *methods* whose implementations describe the behavior of that object. A class provides support for following mechanisms

1. **Activation** - An object refers to a memory-block known as an *instance* of its class which contains its own values of the fields defined by that class initialized by calling a special method known as the *constructor* of that class.
2. **Binding** - Every object has its own unique *identity* (address) and when a method defined by the class is called on its object the identity of *this* object is passed to the implementation of that method.

SOLID Principles

1. **Single Responsibility Principle:** There should exactly one reason for changing the functionality supported by an object.
   Rule: define only those members in a class which are commonly required by all of its objects in an identical manner.

2. **Open Closed Principle:** The functionality exposed by an object should be open for extension but closed for modification.
   Rule: include only methods in the set of members published by the class.

Polymorphism

30 March 2022 16:06

Subtype Polymorphism: It is a technique for defining a function that can operate upon multiple (poly) types (morphs) of objects which share some common behavior. A *subclass* is *derived* from a predefined class to *override* (provide new implementations for) its existing methods. A subclass provides support for following mechanisms:

1. **Inheritance** - An object of a subclass automatically supports all the members of the class from which it is derived and when a method defined by a class is called on an object of its subclass which has overridden that method the invocation is dispatched to the implementation provided by the subclass.
2. **Interface** - An *abstract type* which does not support activation can define *pure* (unimplemented) methods which must be implemented by its non-abstract subclasses to support creation of instances.

SOLID Principles

3. **Liskov's Substitution Principle:** A consumer of an object of a particular type should also be able to consume an object of its subtype.
   Rule - Call methods of subclass indirectly through the class from which it is derived.

4. **Interface Segregation Principle:** A consumer of an object should not be forced to depend on an interface of that object which it does not require.
   Rule - Derive different classes which support a common method from an abstract type in which that method is pure.

5. **Dependency Inversion Principle:** An object should be consumed through its interface and not through its implementation.
   Rule - Use a non-abstract subclass of an abstract type only to indirectly create an instance of that abstract type.

Object Oriented Programming: It is a programming methodology (systematic style) based on SOLID principles.

Parametric Polymorphism: It is a technique for implementing a *reusable code pattern* which can be applied to different data-types while supporting *strict type-checking*. It is used for defining a

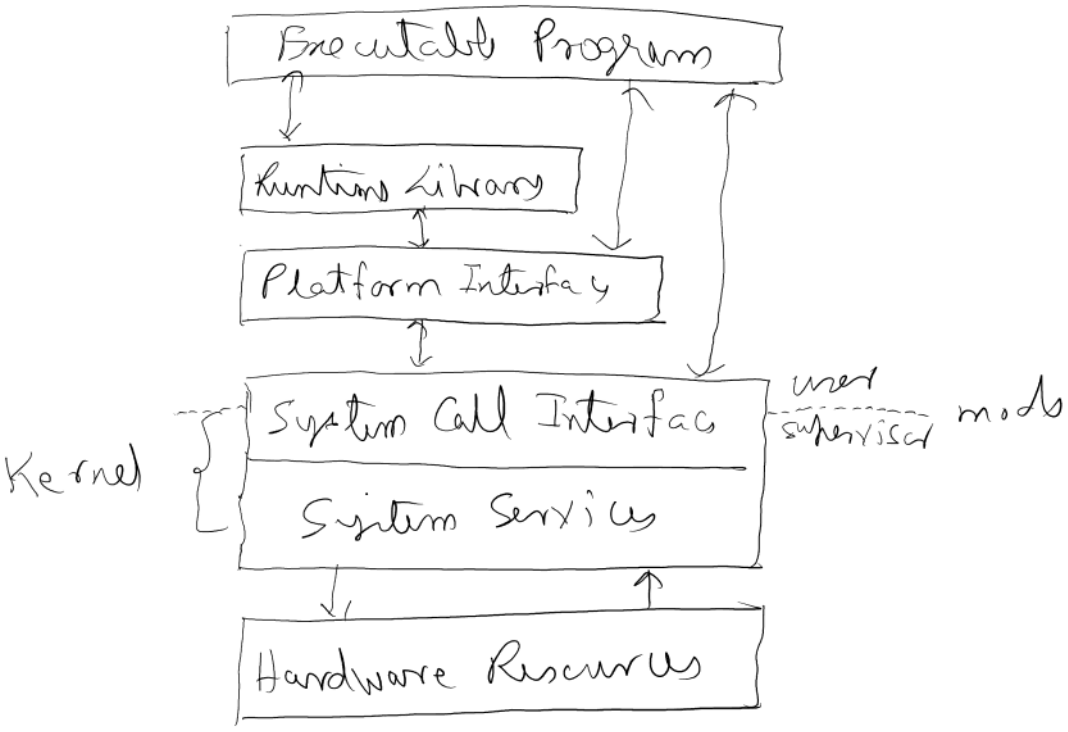
1. **Generic Function** which is parameterized over the data-types associated with its arguments and return value so that its implementation can handle data of any type specified in its usage in a type-safe manner.
2. **Generic Class** which is parameterized over the data-types associated with its members so that its object can handle data of any type specified in its usage in a type-safe manner.

Platform

04 April 2022 15:31

Operating System: It is a software which is automatically loaded and executed when a machine is powered up to provide support for executing any other installed software on demand. It has two main parts

1. **Kernel:** It manages hardware resources available on a machine and allows a process (started by running a program) to share these resources with other simultaneously executing processes while isolating them from each other. It implements *system services* for handling
   (a) **Process Scheduling** enables each process to run concurrently with other processes using its own slice of execution time available on the *hardware processor*.
   (b) **Memory Management** enables each process to access its own portion of *virtual memory* (which spans across physical memory and secondary storage) using *flat logical address space*.
   (c) **IO Control** enables each process to read/write data from/to an input/output device using a *device independent file interface*.
   (d) **IPC Control** enables each process to synchronize execution and exchange data with other concurrently executing processes.
2. **System Call Interface:** It is a machine architecture specific mechanism for allowing a process running in *user-mode* (with limited access to instruction set of the processor) to call upon the operations of system services implemented by the kernel process which executes in *supervisor-mode* (with full access to instruction set of the processor). It also serves as a basis for
   (a) **Platform Interface** - It is a C language based API built on top of system-call interface for consuming system services offered by the kernel in a machine independent manner.
   (b) **Runtime Library** - It is a programming language specific API built on top of platform interface for consuming system services offered by the kernel in a platform independent manner.


Executable Program
Runtime Library
Platform Interface
System Call Interface
System Services
Hardware Resources
Kernel
user
supervisor
mode