Contents

[Contents 1](#_Toc171198615)

[**A.** **Programming skill** 5](#_Toc171198616)

[**1.** **Data alignment** 5](#_Toc171198617)

[**2.** **Memory Layout** (memory of a process) 5](#_Toc171198618)

[**3.** **Synchronization in process, thread in Linux System.** 5](#_Toc171198619)

[**3.1** **Multithread:** 5](#_Toc171198620)

[**3.2** **Mutex:** 6](#_Toc171198621)

[**3.3** **Condition variable:** 7](#_Toc171198622)

[**3.4** **Semaphore:** 9](#_Toc171198623)

[**3.5** **spin lock:** 10](#_Toc171198624)

[**3.6** **atomic:** 11](#_Toc171198625)

[**4.** **Concurrency Programming:** 11](#_Toc171198626)

[**4.1** **Concurrency design target:** 11](#_Toc171198627)

[**4.2** **Data structure Protection:** 11](#_Toc171198628)

[**4.2.1** **Lock-based concurrent data structure** (*blocking* data) 12](#_Toc171198629)

[**4.2.1.1** Thread\_safe using lock 12](#_Toc171198630)

[**4.2.1.2** Thread\_safe using lock and condition variable 12](#_Toc171198631)

[**4.2.1.3** Thread\_safe lookup table using lock 12](#_Toc171198632)

[**4.2.1.4** Thread\_safe list using lock 12](#_Toc171198633)

[**4.2.2** **Lock-free concurrent data structure** (*nonblocking +* ) 12](#_Toc171198634)

[**5.** **Dynamic Programming** 12](#_Toc171198635)

[**6.** **Ranking problem** 13](#_Toc171198636)

[**7.** **Recursion** 13](#_Toc171198637)

[**8.** **Linkage** 13](#_Toc171198638)

[**B.** **Embedded Knowledge** 14](#_Toc171198639)

[**1.** **LSI** 14](#_Toc171198640)

[**2.** **Embedded system architecture** 14](#_Toc171198641)

[**3.** **Hardware component** 14](#_Toc171198642)

[**4.** **64 bit** 14](#_Toc171198643)

[**4. Cache memory:** 15](#_Toc171198644)

[**5.** **Register:** 15](#_Toc171198645)

[**6.** **Virtual memory** 16](#_Toc171198646)

[**7.** **Unix OS architecture:** 16](#_Toc171198647)

[**8.** **QNX** 17](#_Toc171198648)

[**9.** **GNU** 17](#_Toc171198649)

[**10.** **Linux kernel:** 18](#_Toc171198650)

[**11.** **Device Tree** 19](#_Toc171198651)

[**12.** **Bootloader:** 19](#_Toc171198652)

[**13.** **Rootfs:** 20](#_Toc171198653)

[**1.** **Inter-Process communicate (IPC)** 21](#_Toc171198654)

[**2.** **Yocto project** 23](#_Toc171198655)

[**3.** **Buildroot** 23](#_Toc171198656)

[**4.** **Middleware** 24](#_Toc171198657)

[**5.** **Driver** 24](#_Toc171198658)

[**6.** **Makefile and make** 24](#_Toc171198659)

[**7.** **MCU/MPU peripherals:** 25](#_Toc171198660)

[**8.** **Interrupt** 28](#_Toc171198661)

[**9.** **Multitask concept:** 28](#_Toc171198662)

[**10.** **Firmware** 30](#_Toc171198663)

[**11.** **SoC vs MCU** 30](#_Toc171198664)

[**12.** **Virtualization** 30](#_Toc171198665)

[**13.** **Platform:** 30](#_Toc171198666)

[**14.** **JTAG** 31](#_Toc171198667)

[- Refer: http://nguyenquanicd.blogspot.com/2019/11/jtag-bai-1-tong-quan-ve-giao-thuc-jtag.html 31](#_Toc171198668)

[**15.** **XIP** 32](#_Toc171198669)

[**C.** **Programming Knowledge** **C/C++** 33](#_Toc171198670)

[**1.** **Object Oriented Programming (OOP trait)** 33](#_Toc171198671)

[**1.** **Inheritance:** 33](#_Toc171198672)

[**2.** **Polymorphism:** 33](#_Toc171198673)

[**3.** **Atraction:** 33](#_Toc171198674)

[**4.** **Encapsulation:** 34](#_Toc171198675)

[**5.** **Static member** 36](#_Toc171198676)

[**6.** **Allocate and de-allocate** 36](#_Toc171198677)

[**7.** **Con trỏ this:** 37](#_Toc171198678)

[**8.** **Hàm Friend:** 37](#_Toc171198679)

[**9.** **Class friend:** 37](#_Toc171198680)

[**10.** **Const Reference** 37](#_Toc171198681)

[**II.** **Specifier keywords:** 38](#_Toc171198682)

[**1.** **Static:** 38](#_Toc171198683)

[**2.** **Inline** 38](#_Toc171198684)

[**3.** **Volitate** 38](#_Toc171198685)

[**4.** **Const** 39](#_Toc171198686)

[**5.** **attribute** 39](#_Toc171198687)

[**6.** **Extern** 39](#_Toc171198688)

[**III.** **Data structure** 40](#_Toc171198689)

[**1.** **String** 40](#_Toc171198690)

[**2.** **Array** 40](#_Toc171198691)

[**3.** **Struct** 41](#_Toc171198692)

[**4.** **Union** 42](#_Toc171198693)

[**5.** **Enum** 42](#_Toc171198694)

[**6.** **Pointer (\*) vs Reference (&):** 43](#_Toc171198695)

[**7.** **Smart Pointer** 44](#_Toc171198696)

[**IV.** **Some features** 45](#_Toc171198697)

[**1.** **Function macros:** 45](#_Toc171198698)

[**2.** **Function pointer:** 45](#_Toc171198699)

[**3.** **Variadic Function:** 46](#_Toc171198700)

[**4.** **Scope rules:** 46](#_Toc171198701)

[**5.** **Tham Biến vs Tham Trị:** 46](#_Toc171198702)

[**6.** **Exception** 48](#_Toc171198703)

[**7.** **Standard Template Library (STL)** 49](#_Toc171198704)

[**8.** **Namespace** 51](#_Toc171198705)

[**9.** **Template** 51](#_Toc171198706)

[**V.** **Advance C++** 52](#_Toc171198707)

[**1.** **Dispatching** 52](#_Toc171198708)

[**2.** **Vtable** 52](#_Toc171198709)

[**3.** **Design Patern** 52](#_Toc171198710)

[**VI.** **Some question:** 53](#_Toc171198711)

Knowledge Systematic

# **Network knowledge**

## **OSI Model**

## **Application Protocol**

### **HTTP/HTTPs and REST**

Refer: https://stackoverflow.com/a/44731594

* **REST** is the way **HTTP** should be *used*.
* **REST** is a protocol to exchange any(XML, JSON etc ) messages that can use HTTP to transport those messages.
* **REST** is an architectural style for an API that uses HTTP requests like GET, PUT, POST, and DELETE to manage CRUD (Create, Read, Update, and Delete) operations through which applications and servers communicate.
* **REST** enforces the use of the available HTTP commands as they were meant to be used.

For example, I could do:

|  |
| --- |
| GET  http://example.com?method=delete&item=xxx |

But with REST I would use the "DELETE" request method, removing the need for the "method" query param

|  |
| --- |
| DELETE  http://example.com?item=xxx |

Additional info:

* RESTful API is a standardized implementation of the REST architecture.
* REST API and RESTful API are based on the REST architecture.

### **SOME/IP**

### **gRPC**

### **Cap'n proto**

### **MQTT (paho, QoS, TLS, ...)**

### **Kuksa databroker**

## **TCP/UDP**

## **DNS & DHCP**

## **Cellular**

## **WIFI/Bluetooth**

# **Programming Techniques**

## **Data alignment**

* Refer: https://cppdeveloper.com/c-nang-cao/data-alignment-trong-c-c/
* Căn chỉnh các phần tử của dữ liệu cho phép CPU lấy dữ liệu từ bộ nhớ một cách hiệu quả và do đó cải thiện hiệu năng (performance) của chương trình.
* Trình biên dịch sẽ cố gắng duy trì việc căn chỉnh của các phần tử dữ liệu bằng cách chèn các ô nhớ không sử dụng giữa các phần tử. Kỹ thuật này được gọi là “Padding”.

## **Memory Layout** (memory of a process)

Refer: https://www.geeksforgeeks.org/memory-layout-of-c-program/

* Code segment: Store only code in ROM
* Heap segment: Store dynamic allocation: malloc, calloc, realloce, free, new and delete.
* Data segment: Global & static variables were initialised.
* BSS (uinitialised segment): Global & static variables are not initialised or initialized to 0.
* Stack segment: Local, tempory return address.

\*Note: check an execute file by “size”

## **Synchronization in process, thread in Linux System.**

## **Multithread:**

* A process is an instance of a computer program containing binary code along with the resources above.
* A thread is a component of a process. It is an execution unit and it contains program counter, stack and set of registers.

|  |
| --- |
| *pthread\_t tid;*  *…*  *void\* func\_thread(){...}*  *…*  *main(){*  *pthread\_create(&tid, NULL, func\_thread, NULL);*  *...*  *pthread\_join(tid,NULL);*  *}* |

- **pthread\_create()**: thread được tạo ra sẽ bắt đầu được thực thi ngay lập tức, nhưng nếu ko đc canh bởi pthread\_join() thì sẽ xãy ra trường hợp thread child chưa chạy xong mà main thread đã end → end luôn thread child.

- **pthread\_join():** Join để block main thread và chờ đến khi thread child trả về, rồi mới tiếp tục chạy. (Lưu ý: nó chỉ block main thread mà ko block thread child khác)

## **Mutex:**

* Mutex lock là một cấu trúc dữ liệu, xây dựng theo nguyên tắc mutual exclusion (một thời điểm bất kì, chỉ có tối đa một thread truy cập vào critical resource.)
* 3 thành phần chính trong struct: biến count, biến owner và hàng đợi wait\_list.(count = 1: UNLOCK , < 1: mutex đang ở trạng thái LOCKED [ =0: ko có thread khác ở wai\_list, <0: đang có thread khác đợi ở wait\_list])
* Nếu 1 thread gọi mutex\_unlock() khi CR đang LOCKED thì : CPU tạm ngừng thực thi thread này rồi chuyển sang thực thi thread khác (nói theo ngôn ngữ của CPU). Hay nói theo ngôn ngữ của Linux kernel, thread được thêm vào hàng đợi **wait\_list** và sẽ đi ngủ, sau đó Linux kernel sẽ lập lịch cho thread khác. Do đó, ta nói rằng, mutex lock áp dụng cơ chế **sleep-waiting,** tức là mutex lock thuộc loại **sleep lock**, trái với spinlock thuộc loại **busy lock**
* Trong TH 2 thread đồng thời truy cập CR thì giống với spin\_lock, hàm mutex\_lock dùng thao tác *atomic* (xem hàm *mutex\_lock* → tương tác biến count giống cơ chế atomic??) để thay đổi biến *count*, nên chỉ có một trong hai thread chiếm được
* **POSIX mutex** (POSIX thread library):

|  |
| --- |
| *pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;*  *void\* func\_thread(){*  *pthread\_mutex\_lock(&mutex);*  */ /Do something*  *pthread\_mutex\_unlock(&mutex);*  *}* |

* **Linux mutex** (Solaris *thread* library):

|  |
| --- |
| *DEFINE\_MUTEX(my\_mutexlock); //khơi tạo UNLOCKED (Compile time)*  *or*  *struct mutex my\_mutexlock; // cap phat (Runtime)*  *mutex\_init(&my\_mutexlock);;*  *void\* func\_thread(){*  *mutex\_lock(&my\_mutexlock);*  */ /Do something*  *mutex\_unlock(&my\_mutexlock);*  *}* |

## **Condition variable:**

- Mutex is for exclusive access of shared resources, while conditional variable is about waiting for a condition to be true**.**

|  |
| --- |
| *pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;*  *pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;*  *void\* func\_thread(){*  *pthread\_mutex\_lock(&mutex);*  */ /Do something*  *if(condition comtrue){*  *pthread\_cond\_signal( &cond);}*  *pthread\_mutex\_unlock(&mutex);*  *}*  *main(){*  *…*  *pthread\_cond\_wait(&cond, &mutex); // de cho khac cung dc*  *….*  *}* |

- pthread\_cond\_wait(): **release a lock** specified by mutex and **wait on condition** cond variable.

- pthread\_cond\_signal(): wake up threads waiting for the condition variable by **send a signal.**

**Same mechanism in C++:**

* **wait()**
  + is used by a thread to block its execution until it is notified by another thread.
  + require ***std:unique\_lock*** to **ensures proper unlocking of the mutex when the thread is waiting.**
  + ***Explain:*** 
    - When the thread calls wait(), it atomically releases the associated lock and enters a blocked state waiting for notification.
    - Internally, wait() unlocks the associated mutex and puts the thread to sleep until it is awakened by a call to notify\_one() or notify\_all() from another thread.
* **notify\_one():**
  + wakes up exactly one of the threads that are currently blocked on the condition variable.
  + If multiple threads are waiting, which thread to wake up is non-deterministic and depends on the implementation and the scheduler.
  + The awakened thread will then compete with other threads (if any) for the associated lock. Once the awakened thread acquires the lock, it resumes execution.
* **notify\_all():**
  + wakes up all threads that are currently blocked on the condition variable.
  + All waiting threads are awakened simultaneously and start competing for the lock associated with the condition variable. Once a thread acquires the lock, it resumes execution, while the other threads will go back to waiting for the condition.
  + Note that calling notify\_all() when only one thread is necessary can potentially lead to more contention and lower performance, as all waiting threads will wake up and compete for the lock, even though only one thread can proceed.

## **Semaphore:**

**- Refer:** <https://www.guru99.com/semaphore-in-operating-system.html>,https://linux.die.net/man/7/sem\_overview

- Semaphore là một cấu trúc dữ liệu, semaphore tương tự như một bộ các chìa khóa dự phòng (tại một thời điểm có thể có nhiều thread/process truy cập đồng thời CR (TH dùng Counting semaphore)).

- 2 thành phần chính của struct: biến count và hàng đợi wait\_list. (= 0: semaphore UNAVAILABLE, > 0: semaphore đang ở trạng thái AVAILABLE)

- Căn cứ vào giá trị của biến count, semaphore được chia làm 2 loại:

* **Counting semaphore:** Max của biến count lớn hơn 1. Giá trị cực đại của biến count thể hiện số lượng thread tối đa được phép sử dụng critical resource tại cùng một thời điểm.
* **Binary semaphore:** Nếu biến count chỉ có hai giá trị 0 và 1. Binary semaphore có một số nét tương đồng với mutex lock.

- Khi count đang bằng 0, tức là semaphore đang ở trạng thái UNAVAILABLE, nếu một thread gọi hàm **down**,

thì CPU tạm dừng thực thi thread này rồi chuyển sang thực thi thread khác (nói theo ngôn ngữ của CPU). Hay

nói theo ngôn ngữ của Linux kernel, thread đó được thêm vào hàng đợi **wait\_list** và đi ngủ, sau đó Linux

kernel sẽ lập lịch cho thread khác. Do đó, ta nói rằng, semaphore áp dụng cơ chế **sleep-waiting.**

- Trường hợp 2: đồng thời muốn truy cập CR. Khi đó, cả 2 thread đồng thời thực thi hàm **down**. Tuy nhiên, do biến **count** của *struct semaphore* được bảo vệ bằng một **spinlock** *(ko phải atomic như mutex)*, nên chỉ có một trong hai thread chiếm được CR.

**- Linux Semaphore** (Solaris thread library):

|  |
| --- |
| *DEFINE\_SEMAPHORE(my\_semaphore); //khơi tạo UNLOCKED (Compile time)*  *or*  *struct semaphore my\_semaphore; // cap phat (Runtime)*  *sema\_init(&my\_semaphore, <count>);*  *void\* func\_thread(){*  *up(&my\_mutexlock);*  */ /Do something*  *down(&my\_mutexlock);*  *}* |

**- POSIX semaphore:** <https://www.geeksforgeeks.org/use-posix-semaphores-c/>

|  |
| --- |
| *sem\_t mutex;*  *sem\_init(&mutex, <pshared>, <value>);*  *void\* func\_thread(){*  *sem\_wait(&mutex);*  */ /Do something*  *sem\_post(&mutex);*  *}*  *sem\_destroy(&mutex);*  *example:*  *sem\_init(&mutex, <pshared>, N); → Counting semaphore ( N thread đc access CR đồng thời).*  *sem\_init(&mutex, <pshared>, 1); → Binary semaphore ( chỉ 1 thread đc access CR tại 1 thời điểm)* |

- 2 types by name:

+ Named semapore: A named semaphore is identified by a name of the form */somename. P*rocesses/threads operate on the same named semaphore by passing the same name to ***[sem\_open](https://linux.die.net/man/3/sem_open)****()*,...

+ Unnamed semapore (memory-base semaphore): Do not have a name. Before being used, an unnamed semaphore must be initialized using ***[sem\_init(),…](https://linux.die.net/man/3/sem_init)***

Note: Defferent MUTEX vs SEMAPHORE (Binary) ???https://www.geeksforgeeks.org/difference-between-binary-semaphore-and-mutex/?ref=rp)

## **spin lock:**

- Linux sử dụng *cấu trúc spinlock\_t* để mô tả một spinlock. Bản chất của cấu trúc này phụ thuộc vào kiến trúc bộ xử lý. Thông thường, đây là một biến atomic.

- Busy lock: If a resource is locked, a thread that wants to access that resource may repetitively check whether the resource is available.thì CPU liên tục thực thi hàm acquire\_spinlock cho tới khi spinlock chuyển sang trạng thái UNLOCKED. Tình trạng này được gọi là giao tranh (contended), còn cơ chế chờ đợi này được gọi là chờ bận (busy-waiting).

|  |
| --- |
| *DEFINE\_SPINLOCK(my\_spinlock); //khơi tạo UNLOCKED (Compile time)*  *or*  *spinlock\_t my\_spinlock; // cap phat (Runtime)*  *spin\_lock\_init(&my\_spinlock);*  *void\* func\_thread(){*  *spin\_lock(&my\_spinlock);*  */ /Do something*  *spin\_unlock(&my\_spinlock);;*  *}* |

- hàm *spin\_lock và spin\_unlock* ko cản được thread của Interrupt → để vô hiệu hóa interrupt thì dùng hàm *spin\_lock\_irq và spin\_unlock\_irq* hoặc *spin\_lock\_irqsave/spin\_unlock\_irqrestore.*

- Do bottom-half có độ ưu tiên lập lịch cao hơn các kernel thread thông thường, nên để tránh race condition, ta cần thông báo với kernel rằng: tạm dừng lập lịch cho bottom-half trước khi CPU thực thi critical section của kernel thread thông thường → Dùng hàm *spin\_lock\_bh và spin\_unlock\_bh*

## **atomic:**

- Critical resource chỉ là một biến hay một bit, và thao tác truy cập cũng đơn giản (tăng, giảm,...).

- 2 nhóm thao tác atomic:

+ Một nhóm dùng để truy cập critical resource là số nguyên (atomic integer operation): type atomic\_t, function: atomic\_read(), atomic\_set(),..

+ Một nhóm dùng để truy cập critical resource là bit (atomic bitwise operation): set\_bit(), clear\_bit(),...

## **Concurrency Programming:**

## **Concurrency design target:**

* Enable true concurrent access. Some data structures have more scope for true concurrency than others:
  + Smaller the protected region.
  + Fewer operations are serialized.
  + Greater the potential for concurrency.

## **Data structure Protection:**

* Designing a data structure for concurrency means that multiple threads can access the data structure concurrently, either performing the same or distinct operations, and each thread will see a self-consistent view of the data structure. No data will be lost or corrupted, all invariants will be upheld, and there’ll be no problematic race conditions.
  + - * + **Such a data structure is said to be *thread-safe.***
* In general, a data structure will be safe only for particular types of concurrent access*. It may be possible*

*to have multiple threads performing one type of operation on the data structure concurrently,* ***whereas*** *another operation requires exclusive access by a single thread.*

## **Lock-based concurrent data structure** (*blocking* data)

* Thread will suspend the execution until another thread performs an action. because the thread can’t progress past this point until it’s *unblocked* by the appropriate action of another thread, whether that’s unlocking a mutex, notifying a condition variable, or making a future *ready*.

## Thread\_safe using lock

## Thread\_safe using lock and condition variable

## Thread\_safe lookup table using lock

## Thread\_safe list using lock

## **Lock-free concurrent data structure** (*nonblocking +* )

* Using atomic operations and the associated memory-ordering guarantees in order to ensure that data becomes visible to other threads in the correct order.

**Conclusion:**

* ***lock-based*** containers, there’s always the potential for one thread to have to block and wait for another to complete its operation before the first thread can proceed; preventing concurrency through mutual exclusion is the entire purpose of a mutex lock.
* ***lock-free*** data structure, *some* thread makes progress with every step.
* ***wait-free*** data structure, every thread can make forward progress, regardless of what the other threads are doing; there’s no need for waiting.

## **Dynamic Programming**

* Dynamic Programming is mainly an optimization over plain [recursion](https://www.geeksforgeeks.org/recursion/).
* Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
* **Tabulation (Bottom Up):**The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table.
* **Memoization (Top Down):**The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL.

The following computer problems can be solved using dynamic programming approach −

* Fibonacci number series
* Knapsack problem
* Tower of Hanoi
* All pair shortest path by Floyd-Warshall
* Shortest path by Dijkstra
* Project scheduling
* DP vs Recursion: Dynamic programming is basically, recursion plus using common sense. <https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/>

## **Ranking problem**

## **Recursion**

* Concept: The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.
* Type:

+ A direct recursive function: it calls the ITSELF inside.

+ A indirect recursive function: it calls another function say fun\_new and fun\_new calls IT directly or indirectly.

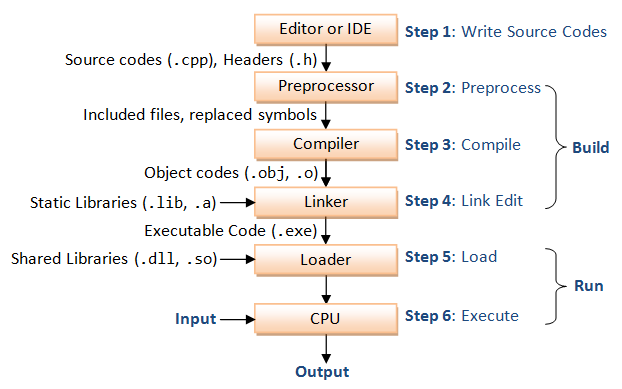
* Usage: provides a clean and simple way to write code.
* Disavantage:

+ recursive and iterative programs have the same problem-solving powers.

+ Recursive program has greater space, time requirements than iterative.

## **Linkage**

* Refer: <https://www.tenouk.com/ModuleW.html>



# **Embedded Knowledge**

**What is Embedded system?**

Refer: https://www.omnisci.com/technical-glossary/embedded-systems

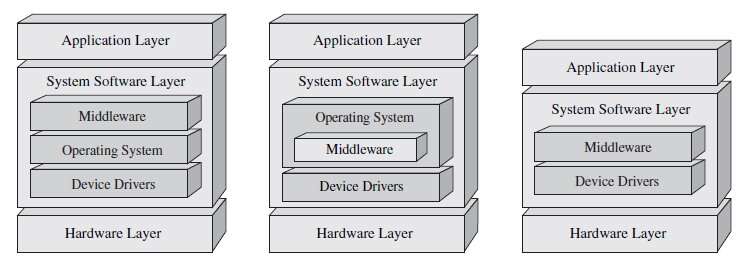
 An embedded system is a combination of computer hardware and software designed for a specific function or functions ( can be within a larger mechanical or electrical system.)[https://vi.wikipedia.org/wiki/H%E1%BB%87\_th%E1%BB%91ng\_nh%C3%BAng](https://vi.wikipedia.org/wiki/Hệ_thống_nhúng)

## **LSI**

- Large-scale integration (LSI) is the process of integrating or embedding thousands of transistors on a single silicon semiconductor microchip. LSI technology was conceived in the mid-1970s when computer processor microchips were under development.

- Today we also have:  very large-scale integration (VLSI) and ultra large-scale integration (ULSI) technologies.

## **Embedded system architecture**

* Application layer.
* Software system layer:

+ OS(+middleware)

+ driver

* Hardware layer.

Figure 1: Embedded system architecture

## **Hardware component**

* CPU: (Central processing unit) process data and command to control system.
* MEMORY: Store data and command.
* I/O modules: connect device, send data signal to CPU and recive control signal.
* Bus system: Connect RAM, CPU, I/O modules together.

+ Bus data: express data among RAM-CPU-I/O and command RAM->CPU-I/O.

+ Bus address:

+ Bus control: express control signal from CPU-> RAM, I/O; notification signal RAM, I/O -> CPU.

## **64 bit**

* Refer: <https://www.geeksforgeeks.org/difference-32-bit-64-bit-operating-systems/> and Wiki
* 64-bit [computers](https://en.wikipedia.org/wiki/Microcomputer) are computers in which 64-bit [processors](https://en.wikipedia.org/wiki/Microprocessor" \t "Microprocessor) are the norm
* From the software perspective, 64-bit computing means the use of [code](https://en.wikipedia.org/wiki/Machine_code" \t "Machine code) with 64-bit [virtual memory](https://en.wikipedia.org/wiki/Virtual_memory" \t "Virtual memory) addresses.
* In computing, there exist two type processor i.e., 32-bit and 64-bit. This type of processor tells us how much memory a processor can have access from a CPU register.
* A computer with a 64-bit processor can have a 64-bit or 32-bit version of an operating system installed. However, with a 32-bit operating system, the 64-bit processor would not run at its full capability.
* Example:
  + The **ARM Cortex-A7 MPCore** is a **32-bi**t microprocessor core licensed by [ARM Holdings](https://en.wikipedia.org/wiki/ARM_Holdings" \t "ARM Holdings) implementing the [ARMv7-A architecture](https://en.wikipedia.org/wiki/ARM_architecture" \t "ARM architecture).
  + The **ARM Cortex-A53** is one of the first two [microarchitectures](https://en.wikipedia.org/wiki/Microarchitecture" \t "Microarchitecture) implementing the [ARMv8-A](https://en.wikipedia.org/wiki/ARMv8-A" \t "ARMv8-A) **64-bit** [instruction set](https://en.wikipedia.org/wiki/Instruction_set) designed by [ARM Holdings](https://en.wikipedia.org/wiki/ARM_Holdings" \t "ARM Holdings)

## **4. Cache memory:**

- Refer:<https://www.geeksforgeeks.org/cache-memory-in-computer-organization/>

**- Cache Memory** is a special very high-speed memory.

- acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

- Cache Performance:

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

+ If the processor finds that the memory location is in the cache, a cache hit has occurred and data is read from cache

+ If the processor does not find the memory location in the cache, a cache miss has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

## **Register:**

* Thanh ghi là đang nói đến cấu trúc phần cứng.
* Trên con vi xử lý, vi điều khiển…có một vùng nhớ đặc biệt. Trong vùng nhớ này mỗi bit, byte có giá trị ảnh hưởng trực tiếp đến hoạt động của chip hoặc chip sẽ dùng các byte đó trong tính toán. Tập hợp của một hay nhiều byte có cùng chức năng gọi là một thanh ghi.
* Nhà sản xuất sẽ phân biệt các thanh ghi với nhau bằng cách đặt tên cho nó. Do nó là cấu tạo của phần cứng nên số lượng là cố định và sẽ không có khái niệm hệ điều hành cấp thanh ghi cho ứng dụng.

## **Virtual memory**

* Purpose of VM:
  + Avoid crash.
  + Manage effectively physical memory: Mapping flexible.
  + Protected stored data, avoid overiding (RAM is random access memory).
* Virtual address: <https://whatis.techtarget.com/definition/virtual-address>
  + A virtual address is a **[binary](https://whatis.techtarget.com/definition/binary) number** in **[virtual memory](https://searchstorage.techtarget.com/definition/virtual-memory)** that enables a process:
    - Use a location in [primary storage](https://searchstorage.techtarget.com/definition/primary-storage) (RAM memory) independently of other processes.
    - Use more space by temporarily relegating some contents to **a**[**hard disk**](https://searchstorage.techtarget.com/definition/hard-disk) or **internal [flash drive](https://searchstorage.techtarget.com/definition/flash-based-solid-state-drive-SSD)**.
  + It is **address type** that is **only use in Software.**

## D:\E\01_Programming\Linux\Anatory of the Linux Kernel.jpg**Unix OS architecture:**

* Refer: https://www.guru99.com/operating-system-tutorial.html
* Usage: Manage HW resource and provide HW usage services.
* Component:

+ Kernel

+ System call ( Các API được cung cấp bởi Kernel)

+ Libraries

+ Application

Figure 2: Architecture of the UNIX operating system

- **System call** is a request in a [Unix-like](http://www.linfo.org/unix-like.html) operating system by an *active process* (i.e., a process currently progressing in the CPU) for a service performed by the kernel, such as *input/output* (I/O) or *process creation* (i.e., creation of a new process). I/O can be defined as any movement of information to or from the combination of the CPU and main memory (i.e. RAM), that is, communication between this combination and the computer's users (e.g., via the keyboard or mouse), its [storage](http://www.linfo.org/storage.html) devices (e.g., disk or tape drives), or other computers.

## **QNX**

**-** is a commercial [Unix-like](https://en.wikipedia.org/wiki/Unix-like)[real-time operating system](https://en.wikipedia.org/wiki/Real-time_operating_system), aimed primarily at the [embedded systems](https://en.wikipedia.org/wiki/Embedded_system)market.

- As a [microkernel](https://en.wikipedia.org/wiki/Microkernel)-based OS, QNX is based on the idea of running most of the [operating system](https://en.wikipedia.org/wiki/Operating_system) [kernel](https://en.wikipedia.org/wiki/Kernel_(operating_system)) in the form of a number of small tasks, named Resource Managers.

- This differs from the more traditional [monolithic kernel](https://en.wikipedia.org/wiki/Monolithic_kernel), in which the operating system kernel is one very large program composed of a huge number of parts, with special abilities. **In the case of QNX,** the use of a microkernel allows users (developers) to turn off any functions they do not need without having to change the OS. Instead, such services will simply not run.

## **GNU**

## **Linux kernel:**

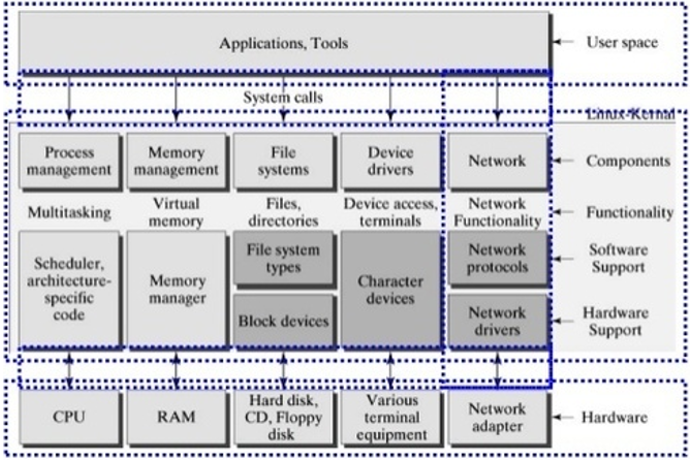
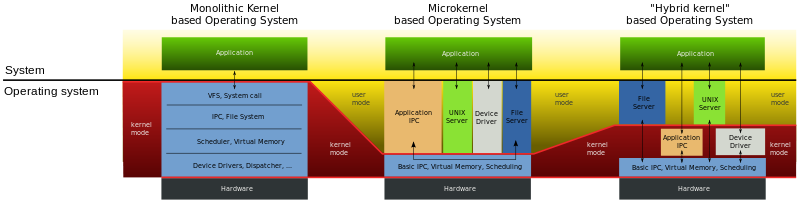
* Refer: <https://vimentor.com/vi/lesson/gioi-thieu-ve-linux-kernel-1>
* Main part of OS.
* Configured to run on a particular board (may be part of a BSP supplied by a hardware vendor).

Figure 3: Linux kernel Structure

* ***3 Types of Kernel****: Dựa vào mức độ can thiệp của Kernel (% CPU in Kernel mode)*
  + [**Hệ điều hành Linux**](http://hoangit.org/he-dieu-hanh/#linux) sử dụng Kernel Monolithic
  + [**Hệ điều hành MacOS**](http://hoangit.org/he-dieu-hanh/#macos) (XNU) và **[Hệ điều hành Windows](http://hoangit.org/he-dieu-hanh/" \l "windown)** 7 sử dụng Kernel Hybrid.

## **Device Tree**

* Refer:
  + - <https://kipalog.com/posts/Device-Tree-trong-Linux>
    - <https://www.raspberrypi.org/documentation/configuration/device-tree.md>
* Declare hardware info of devices in system → manage address (bus address that cpu use to access to device), resources (reg, interrupt, clock, dma,..).
* 2 types:
  + - Base device-tree : include basic infos of SOC, aliases, rmem, gpio, uart controller, spi controller,..
    - overlay device-tree: include infos of optional devices, ex: bus-device (spi device, sdio device, i2c device),..

- Match driver to device-tree:

* + - init field ***.compatible*** in struct ***of\_device\_id*** similar to property ***compatible*** in node in device-tree.

Call MODULE\_DEVICE\_TABLE(**of**, **struct\_id\_name**)

## **Bootloader:**

* Refer: <https://www.sciencedirect.com/topics/engineering/bootloader>
* **Usage:** Main task is loading kernel to RAM.
  + - Khởi tạo phần cứng
    - Thiết lập bộ nhớ RAM (DRAM)
    - Thiết lập bộ xử lý
    - Load hệ điều hành bằng cách đọc thiết bị nhớ, từ mạng, từ serial...
* <https://www.microcontrollertips.com/what-is-an-embedded-bootloader-faq/>
  + - A bootloader performs various hardware checks, **initializes** the processor and peripherals, **configuring registers.**
    - Also used to **update MCU firmware** =>  able to communicate with some form of interface, be it [I2C](http://www.microcontrollertips.com/i2c-k-squared-c/), [SPI](http://www.microcontrollertips.com/need-gpio-mcu-try-serial-peripheral-interface-spi/), USART, USB, or some other protocol.
    - One of the main tasks of bootloaders includes **security** (ex: ARM Trusted firmware).
* **An onboard** bootloader resides in memory in an MCU in an area of ROM or flash memory that is protected from getting written over.

(BIOS itself is copied to RAM, CPU reads introductions from memory and executes them.)

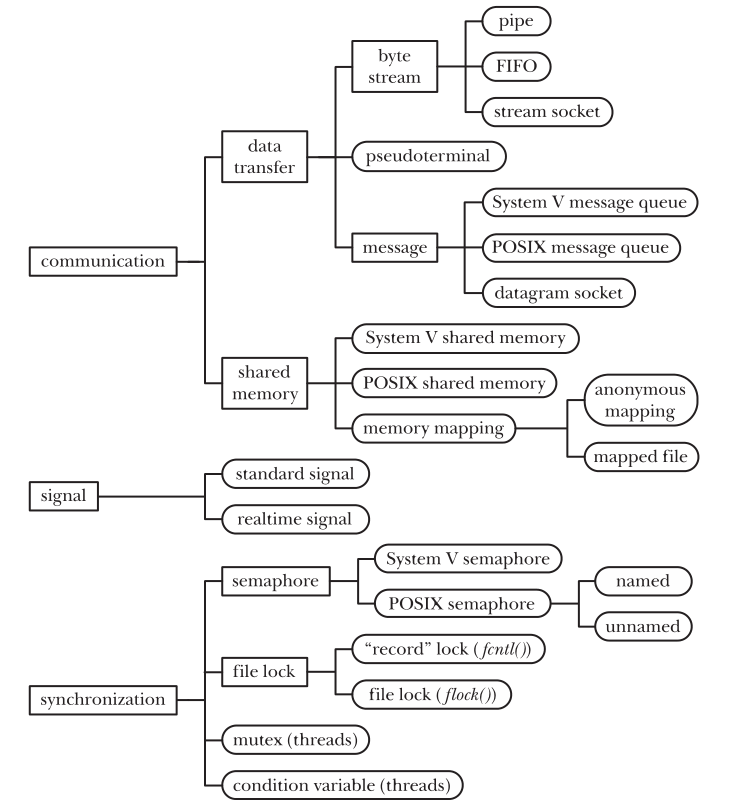
* **Booting** processing:
* Step 1: set up and initialize the RAM;
* Step 2: initialize one serial port (optional);
* Step 3: detect the machine type;
* Step 4: set up the kernel tagged list;
* Step 5: call the kernel image.
* **Types** of Bootloader: Das U-Boot, Barebox.
* Phổ biến nhất là Das U-Boot, rất nhiều chắc năng mạnh mẽ.
* Có thể truy cập được các hệ thống file phổ biến như: FAT, ext2, ext3, ext4.
* Hỗ trợ cả load kernel qua network nữa.
* Hỗ trợ cơ chế truyền tham số cho kernel bằng device tree (cho ARM) khi boot kernel.
* **Programming: ………………..**

## **Rootfs:**

## **Inter-Process communicate (IPC)**

* Refer:

<https://kipalog.com/posts/Tong-quan-ve-giao-tiep-lien-tien-trinh---Interprocess-communication--IPC>



* **3 Main Type:**
  + - Communication: Dùng để trao đổi dữ liệu giữa các process.
    - Synchronization: Hoạt động đồng bộ giữa các process.
    - Signal: Mặc dù signal sinh ra với mục đích khác, nhưng ta vẫn có thể sử dụng chúng như một công cụ đồng bộ trong một vài tình huống. Hoặc hiếm hơn là sử dụng signal như công cụ giao tiếp: signal number được coi như là một thông tin.

## **Yocto project**

* Refer: <https://lazytrick.wordpress.com/2017/01/20/so-sanh-buildroot-va-yocto-project/>
* An open source collaboration project.
* Provides templates, tools and methods to help you create custom Linux-based systems for embedded and IOT products, regardless of the hardware architecture.
* BitBake translates "recipe" files -> make a Sequential task queue.
* Poky is a reference distribution of the Yocto Project

Involves: OpenEmbedded Build System (BitBake + OpenEmbedded-Core) and a set of metadata (metadata includes: recipes, config files…)

## **Buildroot**

- Refer: <https://bootlin.com/pub/conferences/2018/elc/petazzoni-e-ale-buildroot-tutorial/petazzoni-buildroot-tutorial-lab.pdf>

**-** Buildroot is a community-driven and open-source set of tools that will help you generate an entire root file system that you can then flash into a device.

- It can build custom Linux System → 1 or all of options belows:

* toolchains,
* rootfs,
* bootloaders,
* kernels,
* libraries,..

→ we can can set our expectation when “make menuconfig”. Then, it generate “.config” like but not same as “.config” in “build kernel”, it only use for buildroot with its own variables. ex: in the rza1 project, buildroot only use for build “rootfs”.

- Note: add custom patch into buildroot:

* type $make menuconfig → Build options —> ( ) global patch directories.
* Type your path, ex: board/vendor/myboard/patches
* To upstream u-boot, create a “**uboot**” folder within the global patch directory, and copy your patches there.
* To upstream linux, create a “**linux**” folder within the global patch directory, and copy your patches there.

- Some tips: <https://www.viatech.com/en/2015/06/buildroot/>

## **Middleware**

* Refer: [https://www.eetimes.com/document.asp?doc\_id=1276764#](https://www.eetimes.com/document.asp?doc_id=1276764)
* It is any system software that is not the OS kernel, device drivers, or application software that connects software components or applications

## **Driver**

* Device Drivers are only written in C.
* Device driver presents a generic interface to the control and status registers for applications at higher level to access the devices.

## **Makefile and make**

* Refer: http://clinuxcode.blogspot.com/2017/03/make-and-make-file-interview-questions.html

->make is a build automation tool that automatically build executables and libraries by reading a file called Makefile which tells the make how to compile and link a program.  
**Q)What is a makefile?**  
Makefile is the recipe(rule to build) file, which tells the build automation tool "make" how to compile and link.  
**Q)What does a makefile contains?**  
A makefile mainly contains directives(rules) like target,dependency and and rule. It also contains "set" directive to set a variable and comments, statements stating with a #.  
Syntax of a makefile:  
*target: dependencies  
[ a tab ]recipe (system commands)*  
Where: target, is name of the executable(binary),object file or any action like "clean".  
dependency, is the input files required to create the binary and  
recipe, are the system commands that make carries out.

- CMake là một công cụ sinh Makefile đa nền tảng. Nói đơn giản thì CMake sẽ tự động tạo ra Makefiles cho project của chúng ta.

- Cmakelist.txt hướng dẫn Cmake tạo Makefile.

## **MCU/MPU peripherals:**

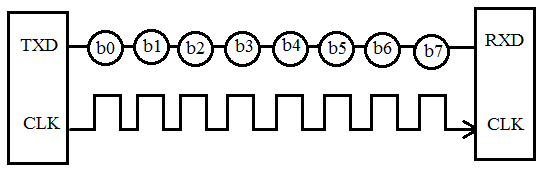
* 1. ***Comunication Protocol:***
     1. ***Serial tranfer***
* It is a technique of sending data one-by-one bit through a data line at each predetermined time.
* Data có thể được truyền qua một cáp hoặc một đường dây ở dạng bit-bit và nó chỉ cần hai cáp (full-duplex).

Figure 4: Truyền nối tiếp

* 2 Methods:
  + **Clock Synchronous:**
    - Have 2 main line:
* Clock (Control line): Inform tranfer timing. 1 clock = 1 bit.
* Data line: Send data.
  + - * Data is transmitted in synchronization with the *Clock signal* (Control signal).
      * Tranfer speed = clock speed = Tần số xung (HZ).
      * Relationship: Master (clock supplier) – Slave.
  + **Clock Asynchronous:**
    - Only 1 main line:
      * Data line: Sending and receiving sides are matched in data format and transfer rate.
      * Clock generator is inside each device.
      * Relationship: equal
* **Note:** 1 line can include serveral wire-signal.
  + - 1. ***UART***
* Universial Asynchronous Receiver Transmitter.
* Used to communicate Chip to external modules: Zigbee, Bluetooth, Wifi, gps…
* Some Concept:
  + Baud rate: Số bit truyền được trong 1s (tốc độ truyền dữ liệu đo bằng đơn vị Baud).
  + …
* Refer: https://dientuviet.com/kien-thuc-co-ban-ve-giao-tiep-uart/

https://tapit.vn/luoc-ly-thuyet-ve-chuc-nang-uart-va-mot-thanh-ghi-trong-chip-stm32f103c8t6/

* + - 1. ***SPI***
* Serial Peripheral Interface is synchronous serial comunication interface. Used for short distance, primarily in embedded systems.
* 4-Wires serial:
  + SCLK (serial clock): determine the speed of transfer.
  + MOSI (Master Output – Slave Input).
  + MISO (Master Input – Slave Output).
  + SS (Slave Select): active low.

- SPI is full-duplex (both directions). Used in typical application: sensors, memory card, display control.

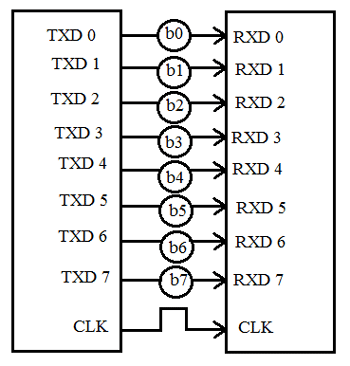
* + - 1. ***I2C (IIC)***
* Inter-Intergrated Circuit is a synchonous serial interface. 1 Master – n Slaves.
* 2-Wires of BUS:
  + SDA (Serial data line).
  + SCL (Serial clock line). Master supllies clock.
    - * Each slave device connect to the BUS has a unique address (Master device will use it to indentify the slave devices)
    1. ***Paralell tranfer***
* Data có thể được truyền theo byte (8 bit) hoặc ký tự hoặc bus tại một thời điểm (nhiều data line). Truyền dữ liệu song song tốn kém nhưng rất nhanh, vì nó đòi hỏi phần cứng và cáp bổ sung. Các ví dụ tốt nhất cho giao tiếp này là máy in cũ, PCI, RAM, v.v.

Figure 5: Truyền song song

* 1. ***Watchdog***
* A hardware timer used to detect and recover microcomputer malfuntions (due to noise, bugs, etc...)
* During normal operation, the microcomputer regularly restarts the watchdog timerto prevent it from elapsing or “timing out”. If there is hardware fault or program fault and the microcomputer cannot resart the watchdog timer. This timer will elapse and generate an Interrupt signal.
* 2 Types:
  + Build inside microcontroller.
  + Made out of microcontroller.
  1. ***TCP/IP***
* Transmission Control Protocol/ Internet Protocol.
* là một bộ giao thức trao đổi thông tin được sử dụng để truyền tải và kết nối các thiết bị trong mạng Internet.
* **Ethernet** là một dạng công nghệ truyền thống dùng để kết nối các mạng LAN cục bộ, cho phép các thiết bị có thể giao tiếp với nhau thông qua một giao thức - một bộ quy tắc hoặc ngôn ngữ mạng chung. *Là một lớp giao thức data-link trong tầng TCP/IP.*
* Refer: <https://vnpro.vn/tin-tuc/gioi-thieu-tong-quan-ve-bo-giao-thuc-tcpip-1103.html>
  1. *USB*
  2. *Socket*
  3. *PCI*

## **Interrupt**

* 2 types:
  + Hardware interrupt (acsynchronous)
  + SoftWare Interrupt - SWI (synchronous, exception)
* Process:
  + CPU nhận signal (vector) từ IPC.
  + CPU switch to kernel mode.
  + Lưu lại value các thanh ghi vào kernel stack.
  + Từ vector tra IDT -> gọi hàm sơ cấp
  + Hàm sơ cấp gọi các hàm irq\_entries -> gọi hàm thứ cấp
  + Hàm thứ cấp: Top-half (đọc ghi data ,…)-> gọi -> bottom-half( schedule by Tasklet, workqueue, softirq).
  + Restore values of PC , registers -> return to previous address -> continue to run main program.
* Workqueue vs Tasklet:
  + Tasklet:
    - Mỗi công việc chỉ được thực thi bởi chính core đã lập lịch công việc đó.
    - Mỗi công việc sẽ chiếm dụng một core từ khi bắt đầu đến khi kết thúc công việc. Do đó, ta nói công việc sẽ được thực thi ở ngữ cảnh atomic context (hay non-preemptive). Vì lý do này, trong quá trình xử lý công việc, ta không được gọi hàm **sleep** (dù trực tiếp hay gián tiếp).
  + Workqueue:
    - Theo cơ chế tasklet, core nào lập lịch cho công việc thì chính core đó sẽ thực hiện công việc ấy trong tương lai. Điều này không bắt buộc trong cơ chế workqueue.
    - Theo cơ chế tasklet, core phải thực hiện công việc từ đầu tới cuối, không được phép thực hiện đan xen các công việc khác (tức là thực thi trong atomic context). Điều này không bắt buộc trong cơ chế workqueue. Do đó, trong quá trình xử lý công việc, ta có thể gọi hàm sleep hoặc schedule.

- Refer: what is SWI?

<https://www.slideshare.net/vibrantgroupmumbai/embedded-system-introduction-to-arm-exception-handling-andsoftware-interrupts-swi>

## **Multitask concept:**

- Multitasking is the [concurrent](https://en.wikipedia.org/wiki/Concurrent_computing) execution of multiple tasks (also known as [processes](https://en.wikipedia.org/wiki/Process_(computing))) over **a certain period of time**.

- A computer executes segments of multiple tasks in **an interleaved manner**.

- Multitasking automatically interrupts the running program, saving its state and  loading the saved state of another program and transferring control to it (Most commonly, within some [scheduling](https://en.wikipedia.org/wiki/Scheduling_(computing)) scheme, one process must be switched out of the CPU so another process can run.).

-  This context switch can be triggered by the process making itself unrunnable, On a [pre-emptive multitasking](https://en.wikipedia.org/wiki/Pre-emptive_multitasking) system, the scheduler may also switch out processes that are still runnable.

1. **Context switch vs Mode switch**

- Refer: <http://www.linfo.org/context_switch.html>

- <https://stackoverflow.com/questions/31273549/what-memory-state-does-the-kernel-have-to-save-between-context-switches>

- https://en.wikipedia.org/wiki/Context\_switch

**a. Context switches:**

- A *context switch* (also sometimes referred to as a *process switch* or a *task switch*) is the switching of the [CPU](http://www.linfo.org/cpu.html) (central processing unit) from one *[process](http://www.linfo.org/process.html)* or *thread* to another.

- A *context* is the contents of a CPU's *[registers](http://www.linfo.org/register.html)* and *program counter* at any point in time.

-  *[kernel](http://www.linfo.org/kernel.html)* (i.e., the core of the [operating system](http://www.linfo.org/operating_systems_list.html)) performing the following activities with regard to processes (including threads) on the CPU:

* *~~(1) suspending the progression of one process and storing the CPU's state (i.e., the context) for that process somewhere in memory,~~*
* *~~(2) retrieving the context of the next process from memory and restoring it in the CPU's registers and.~~*
* *~~(3) returning to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process.~~*
* (1) The CPU (the actual hardware) interrupts the current process based on an internal tinier, switches into kernel mode, and hands control back to the kernel.
* (2) The kernel records the current state of the CPU and memory, which will be essential to resuming the process that was just interrupted.
* (3) The kernel performs any tasks that might have come up during the preceding time slice (such as collecting data from input and output, or I/O, operations).
* (4) The kernel is now ready to let another process run. The kernel analyzes the list of processes that are ready to run and chooses one.
* (5) The kernel prepares the memory for this new process, and then prepares the CPU.
* (6) The kernel tells the CPU how long the time slice for the new process will last.
* (7) The kernel switches the CPU into user mode and hands control of the CPU to the process.

- Context switches can occur only in *[kernel mode](http://www.linfo.org/kernel_mode.html)*.

- Context switching is an essential feature of *[multitasking](http://www.linfo.org/multitasking.html)* operating systems.

- This illusion of *concurrency* is achieved by means of context switches that are occurring in rapid succession (tens or hundreds of times per second)

- There are three potential triggers for a context switch:

* + - Multitasking
    - Interrupt handling
    - User and kernel mode switching: a context switch is not necessary; However, depending on the operating system, a context switch may also take place at this time.

**b. Mode switches:**

**- The existence of these two modes in Unix-like operating systems:**

* Kernel mode is a privileged mode of the CPU in which only the kernel runs and which provides access to all memory locations and all other system resources.
* Other programs, including applications, initially operate in *[user mode](http://www.linfo.org/user_mode.html).*

- When a system call causes the CPU to shift to kernel mode. This is referred to as a *mode switch*rather than a context switch, because it does not change the current process.

## **Firmware**

* Refer: <https://voh.com.vn/cong-nghe/firmware-la-gi-co-nen-nang-cap-firmware-hay-khong-319479.html>
* Firmware is a software program or set of introductions programmed on hardware device that makes the hardware function according to the manufacturer’s intended purposes.
* It provides the mecessary introductions for how the device communicates with the other computer hardware.
* Stored on ROM or PROM of a device. Can be erased and rewritten
* Device has one or more processor, and firmware is executed on those.

\* Note: Firmware vs Driver: <https://www.cybrary.it/2018/08/what-is-firmware/>

- Firmware is installed in the hardware of a device,

- Driver is stored inside of the operating system.

## **SoC vs MCU**

* A MCU *(microcontroller unit)* consists of a CPU, ROM, RAM, and basic peripherals etc. To integrate a micrcontroller to a system one might have to add a Wi-Fi module or some GPU chip.
* A SoC *(System-on-Chip)* will contain a microcontroller and some advanced peripherals like bluetooth, WiFi, so that the whole system is on that chip.

## **Virtualization**

## **Platform:**

- Refer: [https://www.rung.vn/dict/vn\_vn/N%E1%BB%81n\_t%E1%BA%A3ng](https://www.rung.vn/dict/vn_vn/Nền_tảng)

* Platform chính là nền tảng. Đây chính là nên tảng công nghệ sử dụng.
* Là nền tảng, cốt lõi quan trọng để những lập trình viên dựa vào đó thực hiện công việc
  + Hardware platform
  + Software platform

## **JTAG**

## - Refer: <http://nguyenquanicd.blogspot.com/2019/11/jtag-bai-1-tong-quan-ve-giao-thuc-jtag.html>

- Concept: **JTAG** (**Joint Test Action Group)** is an [industry standard](https://en.wikipedia.org/wiki/Technical_standard) for verifying designs and testing [printed circuit boards](https://en.wikipedia.org/wiki/Printed_circuit_board) (PCB) after manufacture.

- Ứng dụng:

Ngày nay, giao thức JTAG được sử dụng rộng rãi với nhiều mục đích khác nhau như:

* Program - Nạp chương trình cho chip.
* Chip test - Kiểm tra kết nối và hoạt động bên trong chip sau khi chế tạo thông qua việc điều khiển các thành phần DFT của chip.
* Debug - hoạt động của chip trong quá trình phát triển phần mềm, firmware và software.
* Connection test - Kiểm tra kết nối khi chip tích hợp trên bo mạch in.

- Thành phần cần có khi sử dụng JTAG:

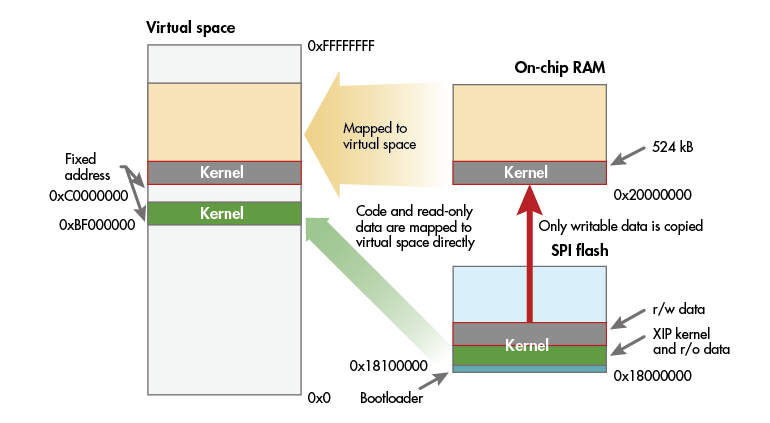
* PC đã cài phần mệm giao tiếp JTAG cd:, Trace32 tricore,...
* Chip teMột phần cứng kết nối giữa máy tính và bo gắn chip cần test. Phần cứng này gọi là “bus master” trong IEEE 1149.1. Nó có nhiệm vụ nhận các gói dữ liệu và điều khiển từ phần mềm máy tính thông qua USB, PCI, Ethernet, … và thực thi điều khiển chip thông qua kết nối JTAG.
* Board có chip cần test.

- Implement JTAG into a Chip: chip cần

1. Một TAP – giao tiếp JTAG (gồm 4 tín hiệu bắt buộc: TMS, TCK, TDI,TDO).
2. Một TAP controller – bộ điều khiển giao thức JTAG.
3. Instruction Register - IR)– thanh ghi chứa lệnh (instruction) sẽ được thực thi.
4. Data register – DR.

## **XIP**

* Refer: https://www.electronicdesign.com/technologies/embedded-revolution/article/21805857/xip-with-linux-a-new-spin-on-embedded-architectur
  1. Purpose:
* XIP Linux was first added to the Linux kernel for the PowerPC, with 2 purpose:
  + Speeding up boot time.
  + Reducing RAM usage.
* Later, this capability was extended to the ARM tree.
  1. Concept:
* Execute-in-Place (XIP) Linux with AXFS enables embedded systems to run Linux within a memory-constrained system by executing most of the code in-place from flash memory like an MCU.
  1. Action:
* It allows code to be executed directly from external serial flash.
* The code and constants can be kept in CPU-accessible ROM.
* Only writeble data are copied to RAM.



* 1. **Snap**
  2. **Concept**: gần giống với **apt** trong ubuntu
* Snap is a software [packaging](https://en.wikipedia.org/wiki/Package_manager) and [deployment](https://en.wikipedia.org/wiki/Software_deployment) system developed by [Canonical](https://en.wikipedia.org/wiki/Canonical_(company)" \o "Canonical (company)) for [operating systems](https://en.wikipedia.org/wiki/Operating_system) that use the [Linux](https://en.wikipedia.org/wiki/Linux" \o "Linux) kernel.
* The packages, called *snaps*, and the tool for using them, *snapd*, work across a range of [Linux distributions](https://en.wikipedia.org/wiki/Linux_distribution" \o "Linux distribution)
* allow [upstream](https://en.wikipedia.org/wiki/Upstream_(software_development)) software developers to distribute their applications directly to users.
* Snaps are self-contained applications running in a sandbox with mediated access to the host system.
  1. **Practice**:
* [Create your first snap | Ubuntu](https://ubuntu.com/tutorials/create-your-first-snap#1-overview)

# **Programming Knowledge** **C/C++**

## **Object Oriented Programming (OOP trait)**

### **Inheritance:**

* Inherit member and method from base class.
* 2 types:
  + ***Multiple level Inheritance:* B inherit A Then C inherit B → C can access ALL in A and B.**
  + ***Multiple Inheritance* (Only in C++): Class C: public B, public A;**

**- Trong TH kế thừa 2 lần 1 class: A inherit B,C mà B,C cùng inherit D thì sẽ compile lỗi**

**→ Cần add từ khóa virtual khi khai báo class B,C:**

**class B: virtual public D{};**

**class C: virtual public D{};**

### **Polymorphism:**

* 1 method = A lot of usages.
* 2 types:
* *Overloading*: Same: *Name* , but different implement & *parameter (Return type* or *number or type of parameters)*. Define trong 1 Class hoăc là các hàm global bình thường.
  + **Operator overloading**
  + **Method overloading**
* *Overriding***:** Same: *Name & parameter*, but different *implement*. Define ở các class khác nhau.
  + **virtual**: keyword to declare a virtual method (can be overided).
  + ***pure virtual function:*** *virtual method\_name () =0;*

ERROR if not define in Child class)

* + Referred to as dynamic linkage, or late binding.
  + Muốn có *type of parameters khác nhau thì nên khai báo type void\* trong parent class. đến khi override trong child class thì cast về type mong muốn.*

### **Atraction:**

* Refer: <https://yellowcodebooks.com/2017/09/26/java-bai-32-tinh-truu-tuong-abstraction/>
* Concept use in design phase:
  + Bóc tách **các** đối tượng thực tế
  + Tìm được những data&method **chung**
  + **Gom** chúng lại, tạo thành Attract-class.
* Attract class trait:
  + KO ý nghĩa hoặc KO thể dùng tạo object.
  + Có ít nhất 1 Attract-method inside.( vd pure function)
  + Đặc trưng cho nhóm đối tượng đó.
* Purpose
  + Only **khung sườn** cho các class con.
  + Optimize code, tạo tính liên kết.

### **Encapsulation:**

* Data hiding
* **Prevent** access directly member (private), must through method.

MIX (Inheritance + Polymorphism(virtual)) **(Confirm lan 2 Ok)**

|  |  |
| --- | --- |
| Asume | *Class Parent{};*  *Class Child : public Parent {};* |
| Type 1 | Accessibility of Child object:  *Child derive;* // Parent(constructor) -> Child(constructor).  ….  *[Out of scope]*  // Child(de-constructor) -> Parent(de-constructor).  Content:  - *"derive”* Obj can access to *ANYTHING* within both (*Parent* & *Child* class). |
| Type 2 | Accessibility of Parent pointer: ( to member of child and parent)  *Parent\* Base = new Child(); | Parent(constructor) -> Child(constructor).*  *….*  *delete(Base); (freed memory)*  *Base = NULL; (pointed dangling ptr to NULL) Only Parent(de-constructor).*  *Or [Out of scope]*  Content:  - "Base pointer" Only access to "Parent\_method".  - Add "virtual" in front of "that Parent-method" --> Access to Overided\_in\_Child. |
| Type 3 | Direct access permission to Parent members:   |  |  |  |  | | --- | --- | --- | --- | | Access from | public | protected | private | | Same parent class | O | O | O | | Derived classes | O | O | X  (But can access via other public,protected member of parent) | | Outside classes | O | X | X |     A derived class inherits all base class methods with the following exceptions:  - Constructors, destructors and copy constructors of the base class.  - Overloaded operators of the base class.  - The friend functions of the base class. |
| Type 4 |  |

**\*\*\* Giải thích:**

* Inherit:
  + Tại vì class child kế thừa các member của Parent nên khi khai báo 1 Object child thì

=> constructor của parent cũng được gọi để khởi tạo các member cho child kế thừa.

* + Lưu ý constructor parent này là mặc định ko đối số -> cần phải viết 1 consructor ko đối số cho class parent. hoặc cả parent và child đều ko viết constructor -> Ko viet compile error
  + De-constructor của parent cũng đc gọi cùng với de-constructor của child khi Object child bị hủy.
* Encapsulation and information hiding:
  + Các đối tượng khác nhau, không có quyền chỉnh sửa dữ liệu thành viên của nhau,
  + Nó chỉ có thể truyền thông điệp cho nhau mà thôi.

-> đối tượng khác muốn thay đổi thuộc tính thành viên của đối tượng nội tại, thì chúng cần truyền thông điệp cho đối tượng, và việc

* + Quyết định thay đổi hay không vẫn do đối tượng nội tại quyết định.

### **Static member**

* 1. **Static variable:** is shared by all objects of the class.
* All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done.

* 1. **Static function:** can be called even if no objects of the class exist.
* Các hàm non-static được gọi thông qua 1 Object, còn hàm static được gọi thông qua toán tử phạm vi ”::”.
* A static member function can only access static data member, other static member functions and any other functions from outside the class.

|  |
| --- |
| Example:  class A{  static int cnt;  int id;    A(){  id = cnt;  cnt++;}  .....  static void func(){  cout << cnt;}  };    int A::cnt = 0;  A::func(); |

***\*\*\*Note:*** When using container with a class: vector<A> var, And gonna sort(var.begin(),var.end()); => Need to overload operator "<"

|  |
| --- |
| Example:  bool operator <(A &a){  return this->id <a.id;  } |

### **Allocate and de-allocate**

* Toán tử new và new[] --- Toán tử delete và delete[]

|  |
| --- |
| type \*num = new type; //Khai báo biến trỏ type voi sizeof(type) byte  type \*num = new type(arg1,arg2,..); //Tương tự trên nhưng type CÓ constructor.  type \*nums = new type[10]; //Khai báo biến trỏ nums với sizeof(type)\*10 bytes  =>  delete num;  delete[] nums; |

### **Con trỏ this:**

* Trỏ vào dữ liệu thành viên của chính nó (ko để ý access level), và chỉ có phạm vi tác dụng trong một lớp.
* Chỉ hoạt động với các dữ liệu và hàm thành viên không tĩnh (non-static)

### **Hàm Friend:**

* Được phép truy cập tất cả member của ClassA (ko quan tâm accsess level)

|  |
| --- |
| Declare:  friend int sum(ClassA); **//khai báo prototype trong ClassA.**  Define:  int sum(ClassA x) {return x.a +a.b;} **// Define bên ngoài ClassA.** |

### **Class friend:**

|  |
| --- |
| Declare:  Class A{  int aa;  friend class B;  };  **// khai báo trong classA, classA là bạn của classB (Nhưng chú ý B ko phải là bạn của classA)**  Define:  class B{  void show( A x){  cout << x.aa; }  **// Here, ở trong classB, obj của classA có thể access tới all member y như đang ở trong classA (ko qt access level).**  }; |

### **Const Reference**

* Giải thích: đối tượng tham chiếu sẽ tham chiếu đến địa chỉ của đối tượng gốc.
  + - * + Dữ liệu của đối tượng tham chiếu sẽ được ánh xạ theo địa chỉ của đối tượng được tham chiếu (không thực hiện việc sao chép trực tiếp mà là gián tiếp thông qua địa chỉ của biến tham chiếu). Tuy nhiên, cũng vì lí do này mà đối tượng được tham chiếu có thể bị thay đổi giá trị (tương tự như truyền theo tham biến)
        + **Vi phạm tính đóng gói trong lập trình hướng đối tượng**.(các đối tượng khác nhau, không có quyền chỉnh sửa dữ liệu thành viên của nhau, nó chỉ có thể truyền thông điệp cho nhau mà thôi)
        + C++ cung cấp cho ta từ khóa const để quy định một đối tượng khi được tham chiếu sẽ không bị làm thay đổi member\_data nó được gọi là tham chiếu hằng.

#### **Biến:** refer thông qua địa chỉ truyền vào, nhưng ko cho phép thay dổi giá trị.

Hiện tại ta có 3 loại tham chiếu trong define argument của Hàm :

- Tham trị: ko làm thay dổi giá trị của Obj (argument) khi out hàm.

- Tham chiếu: dữ liệu của Obj (argument) có thể bị thay đổi trong khi thực thi Hàm và nó được lưu lại.

- Tham chiếu hằng: tương tự như tham chiếu nhưng không cho phép thay đổi dữ liệu của Obj (argument) .

#### **Hàm:**

- Hàm của đối tượng được tham chiếu hằng (const ClassA &p), sẽ ko đc phém gọi trong Hàm.

tham chiếu bằng con trỏ hoặc reference (ClassA \*p OR ClassA &p) thì cứ dùng BT.

-> Nếu arg define dạng tham chiếu Hằng mà mà muốn dùng hàm của Obj tham chiếu đó thì phương thức đó phải thêm "const" sau đuôi.

vd: int GetTu(void) const;

## **Specifier keywords:**

### **Static:**

#### **Variable:**

#### Static Global: chỉ được dùng trong file đó, Nhưng ko extern đc từ File khác.

#### Static in function: vùng nhớ đều ko hủy khi thoát khỏi hàm, Nhưng static var chỉ hoạt động trong function của nó.

#### **Function**

#### Nó là hàm local trong file nó đc khai báo, KHÔNG dùng được cho file khác.

### **Inline**

* Hàm nội tuyến, trình biên dịch sẽ khởi tạo một thân hàm và chèn nó vào vị trí được gọi tại mỗi thời điểm mà hàm đó được gọi, thay vì nó chỉ chèn lời gọi hàm (giống function macro). Việc làm này sẽ skip argument passing and stack mantainance → improve execution time but it increases code size.

|  |
| --- |
| Example:  ***inline*** type tên\_hàm(danh\_sách\_tham\_số)  {  Thân hàm;  } |

### **Volitate**

- Tells the compiler not to optimize when applied to a variable. By declaring a variable volatile, we can tell the compiler that the value of the variable may change any moment from outside of the scope of the program.

- Volatile keyword is useful for memory-mapped peripheral registers, global variables modified by an interrupt service routine, global variables accessed by multiple tasks within a multi-threaded application.

### **Const**

- Means that the value of the variable will not be changed. But the value of the variable can be changed using a pointer. The const identifier can be used like this:

|  |
| --- |
| const int a; or int const a;  //Both means the same and it indicates that a is an constant integer.  const int \*p; or int const \*p  //pointer is pointing to an constant integer.  int\* const p;  //The const pointer to a non-constant integer.  int const \* a const;  //a constant pointer to a constant integer. |

### **attribute**

### **Extern**

* Refer: <https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/>
* Variables and function can be declared number of times but defined only. If not, throw error “Multiple defination”. Ie:

+ Trong 1 file src, funtion hoặc var ko thêm static thì nó là global (hay external linkage) -> file khác trong cùng được link nếu re-define -> error. **( Tested OK)**

* Extends the visibility of the variables and functions.

File src nào muốn dùng var và function được define ở 1 file src khác thì nó cần có 2 điều kiện:

* + Var và function đó ko bị giới hạn local bởi từ khóa “static”.
  + Có declare của var hoặc function đó. (extern trực tiếp trong file src or include 1 file header chứa declararion của var hoặc function đó).

#### **Variable:**

#### External variables can be declared number of times but defined only once

+ This is declaration: can do this declaration as many times as we want.  (extern keyword explicitly when we want to declare variables without defining them).

|  |
| --- |
| Extern in var; |

+ This is defination:  the memory for var is also allocated. Only once in a scope (local< file< program);

|  |
| --- |
| int var;  int var = 0; // a number |

+ This include defination and delaration: Only once in a scope and throw a warning. (extern này chỉ thông báo với compiler là nó là biến extenal linkage -> file src khác muốn xài thì phải có declare của nó)

|  |
| --- |
| main.c:2:12: warning: ‘x’ initialized and declared ‘extern’  extern int x = 32; |

#### **Function:**

* a function is declared or defined, the extern keyword is implicitly assumed.
* The function’s visibility to the whole program, provided those files contain a declaration of the function (you need to add that funtion’s declaration at this src file or inlcude a header contains that declaration). If not, run OK but compiler throw a warning “implicit declaration”.

## **Data structure**

### **String**

#### **Char in C/C++:**

#### Ko thể asign giá value hoặc so sánh trực tiếp: chỉ dùng hàm built-in.

|  |
| --- |
| Example:  const char a[] ="aa";  const char b[] ="aa";  So sánh:  NOT if (a == b) printf("Hello World");  OK if (strcmp(a,b) == 0) printf("Hello World");  Asign:  NOT b ="aa";  OK strcpy(b,a); |

#### **String type is only C++**

#### String thực chất là một vector<char> có bổ sung thêm một số hàm và thuộc tính, do đó, nó có toàn bộ các tính chất của 1 vector, như hàm size(), push\_back(), toán tử [], ...

### **Array**

#### **Concept:**

#### An array is a container object that holds a fixed number of values of a single type.

#### **Declaration:**

* Type name[N]; → The above array is a **static** array that has memory allocated **at compile time**.
* Note: trên các Conline compiler support **N là 1 biến** → A **dynamic array** can be created in C, using the malloc function and the memory is allocated **on the heap at runtime.**

### **Struct**

#### **Concept:**

#### Struct is a way to combine multiple fields to represent a composite data structure, which further lays the foundation for Object Oriented Programming.

#### **Declaration:**

#### Như C; ko cần typedef - dùng trực tiếp để declare biến

|  |
| --- |
| Example:  struct strA{  int b;  };  -> strA x; --> ko có cần "struct strA x" như C. |

#### Bên trong có thể viết hàm con và constructor;

|  |
| --- |
| Example:  struct strA{  int b;  strA(int n){b=n;}  void hello(){ cout <<"ahihi";};  }; |

#### **Usage:**

|  |
| --- |
| Define a pointer:  strA \*x = new strA; // allocate vùng nhớ cho con trỏ x, ko cần construcror; ( new usage: type \*x = new type;)  strA \*x = new strA(2); // allocate vùng nhớ cho con trỏ x, sẳn tiện init giá trị cho member bên trong b =2;  Call struct’s func:  x->hello(); // x is a struct pointer  or  strA x = strA(2);  x.hello(); // x is an instance of struct |

- Vector struct:

|  |
| --- |
| Example:  Declare:  Vector <strA> x;  Allocate n size:  x = vector<strA> (n); |

### **Union**

* User-defined data type in C, which is used to store a collection of different kinds of data, just like a structure. However, with unions, you can only store information in one field at any one time.
* Memory allocated for the union is equal to memory needed for the largest member of it, and all members share this same memory space.
* In C, struct and union types cannot have static members. In C++, struct types are allowed to have static members, but union cannot have static members in C++ also.
* Difference between Struct vs Union:

*https://www.geeksforgeeks.org/c-language-2-gq/structure-union-gq/*

|  |
| --- |
| Example:  union Courses  {  char WebSite;  int Price;  };  union Courses x;  x.WebSite =1;  x.Price =10;  printf("Hello world %d - %d\n", x.Price, x.WebSite);  x.WebSite =1;  printf("Hello world %d - %d", x.Price, x.WebSite);  => output:  Hello world 10 - 10  Hello world 1 – 1  **→ only store information in one field at any one time.** |

### **Enum**

* An user-defined data type which can be assigned some limited values.

|  |
| --- |
| **// Defining enum1 Gender**  enum Mode {  None,  Active,  De-active  }; |

* Some restrictions:
  + Two enumerations cannot share the same names.
  + No variable can have a name which is already in some enumeration.
  + Enums are not type safe. (can implicit conversion to int, can compare between 2 vars of different Enum).
    - * + Those are reasons why we have **Enum class**: Its use is contrary to the stated restrictions.

|  |
| --- |
| **Declaration**  enum class Color{ Red, Green, Blue};  **Initialisation**  Color col = Color::Red; |

### **Pointer (\*) vs Reference (&):**

Sizeof(pointer) is always 8 bytes:

sizeof(char\*) = sizeof(int\*) =… = 8 bytes

#### **General Features:**

* Pointer: Can be NULL and can be re-assigned many times.
* Reference: (Not exist in C) Always refers to an object, CANNOT be re-assigned after binding.
* There's no "reference arithmetic" (but you can take the address of an object pointed by a reference and do pointer arithmetic on it as int &obj + 5).

#### **Usages:**

* Use references in function parameters and return types to provide useful and self-documenting interfaces.
* Use pointers for implementing algorithms and data structures.

| **Reference** | **Pointer** |
| --- | --- |
| The value of a reference cannot be reassigned | The value of a pointer can be reassigned |
| It can never hold a *null*value as it needs an existing value to become an alias of | It can hold or point at a *null* value and be termed as a *nullptr* or *null pointer* |
| It cannot work with arrays | It can work with arrays |
| To access the members of class/struct it uses a ‘**.**‘ | To access the members of class/struct it uses a ‘**->**‘ |
| The memory location of reference can be accessed easily or it can be used directly | The memory location of a pointer cannot be accessed easily as we have to use a dereference ‘**\***‘ |

Some api info:

- Allocate a pointer: type \*var = (type\*) malloc(size(type)\*num\_element);

→ reallocate: var = realloc(var,sizeof(type)\*new\_num);

→ realloc() ko làm mất data ở những block còn lại (nếu size mới < size cũ).

### **Smart Pointer**

#### **Usage of Pointer:**

* Access to memory in heap (dynamic memory).

#### **Problem:**

* **Memory leaks**: allocated memory for pointer but never free it during running program.
* **Dangling pointer**: after de-allocate/free the pointed memory but don’t point that pointer to NULL or nullptr (usually occurs at the [object destruction](https://www.geeksforgeeks.org/exception-handling-and-object-destruction-in-c-set-1/) time.)
* **Wild Pointers:** Declared but never point to a valid memory (or allocated by new in C++)
* **Buffer Overflow**: User use that pointer to access outside of allocated memory.

#### **Type of Smart Pointer:**

* Unique\_ptr:
  + Make sure that only exactly one copy of an object exists (Holding by that unique\_ptr)
  + if  already holds a pointer to an existing object, this object is deleted first and then the new pointer stored -> using **reset(…)**
  + Does not support copying (operator is deleted in wrapper class). But support **move** to give ownership to another unique\_pr.
* Shared\_ptr:
  + Reference-counted smart pointer. Use when you want to share the raw pointer (object) to multiple owners.
  + The raw pointer (object) is not deleted until all shared\_ptr owners have gone out of scope.
* Weak\_ptr:
  + is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by [std::shared\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr" \o "cpp/memory/shared ptr)
  + is used to track the object, and it is converted to [std::shared\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr" \o "cpp/memory/shared ptr) to acquire temporary ownership. If the original [std::shared\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr" \o "cpp/memory/shared ptr) is destroyed at this time, the object's lifetime is extended until the temporary [std::shared\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr" \o "cpp/memory/shared ptr) is destroyed as well.

Note:

* Java or C# have mechanism to deallocate unused memory call as **Garbage Collection Mechanisms**

## **Some features**

### **Function macros:**

- Function-like macros can take arguments, just like true functions. To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like.

- Is a macro, that causes nothing else but a simple textual replacement within your code.

- Macros are "resolved" by pre-processor.

|  |
| --- |
| Example:  #define Max(a,b) ((a)>(b)) ? (a):(b))  -> int a = Max(3,4); |

### **Function pointer:**

|  |
| --- |
| Ex:  void p1(){cout << "haha" << endl;}  void p2(){cout << "hihi" << endl;}    =>  void (\*funp)(void);    funp = p1; // or funp = &p1;  funp();  funp =p2;  funp(); |

### **Variadic Function:**

**-** A [function](https://en.wikipedia.org/wiki/Function_(programming))of indefinite [arity](https://en.wikipedia.org/wiki/Arity), i.e., one which accepts a variable number of [arguments](https://en.wikipedia.org/wiki/Argument_(computer_science)).

|  |
| --- |
| #include *<stdarg.h>*  #include *<stdio.h>*  double average(int count, ...) {  va\_list ap;  int j;  double sum = 0;  va\_start(ap, count); */\* Requires the last fixed parameter (to get the address) \*/*  **for** (j = 0; j < count; j++) {  sum += **va\_arg**(ap, int); ***/\* Increments ap to the next argument. \*/***  }  va\_end(ap);  **return** sum / count;  }  int main(int argc, char **const** \*argv[]) {  printf("%f**\n**", average(3, 1, 2, 3));  **return** 0;  } |

### **Scope rules:**

* Refer: <https://www.w3schools.in/c-tutorial/variable-scope/>
  + File scope
  + Block scope
  + Funtion prototype scope
  + Funtion scope

### **Tham Biến vs Tham Trị:**

* Tham trị: truyền giá trị vào hàm.

|  |
| --- |
| Ex:  void swap(int a, int b); |

* Tham biến: truyền địa chỉ vào hàm.

|  |
| --- |
| Ex:  void swap(int &a, int &b); --> Reference : Only C++  void sum(int \*a, int \*b); --> Pointer : C & C++ |

### **Exception**

* Keywords: try - throw – catch
  + throw:
    - built-in C++ standard (std::exception)
    - defined-in yourself

* + catch(ex\_type &e): 3 types
    - Expexcted thow type : standard or new-define.
    - std::exception : All standard
    - (...) : anything

|  |
| --- |
| New-define Example:  Inherit:  struct MyException : public exception {  const char \* what () const throw () {  return "C++ Exception";  }  };  Alsolute new:  struct MyException {  const char \* what (){  return "C++ Exception";  }  }; |

### **Standard Template Library (STL)**

#### **Algorithms**

* Generic function templates for operating on containers.
* Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

#### **Containers**

- Generic class templates for storing collection of data.

* Used to manage collections of objects of a certain kind.

##### **Vector**

- Similar to a C array, but dynamic.

- Provides an alternative to the built in array.

- A vector is self grown.

- Support iterators, .. and only push\_back(), pop\_back().

##### **Pair**

*-* Stores a pair of objects, first of type T1, and second of type T2.

- No support iterators, ex: begin(), end(),..

|  |
| --- |
| struct pair<T1, T2>{  T1 first;  T2 second;}; |

##### **List**

- Lists are sequence containers that allow non-contiguous memory allocation. Normally, when we say a List, we talk about doubly linked list.

- Support iterators, and push\_back(), push\_front() & pop\_back(), pop\_front(0,...

##### **Map**

- Most useful when we want to store (and possibly modify) an associated value.

- We provide a key/value pair. The key serves as an index into the map, the value serves as the data to be stored.

- Support iterators, and insert().

##### **Link List**

- Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

Example: <https://onlinegdb.com/ByF4HddUw>

##### **7.2.6 Stack**

- Stacks are a type of container adaptors with LIFO type of working,

- Support iterators.

- [push(g)](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/) Adds the element ‘g’ at the top of the stack.  
- [pop()](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/) Deletes the top most element of the stack.

##### **7.2.7 Queue**

- Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement

- Support iterators.

- **push()**  adds the element ‘g’ at the end of the queue.

- **pop()**  deletes the first element of the queue.

##### **7.2.8 Set**

- Sets are containers that store unique elements following a specific order.

- Ko chứa member trùng lặp.

- Support iterators, and insert().

#### **Functions**

##### **7.3.1 Sort**

|  |
| --- |
| container<type> var(n);  **#Sort with simple type:**  sort(var.begin(),var.end());  **#Sort custom with struct type:**  // Use compare function here;  bool comp((type& ws1,type& ws2){ return ws1.memberX < ws2.memberX;}  sort(var.begin(),var.end(),comp);  // use lambda function here;  sort(var.begin(),var.end(),[](type& ws1,type& ws2){return ws1.memberX < ws2.memberX;}); |

#### **Iterator**

* Generalized ‘smart’ pointers that facilitate use of containers.
* They provide an interface that is needed for STL algorithms to operate on STL containers.
* Provide *a general way* for accessing each element in sequential (vector, list) or associative (map, set) containers.

|  |
| --- |
| container::iterator it;  for (it = container\_var.begin(); it != container\_var.end(); ++it) {…} |

Note for container:

* syntax: container<type> var(n), Which can not accepted in defining a STRUCT.

→ that can instead of “new” → type\* var = new type[n];

### **Namespace**

* Phân vùng source code, giải quyết những trường hợp cùng tên hàm nhưng khác nội dung, ý nghĩa.

### **Template**

* Refer:
  + <https://www.geeksforgeeks.org/template-specialization-c/>
  + https://www.geeksforgeeks.org/templates-cpp/
* “template” là từ khóa báo cho trình biên dịch rằng đoạn mã sau đây định nghĩa cho nhiều kiểu dữ liệu và mã nguồn của nó sẽ được compile sinh ra tương ứng cho từng kiểu dữ liệu trong quá trình biên dịch.
* Template specialization: It is possible in C++ to get **a special behavior** for a **particular data type.** This is called template specialization.

|  |
| --- |
| **A generic sort function**  template <class T>  void sort(T arr[], int size)  {  // code to implement Quick Sort  }    **Template Specialization: A function specialized for char data type**  template <>  void sort<char>(char arr[], int size)  {  // code to implement counting sort  } |

## **Advance C++**

### **Dispatching**

* Refer: [Understandig Virtual Tables in C++ | pablo arias (pabloariasal.github.io)](https://pabloariasal.github.io/2017/06/10/understanding-virtual-tables/)
* The compiler will create a routine for function() and remember its address. This routine will be executed every time the compiler finds a call to foo() on an instance of A. Keep in mind that only one routine exists per class method, and is shared by all instances of the class.
* **Dispatching** just refers to the action of finding the right function to call
* **Static dispatch** or **early binding**: the compiler knows which routine to execute during compilation.
* There is some cases (declare virtual function in Base class), Compiler can NOT know which routine will be called by Base\* -> The compiler has to find the right function definition (i.e. the most specific one) at runtime that is **Dynamic dispatch** or **Late binding**

### **Vtable**

* For every class that contains virtual functions, the compiler constructs a *virtual table*, a.k.a *vtable*
* **The *vtable* contains an entry/address for each virtual function accessible by the class and stores a pointer to its definition.**
* Entries in the *vtable* can point to either functions declared in the class itself (e.g. C::bar()), or virtual functions inherited from a base class (e.g. C::qux()).
* **Note:** vtables exist at the class level, meaning there exists a single vtable per class, and is shared by all instances.
* Every time the compiler creates a *vtable* for a class, it adds an extra argument to it: a pointer to the corresponding virtual table, called the ***vpointer***.
* When a call to a virtual function on an object is performed, **the *vpointer* of the object is used to find the corresponding *vtable* of the class.**

### **Design Principle**

#### **3.1 SOLID**

[SOLID Principles in Programming: Understand With Real Life Examples - GeeksforGeeks](https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/)

#### **3.2 SoC – Separation of Concern**

### **Design Patern**

#### **4.1 Creation**

4.1.1 Singleton pattern

*Ensure a class only has one instance, and provide a global point of access to it*

a. Problem:

* Only need 1 Instance for that Class to manage 1 Resource.
* Able to access that instance like Global variable.

c. Solution:

* Prepare: In that Class declaration:
  + Constructor will be in Private to prevent new operator.
  + Static T& getInstance(): Allow call from outside then return back “static T instance”
* Usage:
  + T::getInstance().T’s\_method()

c. Achieve:

* Maintain/Share 1 resource by 1 unique Instance.
* Call wherever in program but NOT global variable.

4.1.2 Factory pattern

a. Problem:

* A Service with common Methods changed respect to type of Worker.
* Different between Service’s methods and Worker’s methods.

c. Solution:

* Prepare:
  + 1 Interface for Worker:
    - Worker’s Methods.
  + 1 Interface for Service, include:
    - Service’s Methods (based on Worker) -> Completed implementation here -> So this pattern is only showed in virtual creator function as below:
      * virtual Worker createWorker() = 0; (Different Service will have different Worker, then have different creator).
  + NOTE:
    - createWorker() is more efficiency in cases:
      * Lazy Instantiation: only create the Worker **when it's needed.**
      * Often&Different Customization of Worker Constructor in Service’s Method.
    - Maintain an instance of Worker in Concrete Service is more efficiency if:
      * All Service’s Methods must use the same Worker instance.
      * Need cover about threadsafe for resource – Worker.
      * Maintain lifecycle of Worker.
      * it would no longer strictly be a **Factory Method Pattern**, but it could still be considered a variant of a **Factory Pattern**,
* Usage:
  + There will be multiple concreted Service.
  + Client code will use Service\* and perform common Methods.
  + Client code will create Service\* with its expected Service concretion.

c. Achieve:

* High level abstraction for Service, then no much change in client code except for changing type to new type of Concrete Service.
* Easier to add more Concrete Service and its Concrete Worker (Open/Closed principle).

4.1.3 Abstract Factory pattern

a. Problem:

* A Serive based on a family/collection Objects (Worker, Material, Tools,..)
* Multiple factory methods: createWorker(), createMaterial(), createTool(),… in a Service.

4.1.4 Builder pattern

Refer: [Builder pattern: A pragmatic approach | Dimitris Platis](https://platis.solutions/blog/2023/01/03/builder-pattern-cpp/)

a. Problem:

* You want to create an Object but it depends on Options/Parameters, then result Objects will be different.
* Number of customizations is big and maybe rise in future, would say it’s complicated to create.

b. Solution – Basic Builder is integrated in Object class:

* Prepare: Create an interface Object:
  + *Static Object createObject(T);*
  + *Object& setOption1(X);*
  + *Object& setOption2(Y);*
  + *Object& setOptionN(Z);*
* Usage: From client code,
  + *auto Instance = Object::create(T)*

*. setOption1(X)*

*. setOption2(Y)*

*.setOptionN(Z);*

c. Solution – Traditional Builder is separate class and return Object:

* Prepare:
  + ObjectBuilder is friend of Object class -> able to modify “all level” data in Object (due to customize Object before return to Client)
  + ObjectBuilder class has:
    - ObjectBuilder *& setOption1(X); Modify Option in Object.*
    - ObjectBuilder *& setOption2(Y);*
    - ObjectBuilder *& setOptionN(Z);*
    - **std::unique\_ptr**<Object> build(): return  *to Client.*
    - **std::unique\_ptr**<Object> obj\_instance;
* Usage:
  + *auto Instance = ObjectBuilder(T)*

*. setOption1(X)*

*. setOption2(Y)*

*.setOptionN(Z);*

*.build()*

c. Achieve:

* Dynamic construct/custom Object with multiple variants.
* Isolate complex construction from Object business logic (Single Responsibility).

#### **4.2 Structure**

4.2.1 Proxy pattern

a. Problem:

* Create a wrapper for a service (External/3rd party lib)
* Not frequently calling.
* Want to add some additional action (Manage lifecycle, Logging, lazy init, caching, access control,…)

c. Solution:

* Prepare:
  + Class as Service\_Proxy:
    - All Service’s Methods (Call to Real\_Services’s Methods)
    - Additional function and variable for additional logic
  + (Optional) If that’s not an external lib, Service\_Proxy will inherit Real\_Service’s Interface. Then Client have use Real\_Service’s Interface (with concrete class is Service\_Proxy) for its usage.
* Usage:
  + There is nothing really different from Client, It just call method via Real\_Service’s Interface (with concrete class is Service\_Proxy).

c. Achieve:

* Proxy will control (lifecycle, additional logic...) Service Object without Clients knowing about it.
* Open for introducing new Proxies (update additional logic) without impact to Clients or Real\_Service (Open/Closed Principle).

4.2.2 Bridge pattern

a. Problem:

* This is BASIC for Abstraction-Implementation concept.
* Your App/Module is combination from multiple dimensions/ sides/ aspects, ex:
  + Shape+Color -> BlueSquare, RedSquare, BlueRectangle, RedRectangle, …
  + HighLayer + LowLayer: SignalDataUser+SignalEngine,
* As you can see, a monolithic code-base will be hard for changes than modular design.
* And also, above problem leads to duplicated code lines.

c. Solution:

* Prepare:
  + Determine the layers or dimensions: Abstraction(interface) + Implementation (Platform).
  + Create their interfaces (In programming - > Abstract classes) for them.
    - Abstraction Layer’s interface:
      * A variable of Implementation Layer’s interface.
      * Methods that use APIs from Implementation Layer’s interface.
    - Implementation Layer’s interface:
      * Logic Methods.
    - Additional function and variable for additional logic
* Usage:
  + In Client, it just calls methods via Abstraction Layers.

c. Achieve:

* Create a platform-independent module.
* Any change will be performed independently without any impact to other Layer/Dimension.
* Easy to introduce new concrete in Layers (Open/Closed Principle\_.
* Isolate business logic in separate Layers.

#### **4.3 Behavior**

4.3.1 Observer pattern

a. Problem:

* Target for Event-Based problem.
* There is 1 event Subject but multiple Observers.

c. Solution:

* Prepare:
  + Interface for Subject who directly listens to Event (Only need 1 concretion for 1 type of event):
    - add(Observer) : into List.
    - remove(Observer): from List.
    - notify(): Loop in List and call Observer.update(T).
  + Interface Observer who listens Event via Subject (Multiple concretion)
    - update(T): called by Subject.notify(), do some business logic.
* Usage:
  + Initialize for Subject that manage Event source.
  + Initialize Observer, then add/remove them in Observer’s List.
  + When Event comes, Subject.noptify() will Observer.update(T)

c. Achieve:

* Avoid multiple Observers have to self-maintain a logic to Event source.
* Save resource due to NOT polling.
* Easier to add/remove Observer (Open/Closed Principle).

4.3.2 Visitor pattern

a. Problem:

* There is 1 or some-future Method/Feature is alien to be introduced/added in 1/collection Object.
* These Method/Feature maybe impact to that fragile Object.

c. Solution:

* Prepare:
  + All alien Methods will be gathered in 1 interface Visitor that mainly include:
    - visitObjectA(ObjectA): use ObjectA to do some actions.
    - visitObjectB(ObjectB): use ObjectB to do some actions.
    - …
  + Objects will add 1 more method to call Visitor’s method:
    - accept(Visitor): call to exact Visitor’s method.
* Usage:
  + Implement all possible method in concreted Visitor.
  + When wanna use that “alien” method -> call Object.accept(Concreted\_Visitor),.

c. Achieve:

* Isolation “alien” method out of Class business logic (Single Responsibility Principle)
* Make easier when introduce new “alien” method/business to Old/Fragile Class (Open/Closed Principle).
* There is a potential case that Visitor can collect useful data during traverse through Collection.

4.3.3 Chain of Responsibility pattern

a. Problem:

* Preform a Task/Request through a chain of Action/Handlers.
* Task/Request can be done only by 1 or all of Action/Handler or be ignored.
* New Action/Handlers can be added more in future business.

c. Solution:

* Prepare: Create a Handler interface, Handler will be concreted by specific business logic that basically include:
  + setNext(Handler\*): set next Handler’s address in chain.
  + Handle(Request):
    - Business logic in this concretion.
    - (Optional) Call next Handler.
* Usage:
  + Initialize all Handler instances.
  + Connect all Handlers in chain with business logic order.
  + When Request coming, Call first Handler (by its logic, it will decide to call next Handler or NOT)

c. Achieve:

* Dynamic to prepare/control the chain of handlers for Request.
* Decoupling Actions are needed for Request (Single Responsibility Principle)
* Enable to expand new Actions in future (Open/Closed Principle)
* Allow to reuse defined Handles to create new chain for new Request.

4.3.4 Command pattern

***Command****is behavioral design pattern that converts requests or simple operations into objects.*

a. Problem:

* App wants to call some services from Receiver, but should NOT directly. Then we need Sender class is created to handle all kinds of requests.
* Number kinds of request maybe huge and extended in future, then how to organize request in Sender?
* If u keep it as functions in Sender class, that totally fine in functional, which is ok with number of request is small (ex: 1,2,..,5), BUT do u think it’s still good if keep all functions with different logic, maybe a ton of line-code in same class, same file.

c. Solution:

* Prepare: Create a Functor, here is Command interface, Command will be concreted by specific business logic that basically include:
  + setABC/Constructor (parameters,….): set necessary parameters for calling Receiver’s service.
  + execute():
    - Call to Receiver and do Business logic in this concretion.
* Usage:
  + Sender holds a Command interface like a variable.
  + App constructs Command object and passes to Sender via a specific Sender’s method.
  + Depend on logic, Sender can immediately perform command or schedule.
* Concern:
  + Why don’t let App calls Commands directly no need Sender??
    - It’s OOP bro, Commands should be managed by a separate Oriented-Object -> Sender.
    - And maybe command need to be scheduled, then Sender can hold that business logic.
  + Why don’t we use std::function/function pointer???
    - If there is only 1 type of request that would be nice.
    - If requests are complex -> different data types or multiple arguments
    - Then a Functor (here is Command interface) will be a proper approach.

c. Achieve:

* Encapsulate a request into an object.
* Easy to add new type of request (Open/Closed principle)

## **Some question:**

#### **The Proper place to use the volatile keyword?**

A volatile is an important qualifier in C programming. Here I am pointing some places where we need to use the volatile keyword.

* Accessing the memory-mapped peripherals register or hardware status register.

**#define COM\_STATUS\_BIT 0x00000006**

**uint32\_t const volatile \*** **const pStatusReg = (uint32\_t\*)*0x00020000*;**

**unit32\_t GetRecvData()**

**{**

**unit32\_t RecvData;**

**//Code to receive data**

**while** **(((\*pStatusReg)** **&** **COM\_STATUS\_BIT)** **== 0)**

**{**

**// Wait until flag does not set**

**//Received data in RecvData**

**}**

**return** **RecvData;**

**}**

* Sharing the global variables or buffers between the multiple threads.
* Accessing the global variables in an interrupt routine or signal handler.

**volatile int giFlag = 0;**

**ISR(void)**

**{**

**giFlag = 1;**

**}**

**int main(void)**

**{**

**while** **(!giFlag)**

**{**

**//do some work**

**}**

**return** **0;**

**}**