Contents

[**I.** **Programming skill** 3](#_Toc28506630)

[**1.** **Multithread** 3](#_Toc28506631)

[**2.** **Data alignment** 3](#_Toc28506632)

[3. **Memory Layout** (memory of a process) 3](#_Toc28506633)

[**4.** **Mutex vs Semaphore** 3](#_Toc28506634)

[**5.** **Dynamic Programming** 3](#_Toc28506635)

[**6.** **Knapsack** 3](#_Toc28506636)

[**7.** **Ranking problem** 3](#_Toc28506637)

[**8.** **Recursion** 3](#_Toc28506638)

[**9.** **Linkage** 3](#_Toc28506639)

[**II.** **Embedded Knowledge** 4](#_Toc28506640)

[**1.** **Embedded system architecture** 4](#_Toc28506641)

[**2.** **Hardware component** 4](#_Toc28506642)

[**3.** **Register:** 4](#_Toc28506643)

[**4.** **Virtual memory** 4](#_Toc28506644)

[**5.** **Unix OS architecture:** 5](#_Toc28506645)

[**6.** **GNU** 5](#_Toc28506646)

[**7.** **Linux kernel:** 6](#_Toc28506647)

[**8.** **Inter-Process communicate (IPC)** 7](#_Toc28506648)

[**9.** **Device Tree** 8](#_Toc28506649)

[**10.** **Bootloader:** 8](#_Toc28506650)

[**11.** **Yocto project** 9](#_Toc28506651)

[**12.** **Middleware** 9](#_Toc28506652)

[**13.** **Driver** 9](#_Toc28506653)

[**14.** **Makefile and make** 9](#_Toc28506654)

[**15.** **MCU peripherals:** 9](#_Toc28506655)

[**16.** **Interrupt** 12](#_Toc28506656)

[**17.** **Firmware** 12](#_Toc28506657)

[**18.** **SoC vs MCU** 12](#_Toc28506658)

[**19.** **Virtualization** 12](#_Toc28506659)

[**20.** **Platform:** 12](#_Toc28506660)

[**21.** **Design Patern** 12](#_Toc28506661)

[**III.** **Programming Knowledge** 13](#_Toc28506662)

[**1.** **Object Oriented Programming (OOP trait)** 13](#_Toc28506663)

[**1.1.** **Inheritance**: 13](#_Toc28506664)

[**1.2.** **Polymorphism**: 13](#_Toc28506665)

[**1.3.** **Attraction**: 13](#_Toc28506666)

[**1.4.** **Encapsulation**: 13](#_Toc28506667)

[**2.** **Static:** 15](#_Toc28506668)

[**2.1.** Variable: 15](#_Toc28506669)

[**2.2.** Usage 15](#_Toc28506670)

[**3.** **String** 15](#_Toc28506671)

[**4.** **Pointer vs Reference:** 15](#_Toc28506672)

[**5.** **Array** 15](#_Toc28506673)

[**6.** **Exception** 16](#_Toc28506674)

[**7.** **Standard Template Library (STL)** 17](#_Toc28506675)

[***7.1.*** Algogithms 17](#_Toc28506676)

[***7.2.*** Containers 17](#_Toc28506677)

[7.2.1. Vector 17](#_Toc28506678)

[7.2.2. *Pair* 17](#_Toc28506679)

[7.2.3. *List* 17](#_Toc28506680)

[7.2.4. *Map* 17](#_Toc28506681)

[7.2.5. *Link List* 17](#_Toc28506682)

[**7.3.** Functions 17](#_Toc28506683)

[**7.4.** Iterator 17](#_Toc28506684)

[**8.** **Namespace** 17](#_Toc28506685)

[**9.** **Template** 17](#_Toc28506686)

Knowledge Systematic

# **Programming skill**

## **Data alignment**

## **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”

## **4. Synchronization in process, thread**

**4.0 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)

**4.1 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);*  *}* |

**4.2 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.**

**4.3 Semaphore:**

**- Refer:** <https://www.guru99.com/semaphore-in-operating-system.html>,

- Refer: 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>)

**4.4** 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*

## **4.5 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(),...

## **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.

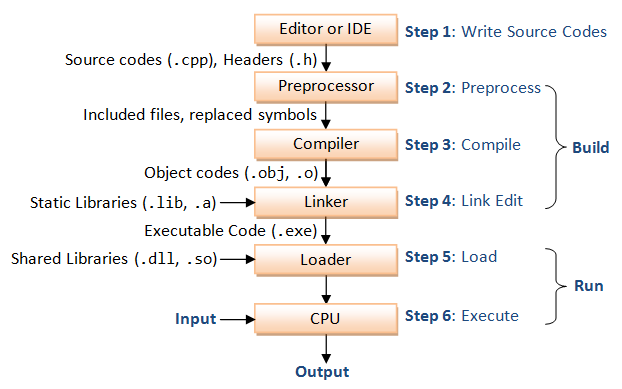
* Usage: provides a clean and simple way to write code.
* Disavantage:

+ 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?**

 [https://vi.wikipedia.org/wiki/H%E1%BB%87\_th%E1%BB%91ng\_nh%C3%BAng](https://vi.wikipedia.org/wiki/Hệ_thống_nhúng)

## Káº¿t quáº£ hÃ¬nh áº£nh cho what is middleware eetime**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.

## **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

## **GNU**

## **Linux kernel:**

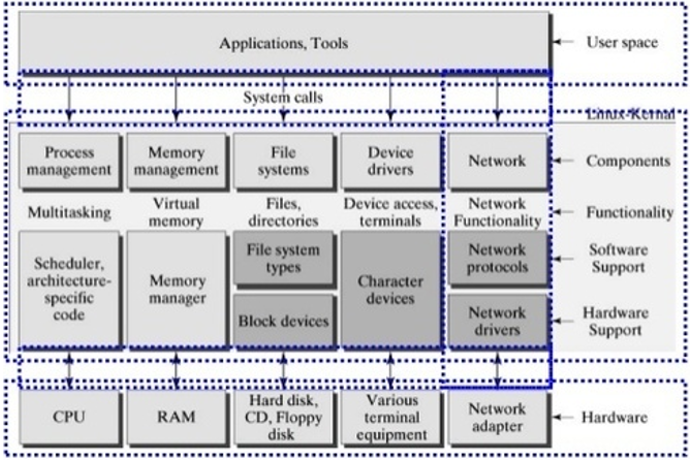
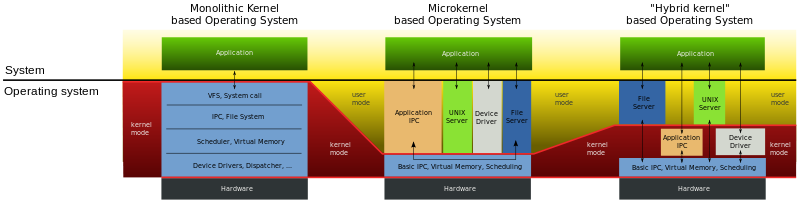
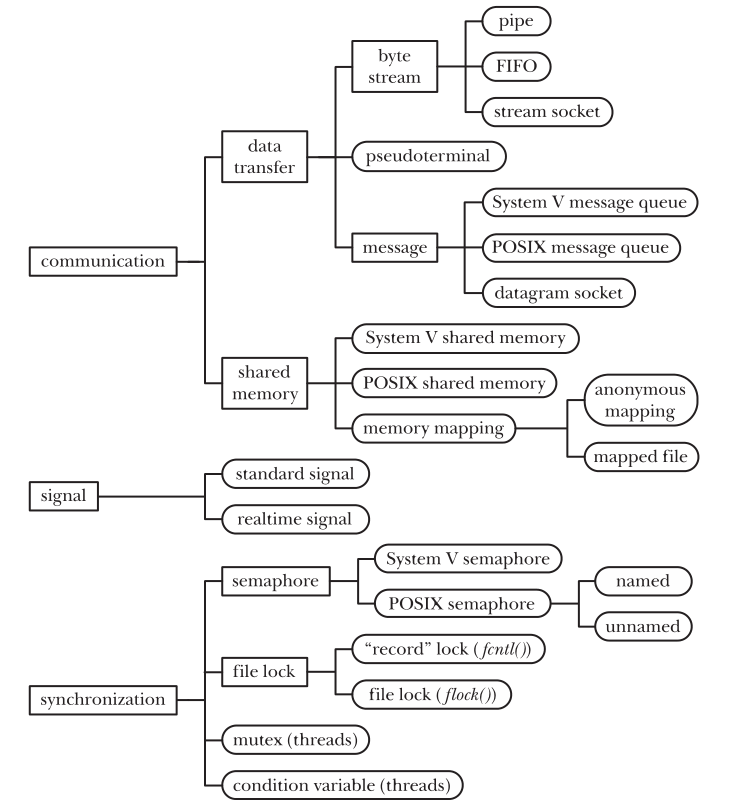
* Refer: <https://vimentor.com/vi/lesson/gioi-thieu-ve-linux-kernel-1>
* Main part of OS.
* Configured to run on a particular board (may be part of a BSP supplied by a hardware vendor).

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/#windown) 7 sử dụng Kernel Hybrid.

## **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.

## **Device Tree**

* Refer: <https://kipalog.com/posts/Device-Tree-trong-Linux>

## **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: ………………..**

## **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…)

## **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 peripherals:**

* 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. ***Serial tranfer***
* It is a technique of sending data one-by-one bit through a data line at each predetermined time.
* 2 Methods:
  + **Clock Synchronous:** have 2 line connect to another:
    - Clock (Control line): Inform tranfer timing. 1 clock = 1 bit.
    - Data line: Send data.
      1. Data is transmitted in synchronization with the *Clock signal* (Control signal).
      2. Tranfer speed = clock speed = Tần số xung (HZ).
      3. Relationship: Master (clock supplier) – Slave.
  + **Clock Asynchronous:** Only 1 line(Don’t need *Signal line)*
    - Data line: Sending and receiving sides are matched in data format and transfer rate.
      1. Clock generator is inside each device.
      2. Relationship: equal
* **Note:** 1 line can include serveral wire-signal.
  + 1. ***UART***
* Universial Asynchronous Receiver Transmitter.
* Used to communicate to modules: Zigbee, Bluetooth, Wifi…
* Some Concept:
  + Baud rate: Số bit truyền được trong 1s (tốc độ truyền dữ liệu đo bằng đơn vị Baud).
  + …
* Refer: <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.
    - 1. 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.
    - 1. Each slave device connect to the BUS has a unique address (Master device will use it to indentify the slave devices)
  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*

## **Interrupt**

* 2 types:
  + Hardware interrupt (acsynchronous)
  + Software interrupt (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.

## **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:**

* 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

## **Design Patern**

# **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 bào 1 Object child thì

=> constructor của parent cũng được gọi để khởi tạo các member cho chile 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).**  }; |

### **Tham chiếu hằng**

* 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

nay` ma` 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 dữ liệu th{nh viên v{ nó được gọi là tham chiếu hằng.

1. **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) .

**b. Hàm:**

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

### **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[50];  char Subject[50];  int Price;  }; |

### **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 enumeration 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 in &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.

## **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(); |

### **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  } |