# the L1 and L2 CPU cache (in modern Intel CPUs, L1 and L2 caches use virtual addresses, which are re-used by multiple programs, whereas the L3 cache stores hardware addresses). Flushing the cache is expensive: if the cache is dirty (i.e. has un-committed changes) it all has to be flushed back to memory, and anyway when the cache gets flushed, you lose the benefit of having that stuff cached

# [**http://fgiasson.com/articles/memorylayout.txt**](http://fgiasson.com/articles/memorylayout.txt)

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.  
  
Program Counter  
Scheduling information  
Base and limit register value  
Currently used register  
Changed State  
I/O State information  
Accounting information

# **Relocatables contains sections where as executables contain segments**

SEGFAULT

NULL → Physical memory → PAGE\_FAULT → VALID ? → Yes → On demand Page Replacing |--> No → SEG\_FAULT → CORE File

Core file generation is Platform ( OS + Architecture ) Dependent

Core file is also ELF file , we can use objdump to check file segments

# **We have to use unions always with structures**

Structure{

Union XYZ;

Int Type\_Of\_union; → This is useful to check which variables are used in union otherwise we cannot differentiate members in Union

}

[Asymptotic Analysis](http://en.wikipedia.org/wiki/Asymptotic_analysis) is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input siz

# **Setting scheduling policy**

Sched\_setschedular

Sched\_getschedular

sched\_set/get affinity → to change/get affinity

Nice valuerange -19 to 20

Sched\_param structure to change thread properties

pthread\_attr\_setschedparam

# **big-endian and little-endian**

Big-endian and little-endian are terms that describe the order in which a sequence of [byte](https://searchstorage.techtarget.com/definition/byte)s are stored in computer memory. Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address). Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first. For example, in a big-endian computer, the two bytes required for the [hexadecimal](https://whatis.techtarget.com/definition/hexadecimal) number 4F52 would be stored as 4F52 in storage (if 4F is stored at storage address 1000, for example, 52 will be at address 1001). In a little-endian system, it would be stored as 524F (52 at address 1000, 4F at 1001).

IBM,Motorola,TCP/IP → big-endian

INTEL Processors → little-endian

When extern is used with a variable, it’s only declared not defined

Extern int i → declaration

Int i -> in another file definition

We can't have generic pointers because it is difficult to dereferencing values as we don't know how many bytes we need to consider

**Recursion in embedded systems**

Embedded systems usually have a limited amount of RAM, while recursion may need a very large 'stack', depending on whether or not tail-recursion is used, and the depth of recursion. Also, without any form of memory protection the stack could grow and overwrite other data (s.a. the program heap) there-by causing corruption and unexpected behaviour.

→ As the stack increases downwards towards heap it may corrupt it

→ stack causes execution slower

→ we need to analyze depth of recursion then only we need to use recursion in embedded systems with some tools

**Smart pointer**

For dynamic memory allocation destructor will not be called when the object goes out of scope,whereas for automatic objects ( stack objects )destructor get called when object goes out of scope , programmer explicitly needs to free memory otherwise memory will leak,and if any exceptions occurs deallocation code may not executed because it will be in later part of code it leads memory leak

Smart pointers works on principle that if object goes out of scope free memory and gives safe and guaranteed behaviour and no need to free memory explicitly

**Unique\_ptr**

→ single owner ,cannot be assigned or copied,fast compared to shared\_ptr

→ deletes when the smart pointer is destructed

→ when we returns unique pointer from function ownership changes but at any given point of time response hold by one smart pointer

std::unique\_ptr<Foo> a(new Foo());

**Shared\_ptr**

→ multiple ownership and transfer of ownership

shared\_ptr<Foo> sp = Foo (new Foo);

shared\_ptr<Foo> sp = make\_shared<Foo>();

Shared pointer maintains reference count for each shared object when the reference count becomes 0 then the memory gets freed

**Threads share data segment ,global,static variables**

**Threads has their own stack pointer,registers,scheduling properties,set of pending and blocked signals,thread specific data**

**By default message queue provides synchronization where as in shared memory programer needs to provide synchronization**

**Threads are lightweight because**

**→ creation ,destruction ,data sharing ,context switching is easy**

**Process vs Thread Switching**

Process switching is context switching from one process to a different process. It involves switching out all of the process abstractions and resources in favor of those belonging to a new process. Most notably and expensively, this means switching the memory address space. This includes memory addresses, mappings, page tables, and kernel resources—a relatively expensive operation. On some architectures, it even means flushing various processor caches that aren't shareable across address spaces. For example, x86 has to flush the TLB and some ARM processors have to flush the entirety of the L1 cache!

Thread switching is context switching from one thread to another *in the same process* (switching from thread to thread across processes is just process switching). Thread switching is much, much cheaper, as it involves switching out only the abstraction unique to threads: The processor state. Switching processor state (such as the program counter and register contents) is generally very efficient. For the most part, the cost of thread-to-thread switching is about the same as the cost of entering and exiting the kernel. On systems such as Linux, that is very cheap.

whenever a context-switch happens and a different process is executing, the code/data segments of the earlier process can even be swapped out of physical memory. Hence even maintaining a function pointer and passing it to the new process is useless as it cannot guarantee that the function code will be held in memory even after newer process is loaded in memory and starts executing.

**Schedulers in OS**

**→ long term schedulers :**  It bring the new process to the ‘Ready State’. It controls *Degree of Multi-programming*, i.e., number of process present in ready state at any point of time.

**→ short term :**  It is responsible for selecting one process from ready state for scheduling it on the running state. Note: Short term scheduler only selects the process to schedule it doesn’t load the process on running.

***Dispatcher*** is responsible for loading the selected process by Short Term scheduler on the CPU (Ready to Running State) Context switching is done by dispatcher only. A dispatcher does following:

1) Switching context.

2) Switching to user mode.

3) Jumping to the proper location in the newly loaded program.

**→ medium term :** It is responsible for suspending and resuming the process. It mainly does swapping (moving processes from main memory to disk and vice versa).

**Benefits(use) of pointers in c:**

* Pointers provide direct access to memory
* Pointers provide a way to return more than one value to the functions
* Reduces the storage space and complexity of the program
* Reduces the execution time of the program
* Provides an alternate way to access array elements
* Pointers can be used to pass information back and forth between the calling function and called function.
* Pointers allows us to perform dynamic memory allocation and deallocation.
* Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.
* Pointers allows us to resize the dynamically allocated memory block.
* Addresses of objects can be extracted using pointers

**Drawbacks of pointers in c:**

* Uninitialized pointers might cause segmentation fault.
* Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
* Pointers are slower than normal variables.
* If pointers are updated with incorrect values, it might lead to memory corruption.

**Quick Sort is preferred for arrays and merge Sort for linked lists**

**Quick sort is in place Algorithm (no extra memory needed) Merger is out of space (extra memory required for sorting)**

**Quick sort is better if we take Better Pivot**

[**Merge sort**](http://en.wikipedia.org/wiki/Merge_sort) **is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.**

If the operation or algorithm takes constant time to perform irrespective of size of input then the time complexity for that algorithm is O(1)

A C++ friend functions are special functions which can access the private members of a class. They are considered to be a loophole in the Object Oriented Programming concepts, but logical use of them can make them useful in certain cases.

For instance: when it is not possible to implement some function, without making private members accessible in them. This situation arises mostly in case of operator overloading.

Or another example when you want to compare two private data members of two different classes in that case you need a common function which can make use of both the private variables of different class. In that case you create a normal function and make friend in both the classes, as to provide access of theirs private variables.

At work we use friends for testing code, extensively. It means we can provide proper encapsulation and information hiding for the main application code. But also we can have separate test code that uses friends to inspect internal state and data for testing.

**New operator overloading**

Void \*operator new(size\_t s,int i);

Sample \*s = new(10)sample; ,here we are passing size as argument and data type we are passing differently

**In one class the delete operator can be overloaded only once**

**By default conversion operators have name and return type same as object type to which we convert**

**Strings storage**

char \*str;

str = "GfG"; /\* Stored in read only part of data segment \*/

\*(str+1) = 'n'; /\* Problem: trying to modify read only memory \*/

// here “GfG” will store in read only data segment while str will store in stack

Trying to modify it will give error

\*(str+1) = 'n';

char str[] = "GfG"; /\* Stored in stack segment like other auto variables \*/

\*(str+1) = 'n'; /\* No problem: String is now GnG \*/

**Double pointer in C**

→ in Linked list while Changing the head in function

→ while passing pointer by address to function and allocate memory use that memory back in calling function

Double pointers are mostly used in the out parameters of a function. Where the required object (structure) is allocated from the heap within in the called function and the caleee needs to deal with the object being created by called function

In simple words, **Use \*\* when you want to preserve (OR retain change in) the Memory-Allocation or Assignment even outside of a function call.** (So, Pass such function with double pointer arg.)

void allocate(int\*\* p)  
{  
 \*p = (int\*)malloc(sizeof(int));  
}  
  
int main()  
{  
 int\* p = NULL;  
 allocate(&p);  
 \*p = 42;  
 free(p);  
}

Most of the times passing big structures by value will need to copy whole structure to stack thus using more memory and reduced performance so single pointers will be like used

**Casting in CPP**

**Const\_cast :**

→ to remove the constantness of a pointer or reference we need to use this

Int a = 10;

const int \*p = &a;

Int \*p1 = const\_cast<int\*>p

→ if we assign a const pointer to non constant pointer we will get error ,but with const\_cast we can overcome this

→ if we want to pass const pointer to any third party library that will take only non const pointer we have to remove constantness with const\_cast

/\* use with ony non constant variables \*/

**Static\_cast**

**→** It will only perform the cast if the type types are related. If the types are not related, you will get a compiler error

→ type casting compatibility check

Char c;int \*a=(char\*)&c ; → valid in c and program will compile but get run time issues

Int \*a = static\_cast<int\*>(&a ) → gives error at compile time

→ implicit type conversion

For casting to and from void\*, static\_cast should be preferred.

int\* a = new int();  
void\* b = static\_cast<void\*>(a);  
int\* c = static\_cast<int\*>(b);

**A,b,c pointing to same location**

**Dynamic cast**

The **dynamic\_cast** operator is intended to be the most heavily used RTTI component. It doesn't give us what type of object a pointer points to. Instead, it answers the question of whether we can **safely assign** the address of an object to a pointer of a particular type.

This is used to call derived class functions from base class pointer

The need for **dynamic\_cast** generally arises because you want to perform derived class operation on a derived class object, but you have only a pointer or reference-to-base

Downcast will lead unexpected results as base class dont have some derived class functions

For dynamic cast we need to have atleast one virtual function in base class

**reinterpret\_cast** is a type of casting operator used in C++.

* It is used to convert one pointer of another pointer of any type, no matter either the class is related to each other or not.
* It does not check if the pointer type and data pointed by the pointer is same or not.

int\* a = new int();  
void\* b = reinterpret\_cast<void\*>(a);  
int\* c = reinterpret\_cast<int\*>(b);

Here a,c point to same location but for b it is undefined in standard

Reinterpret\_cast is usefull when programmer dealing with vendor API’s that accept user defined data type ,in this case programmer need to cast to uderdefined datatype other wise we will getcompilation error(here ststic\_cast wont work)

You cannot call a non-virtual member function of the derived class with a pointer to the base class.

You'll need a pointer to the derived class. The simplest method is to use dynamic\_cast to get a pointer to the derived class, check whether the cast was successful, then call the derived class member function using a derived class pointer.

A better method would be to provide a virtual member function in the base class and implement it in the derived class.

**Fork**

If we fork, file descriptors are also shared (offset is common ) for both process so the read write operations affect the file content only on close

**What is the need of mutable?**

Sometimes there is requirement to modify one or more data members of class / struct through const function even though you don’t want the function to update other members of class / struct. This task can be easily performed by using mutable keyword

**Initializer list**

Constants,reference,in class hierarchy calling base parameterised constructor

**Storage Classes**

* The storage class determines the part of memory where storage is allocated for an object (particularly variables and functions) and how long the storage allocation continues to exist.
* A scope specifies the part of the program which a variable name is visible, that is the accessibility of the variable by its name. In C program, there are four storage classes: automatic, register, external, and static

**REGISTER VARIABLE - register**

* Automatic variables are allocated storage in the main memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing directly in the CPU.
* Registers are memory located within the CPU itself where data can be stored and accessed quickly. Normally, the **compiler determines** what data is to be stored in the registers of the CPU at what times.
* However, the C language provides the storage class register so that the programmer can suggest to the compiler that particular automatic variables should be allocated to CPU registers, if possible and it is not an obligation for the CPU to do this.

# **EXTERNAL VARIABLE - extern**

* All variables we have seen so far have had limited scope (the block in which they are declared) and limited lifetimes (as for automatic variables).
* However, in some applications it may be useful to have data which is accessible from within any block and/or which remains in existence for the entire execution of the program. Such variables are called **global variables**, and the C language provides storage classes which can meet these requirements; namely, the **external** (extern) and **static** (static) classes.

STATIC

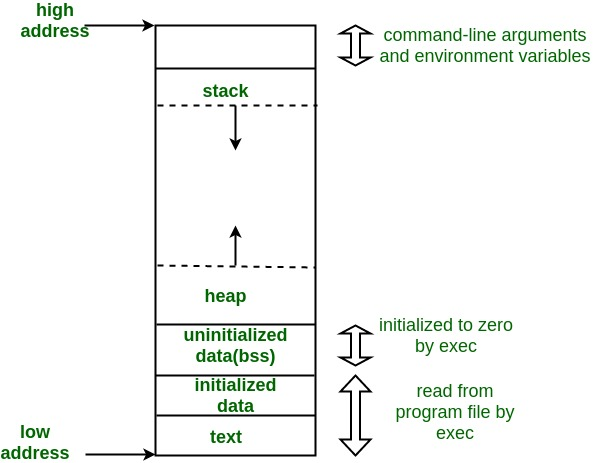
The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

AUTO:This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Storage place** | **Scope** | **Life** | **Default Value** |
| auto | CPU Memory | body | Within the Function | Garbage value |
| static | CPU Memory | function | program | 0 (zero) |
| extern | CPU Memory | program | Till the end of the main program. | 0 (zero) |
| register | Register memory | body | Within the Function | Garbage value |



/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Memory Layout of a c Program\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1. Text segment

2. Initialized data segment

3. Uninitialized data segment

4. Stack

5. Heap

## **Text or Code Segment**

*Code segment*, also known as *text segment* contains machine code of the compiled program. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified.

## **Data Segments**

*Data segment* stores program data. This data could be in form of initialized or uninitialized variables, and it could be local or global. Data segment is further divided into four sub-data segments (initialized data segment, uninitialized or .bss data segment, stack, and heap) to store variables depending upon if they are local or global, and initialized or uninitialized.

### **Initialized Data or Data Segment**

*Initialized data* or simply *data segment* stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand.

### **Uninitialized Data or .bss ( Block Started by Symbol)Segment**

Contrary to initialized data segment, *uninitialized data* or *.bss segment* stores all uninitialized global, static, and external variables (declared with extern keyword). Global, external, and static variable are by default initialized to zero. This section occupies no actual space in the object file; it is merely a place holder. Object file formats distinguish between initialized and uninitialized variables for space efficiency; uninitialized variables do not have to occupy any actual disk space in the object file

The .bss segment is an optimization. The entire .bss segment is described by a single number, probably 4 bytes or 8 bytes, that gives its size in the running process, whereas the .data section is as big as the sum of sizes of the initialized variables.

**bss’** is for the uninitialized data in RAM which is initialized with zero in the startup code.

Typically only the length of the bss section, but no data, is stored in the [object file](https://en.wikipedia.org/wiki/Object_file). The [program loader](https://en.wikipedia.org/wiki/Program_loader) allocates and initializes memory for the bss section when it loads the program. Operating systems may use a technique called [zero-fill-on-demand](https://en.wikipedia.org/w/index.php?title=Zero-fill-on-demand&action=edit&redlink=1) to efficiently implement the bss segment ([McKusick & Karels 1986](https://en.wikipedia.org/wiki/.bss#CITEREFMcKusickKarels1986)). In embedded software, the bss segment is mapped into memory that is initialized to zero by the C [run-time system](https://en.wikipedia.org/wiki/Run-time_system) before [main()](https://en.wikipedia.org/wiki/Main_function_(programming)#C_and_C.2B.2B) is entered.

The size that BSS will require at runtime is recorded in the object file, but BSS (unlike the data segment) doesn't take up any actual space in the object file."

### **Stack Segment**

*Stack segment* is used to store all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. Local variables have a scope to the block which they are defined in; they are created when control enters into the block. Local variables do not appear in *data* or *bss* segment. Also all recursive function calls are added to stack. Data is added or removed in a last-in-first-out manner to stack. When a new stack frame needs to be added (as a result of a newly called function), the stack grows downward

### **Heap Segment**

*Heap segment* is also part of RAM where dynamically allocated variables are stored. In C language dynamic memory allocation is done by using malloc and calloc functions. When some more memory need to be allocated using malloc and calloc function, heap grows upward as shown in above diagram.

---> Based on the C implementation, the code segment can also contain read-only string literals. For example, when you do printf("Hello, world") then string "Hello, world" gets created in the code/text segment. You can verify this using size command in Linux OS

**What are the necessary conditions for deadlock?**

***Mutual Exclusion****:* There is s resource that cannot be shared.

***Hold and Wait****:* A process is holding at least one resource and waiting for another resource which is with some other process.

***No Preemptio****n:* The operating system is not allowed to take a resource back from a process until process gives it back.

***Circular Wait****:* A set of processes are waiting for each other in circular form.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***How to Reduce the size of a Executable**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

The following are different ways of reducing size of executable:-

* Use shared/dynamic libraries instead of static libraries while linking an executable.
* Reduce the usage of data segment by using heap at runtime using malloc().
* Eliminate redundant code (dead code) by compiling the program at higher level of optimization.

Debugging techniques

→ logging

→ assertions

→ gdb

→ valgrind

**Valgrind**

**Compile your program with -g to include debugging information so that Memcheck's error messages include exact line numbers**

**valgrind --leak-check=yes myprog arg1 arg2**

**FileName : Memcheck.c**

**#include <stdio.h>**

**int main()**

**{**

**int \*x = (int\*)malloc(10);**

**x[10] = 4;**

**//free(x);**

**return 0;**

**}**

**Gdb**

**List → l → list around 10 lines of current position**

**List followed by function name gives source code of function**

**Break point -> to stop executon at particular linu number or function , : b funcion\_name or line number ,info breakpoints gives the list of breakpoints and their id’s**

**Delete break point od of break point → delete the break point**

**Step ->> step into a function code**

**Continue → c → continue from prestnt instruction till break point or till end of program**

**Attaching running program**

**Gdb attach pid or process name**

**Back trace : list all the function nemes and their id till we encountered with the id’s we can go back and print some variable information for debugging**

**Gdb with threads**

**Info threads → list all the thread id’s**

**To enable threads debugging with gdb we need to run this command at start of gdb session**

**set non-stop on**

**set target-async on**

**set pagination off**

**Though mutex & semaphores are used as synchronization primitives ,there is a big difference between them. In the case of mutex, only the thread that locked or acquired the mutex can unlock it. In the case of a semaphore, a thread waiting on a semaphore can be signaled by a different thread. Some operating system supports using mutex & semaphores between process. Typically usage is creating in shared memory.**

**gcc -g Memleak.c**

-g tells the compiler to store symbol table information in the executable. Among other things, this includes: symbol names,type info for symbols,files and line numbers where the symbols came from

==25097== Memcheck, a memory error detector

==25097== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.

==25097== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==25097== Command: ./a.out

==25097==

==25097== Invalid write of size 4

==25097== at 0x400544: main (Memleak.c:7)

==25097== Address 0x5203068 is 24 bytes after a block of size 16 in arena "client"

==25097==

==25097==

==25097== HEAP SUMMARY:

==25097== in use at exit: 10 bytes in 1 blocks

==25097== total heap usage: 1 allocs, 0 frees, 10 bytes allocated

==25097==

==25097== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1

==25097== at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)

==25097== by 0x400537: main (Memleak.c:6)

==25097==

==25097== LEAK SUMMARY:

==25097== definitely lost: 10 bytes in 1 blocks

==25097== indirectly lost: 0 bytes in 0 blocks

==25097== possibly lost: 0 bytes in 0 blocks

==25097== still reachable: 0 bytes in 0 blocks

==25097== suppressed: 0 bytes in 0 blocks

==25097==

==25097== For counts of detected and suppressed errors, rerun with: -v

==25097== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

Here25097 is PID of Process,Heap Summary Tells how many Leaks are there (allocations , deallocations)

"definitely lost": your program is leaking memory -- fix it!

"probably lost": your program is leaking memory, unless you're doing funny things with pointers (such as moving them to point to the middle of a heap block).

**Volatile**

volatile in C actually came into existence for the purpose of not caching the values of the variable automatically. It will tell the machine not to cache the value of this variable. So it will take the value of the given volatile variable from the main memory every time it encounters it. This mechanism is used because at any time the value can be modified by the OS or any interrupt. So using volatile will help us accessing the value afresh every time.

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application

**Reference Variable**

A **reference variable** is an alias, that is, another name for an already existing **variable**. Once a **reference** is initialized with a **variable**, either the **variable** name or the **reference** name may be used to refer to the **variable**.

* Use references in function parameters and return types to define useful and self-documenting interfaces.with reference we can directly use variables ( -> operator is not needed )
* Use pointers to implement algorithms and data structures. In linked list where we canot use initialize refernce to NULL and cant use reference to point to another address ( for links )

**#Pragma**

In general you **should not** use #pragma pack. Yes, it will make your structures smaller in memory since it eliminates all padding between struct members. But it can make ***accessing*** those members much more expensive since the members may no longer fall along their required alignment. For example, in ARM architectures, 4-byte ints are typically required to be 4-byte aligned, but in a packed struct they might not be. That means the compiler needs to add extra instructions to safely access that struct member, or the developer has to access it byte-by-byte and reconstruct the int manually. Either way it results in more code than an aligned access, so your struct ends up smaller but your accessing code potentially ends up slower and larger.

You **should** use #pragma pack when your structure must match an **exact** data layout. This typically happens when you are writing code to match a data transport or access specification... e.g., network protocols, storage protocols, device drivers that access HW registers. In those cases you may need #pragma pack to force your structures to match the spec-defined data layout. This will possibly incur the same performance penalty mentioned in the previous paragraph, but may be the only way to comply with the specification.

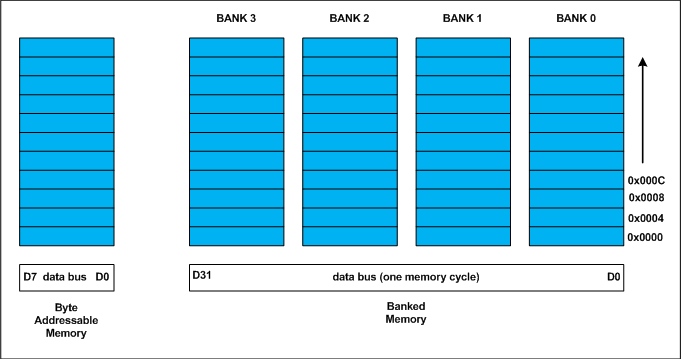
**Data Alignment:**

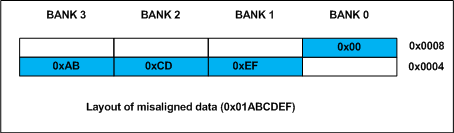
Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.

Historically memory is byte addressable and arranged sequentially. If the memory is arranged as single bank of one byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of integer in one memory cycle. To take such advantage, the memory will be arranged as group of 4 banks as shown in the above figure.

The memory addressing still be sequential. If bank 0 occupies an address X, bank 1, bank 2 and bank 3 will be at (X + 1), (X + 2) and (X + 3) addresses. If an integer of 4 bytes is allocated on X address (X is multiple of 4), the processor needs only one memory cycle to read entire integer.

Where as, if the integer is allocated at an address other than multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycle to fetch the data.





**When is copy constructor called?**

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.

2. When an object of the class is passed (to a function) by value as an argument.

3. When an object is constructed based on another object of the same class.

4. When compiler generates a temporary object.

It is however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example being the [return value optimization (sometimes referred to as RVO)](http://en.wikipedia.org/wiki/Return_value_optimization).

Source:<http://www.geeksforgeeks.org/g-fact-13/>

**When is user defined copy constructor needed?**

If we don’t define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any run time allocation of resource like file handle, a network connection..etc.

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. Assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls copy constrictor and (2) calls assignment operator. See [this](http://www.geeksforgeeks.org/copy-constructor-vs-assignment-operator-in-c/) for more details.

**Constant Volatile**

In an embedded system, this is typically used to access hardware registers that can be read and are updated by the hardware,

An object marked as const volatile will not be permitted to be changed by the code (an error wil be raised due to the const qualifier) - at least through that particular name/pointer.

The volatile part of the qualifier means that the compiler cannot optimize or reorder access to the object.

but the value can be modified from the outside

**How is a semaphore decremented? What is a named pipe? How does it differ from a socket? How does a unix socket differ from a network socket?**

IN PLACE OUT PLACE ALGORITHMS ?

## malloc/free

* Allocates/release memory
  1. Memory allocated from 'Heap'
  2. Returns a void\*
  3. Returns NULL on failure
  4. Must specify the size required in bytes.
  5. Allocating array requires manual calculation of space.
  6. Reallocating larger chunk of memory simple (No copy constructor to worry about)
  7. They will NOT call new/delete
  8. No way to splice user code into the allocation sequence to help with low memory.
  9. malloc/free can NOT be overridden legally

**Difference between library call and system call**

**Basically there are two modes of linux kernel viz.**

1. **User mode,**
2. **Kernel mode.**

**Any linux kernel switches itself back and forth between these two modes. Generally, *Library calls* get executed in *User mode* and *System calls* get executed in *Kernel mode*. In operating system terms, Kernel mode is *Atomic* in nature and its in *Supervisory mode.* Almost all Library calls need help from the kernel to perform its tasks. Each library call in turn calls underlying system call. Let us make idea more clearer using following example**

1. **fopen() : is a Library call,**
2. **open() : is a System call.**

**Whenever in c program, you use fopen() from header file. Programming environment calls system call open() from kernel and perform its file opening task. Again after executing, control flow return to user mod**

Inter Process Communication (IPC)

NetWork Protocals (TCP/UDP) have packeting overhead and are bound to even more resources (e.g. ports, loopback interface). --->More Overhead

Shared memory is faster because the data is not copied from one address space to another, memory allocation is done only once, andsyncronisation is up to the processes sharing the memory. mostly, shared memory just give more control but if you use your shared memory like a pipe, it probably won't make much difference depending on your kernel implementation.

Why would this be one of the fastest? Minimal overhead. Overhead is generated whenever you make a call to another function. Be it a kernel or library, if your IPC makes no calls to any other function, then you've done away with a large bottleneck. Shared memory IPCs have no requirement for third party function calls.

**Macro VS Inline**

Advantage: Macros and Inline functions are efficient than calling a normal function. The times spend in calling the function is saved in case of macros and inline functions as these are included directly into the code.

Disadvantage: Macros and inline functions increased the size of executable code. - See more at: http://www.coolinterview.com/interview/4362/#sthash.yt4cmpqu.dpuf

Difference in inline functions and macro

1) Macro is expanded by preprocessor and inline function are expanded by compiler.

2) Expressions passed as arguments to inline functions are evaluated only once while \_expression passed as argument to inline functions are evaluated more than once.

More over inline functions are used to overcome the overhead of function calls. Macros are used to maintain the readbility and easy maintainence of the code.

* Inline functions follow all the protocols of type safety enforced on normal functions.
* Inline functions are specified using the same syntax as any other function except that they include the inline keyword in the function declaration.
* Expressions passed as arguments to inline functions are evaluated once.
* In some cases, expressions passed as arguments to macros can be evaluated more than once.
* macros are expanded at pre-compile time, you cannot use them for debugging, but you can use inline functions.

**Why sizeof is Operator Not a function**

sizeof operator is compile time entity not runtime and don't need parenthesis like a function. When code is compiled then it replace the value with the size of that variable at compile time but in function after function gets execute then we will know the returning value.

A function by definition is allocated space at runtime and can operate on data only.

While sizeof operates on data types as well, which are compiler specific. A function does not know how to interpret the datatype "int" and ask the compiler for its size, by design.

Before C99 sizeof was totally compile-time, but in the new standard, it can be used to get information at runtime as well, for variable-length arrays.

**Inline functions provide following advantages:**

1) Function call overhead doesn’t occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

**Pointer Vs Reference**

A pointer can be re-assigned any number of times while a reference cannot be re-seated after binding.

Pointers can point nowhere (NULL), whereas reference always refer to an object.

You can't take the address of a reference like you can with pointers.

There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).

A pointer can be re-assigned:

int x = 5;

int y = 6;

int \*p;

p = &x;

p = &y;

\*p = 10;

assert(x == 5);

assert(y == 10);

A reference cannot, and must be assigned at initialization:

int x = 5;

int y = 6;

int &r = x;

A pointer has its own memory address and size on the stack (4 bytes on x86), whereas a reference shares the same memory address (with the original variable) but also takes up some space on the stack. Since a reference has the same address as the original variable itself, it is safe to think of a reference as another name for the same variable. Note: What a pointer points to can be on the stack or heap. Ditto a reference. My claim in this statement is not that a pointer must point to the stack. A pointer is just a variable that holds a memory address. This variable is on the stack. Since a reference has its own space on the stack, and since the address is the same as the variable it references. More on stack vs heap. This implies that there is a real address of a reference that the compiler will not tell you.

int x = 0;

int &r = x;

int \*p = &x;

int \*p2 = &r;

assert(p == p2);

You can have pointers to pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.

int x = 0;

int y = 0;

int \*p = &x;

int \*q = &y;

int \*\*pp = &p;

pp = &q;//\*pp = q

\*\*pp = 4;

assert(y == 4);

assert(x == 0);

Pointer can be assigned nullptr directly, whereas reference cannot. If you try hard enough, and you know how, you can make the address of a reference nullptr. Likewise, if you try hard enough you can have a reference to a pointer, and then that reference can contain nullptr.

int \*p = nullptr;

int &r = nullptr; <--- compiling error

Pointers can iterate over an array, you can use ++ to go to the next item that a pointer is pointing to, and + 4 to go to the 5th element. This is no matter what size the object is that the pointer points to.

A pointer needs to be dereferenced with \* to access the memory location it points to, whereas a reference can be used directly. A pointer to a class/struct uses -> to access it's members whereas a reference uses a ..

A pointer is a variable that holds a memory address. Regardless of how a reference is implemented, a reference has the same memory address as the item it references.

References cannot be stuffed into an array, whereas pointers can be (Mentioned by user @litb)

Const references can be bound to temporaries. Pointers cannot (not without some indirection):

const int &x = int(12); //legal C++

int \*y = &int(12); //illegal to dereference a temporary

**When to use fopen over open**

fopen provides you with buffering IO that may turn out to be a lot faster than what you're doing with open

A FILE \* gives you the ability to use fscanf and other stdio functions

Your code may someday need to be ported to some other platform that only supports ANSI C and does not support the open function.( open is not portable and environment specific )

In places where you are mainly reading or writing a file sequentially, the buffering support is really helpful and a big speed improvement. But it can lead to some interesting problems in which data does not end up in the file when you expect it to be there. You have to remember to fclose or fflush at the appropriate times.

**Buffered IO ,Non Buffered IO**

*User buffered I/O*, shortened to *buffering* or *buffered I/O*, refers to the technique of temporarily storing the results of an I/O operation in user-space before transmitting it to the kernel (in the case of writes) or before providing it to your process (in the case of reads). By so buffering the data, you can minimize the number of system calls and can block-align I/O operations, which may improve the performance of your application.

For example, consider a process that writes one character at a time to a file. This is obviously inefficient: Each write operation corresponds to a write() system call, which means a trip into the kernel, a memory copy (of a single byte!), and a return to user-space, only to repeat the whole ordeal. Worse, filesystems and storage media work in terms of *blocks*; operations are fastest when aligned to integer multiples of those blocks. Misaligned operations, particularly very small ones, incur additional overhead.

User buffered I/O avoids this inefficiency by buffering the writes in a data buffer in user-space until a certain threshold is reached—ideally, the underlying filesystem's block size or an integer multiple thereof. To use our previous example, we'd simply copy each character into the buffer and call write() only when the block size is reached.

One example is standard error under a C runtime library - this is usually unbuffered by default. Since errors are (hopefully) infrequent, you want to know about them immediately. On the other hand, standard output *is* buffered simply because it's assumed there will be far more data going through it.

Another example is a logging library. If your log messages are held within buffers in your process, and your process dumps core, there a very good chance that output will never be written.

**Memcpy vs strcpy**

strcpy copies character from source to destination one by one until it find NULL or '\0' character in the source

where as memcpy copies data (not character) from source to destination of given size n, irrespective of data in source.

memcpy should be used if you know well that source contain other than character. for encrypted data or binary data, memcpy is ideal way to go.

**Size of Character ‘a’ in C/C++**

In C, the type of a character *constant* like 'a' is actually an int, with size of 4 (or some other implementation-dependent value). In C++, the type is char, with size of 1. This is one of many small differences between the two languages

.

Memmv vs memcp example

#include <memory.h>  
#include <string.h>  
#include <stdio.h>  
  
char str1[17] = "abcdef";  
  
int main()  
{  
  
 printf( "The string: %s\n", str1 );  
 memcpy( (str1+6), str1, 10 );  
 printf( "New string: %s\n", str1 );  
  
 strcpy\_s( str1, sizeof(str1), "aabbcc" ); // reset string  
  
 printf( "The string: %s\n", str1 );  
 memmove( (str1+6), str1, 10 );  
 printf( "New string: %s\n", str1 );  
  
}

Output The string: abcdef  
New string: abcdefabcdefabcd  
The string: abcdef  
New string: abcdefabcdef

**Constant Pointers**

A constant pointer is a pointer that cannot change the address its holding. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.

int \* const ptr;

**Pointer to Constants**

a pointer through which one cannot change the value of variable it points is known as a pointer to constant. These type of pointers can change the address they point to but cannot change the value kept at those address.

const int\* ptr;

**Constant Pointer to a constant**

A constant pointer to constant is a pointer that can neither change the address its pointing to and nor it can change the value kept at that address

const int\* const ptr;

**rvalue and lvalue**

The expression appearing on right side of the assignment operator is called as rvalue. Rvalue is assigned to lvalue, which appears on left side of the assignment operator. The lvalue should designate to a variable not a constant.

**Function Pointers**

#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

printf("Value of a is %d\n", a);

}

int main()

{

// fun\_ptr is a pointer to function fun()

void (\*fun\_ptr)(int) = &fun;

/\* The above line is equivalent of following two

void (\*fun\_ptr)(int);

fun\_ptr = &fun;

\*/

// Invoking fun() using fun\_ptr

(\*fun\_ptr)(10);

return 0;

}

If we remove bracket, then the expression “void (\*fun\_ptr)(int)” becomes “void \*fun\_ptr(int)” which is declaration of a function that returns void pointer.

**1)** Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

**2)** Unlike normal pointers, we do not allocate deallocate memory using function pointers.

**3)** A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing \*, the program still works.

void (\*fun\_ptr)(int) = fun; // & removed

fun\_ptr(10); // \* removed

**4)** Like normal pointers, we can have an array of function pointers

**5)** Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

// A simple C program to show function pointers as parameter

#include <stdio.h>

// Two simple functions

void fun1() { printf("Fun1\n"); }

void fun2() { printf("Fun2\n"); }

// A function that receives a simple function

// as parameter and calls the function

void wrapper(void (\*fun)())

{

fun();

}

int main()

{

wrapper(fun1);

wrapper(fun2);

return 0;

}

**Use cases of Function pointers**

→ in call back mechanism like in Signals ,threads

→ in qsort function

→ They can also be useful when you want to store an array of functions, to call dynamically for example.

→ Function pointers are useful for passing functions as parameters to other functions

→ **Function pointers in C can be used to perform object-oriented programming in C**

**Call back function**

In simple terms, a Callback function is one that is not called explicitly by the programmer. Instead, there is some mechanism that continually waits for events to occur, and it will call selected functions in response to particular events.

Thus, callbacks allow the user of a function to fine-tune it at runtime(Dynamically function calling).

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), a **callback** is any [executable code](https://en.wikipedia.org/wiki/Executable_code) that is passed as an [argument](https://en.wikipedia.org/wiki/Argument_(computer_science)) to other code, which is expected to *call back* (execute) the argument at a given time. This execution may be immediate as in a **synchronous callback**, or it might happen at a later time as in an **asynchronous callback**

**Function pointer and Pointer to function**

**---> Pointer to a function holds the address of a function**

**// fun\_ptr is a pointer to function fun()**

**void (\*fun\_ptr)(int) = &fun;**

**// fun\_ptr\_arr is an array of function pointers**

**void (\*fun\_ptr\_arr[])(int, int) = {add, subtract, multiply};**

**A function to a pointer is one function which returns a pointer**

**Int \*add(int,int)**

**Advantages of Static Functions**

The function's name isn't visible outside the translation unit (source file) in which it's declared, and won't conflict with another function foo in another source file.

Assuming that you're using [OOP](http://en.wikipedia.org/wiki/Object-oriented_programming), use static functions when they don't depend on any class members. They can still be private, but this way they are optimized as they don't depend on any instance of the related object.

Other than the above, I find static functions useful when you don't want to create an instance of an object just to execute one public function on it. This is mainly the case for helper classes that contain public functions to do some repetitive and general work, but don't need to maintain any state between calls.

**What does linker Do**

**→ combining multiple object files**

**→ instruction ,procedure relocation**

**→ appending bootstrap code**

**Linker is specific to Machine whereas Compiler is architecture based**

**Data Type overloading**

**For addition:-**

**INT\_MAX ---> if(a>INT\_MAX-b) integer overload happen**

**For Multiplication:-**

**Input a,b**

**Output x = a\*b → if (b!=x/a) ===> over flow detected else ===> no overflow**

**Stored Procedure advantages**

They allow faster execution.

They can reduce network traffic

They can be used as a security mechanism.

→ Pointer arithmetic is not allowed on void\* pointers. some compilers adds extension for it but it is neither standard not encouraged to use it

**Putting a 0 before an integer constant makes it an octal number and putting 0x (or 0X) makes it a hexadecimal number.**

In C, *typedef* is considered as a storage class like other storage classes (auto, register, static and extern), nevertheless the purpose of *typedef* is to assign alternative names to existing types.

**#include <stdio.h>**

**int main()**

**{**

**typedef static int points;**

**points x;**

**return 0;**

**}**

**Compiler Error: multiple storage classes in declaration specifiers**

**What are the necessary conditions for deadlock?**

***Mutual Exclusion****:* There is s resource that cannot be shared.

**avoidance**:It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tap drive and printer, are inherently non-shareable.

***Hold and Wait****:* A process is holding at least one resource and waiting for another resource which is with some other process.

**avoidance**:1. Allocate all required resources to the process before start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remained blocked till it has completed its execution.

2. Process will make new request for resources after releasing the current set of resources. This solution may lead to starvation.

***No Preemption:*** The operating system is not allowed to take a resource back from a process until process gives it back.

**avoidance**:Preempt resources from process when resources required by other high priority process.

***Circular Wait:***A set of processes are waiting for each other in circular form.

**avoidance**:Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.

For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

**Banker’s Algorithm**

Bankers’s Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it check for safe state, if after granting request system remains in the safe state it allows the request and if their is no safe state it don’t allow the request made by the process.

Inputs to Banker’s Algorithm

1. Max need of resources by each process.

2. Currently allocated resources by each process.

3. Max free available resources in the system.

Request will only be granted under below condition.

1. If request made by process is less than equal to max need to that process.

2. If request made by process is less than equal to freely availbale resource in the system.

**What is Thrashing?**

Thrashing is a situation when the performance of a computer degrades or collapses. Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults is necessary to in order to appreciate the benefits of virtual memory, thrashing has a negative affect on the system. As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault

**Zombie process:**

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

to prevent zombie state call wait() call in parent process or use signal handler with wait() call otherwise too many zombie ends up zombie queue full making failed to create new process

There is one process table per system. The size of the process table is finite. If too many zombie processes are generated, then the process table will be full. That is, the system will not be able to generate any new process, then the system will come to a standstill. Hence, we need to prevent the creation of zombie processes.

**Orphan Process:**

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

# **What happens when we turn on computer?**

Functions of BIOS

POST (power on self test) : check the hardware,initilizes hardwares (CPU,Video cards,Harddrives) if any error occurs error messages displayed on screen and number of beeps occured,

Then bios searches for Master Boot Record (MBR) typically stored on first sector of Harddrive,which stores boot loader ( GRUB in linux ) then boot loader loads operating system into memory

init is the last step of the kernel boot sequence. It looks for the file */etc/inittab* to see if there is an entry for *initdefault*. It is used to determine initial run-level of the system. A run-level is used to decide the initial state of the operating system

The next step of init is to start up various daemons that support networking and other services. X server daemon is one of the most important daemon. It manages display, keyboard, and mouse. When X server daemon is started you see a Graphical Interface and a login screen is displayed.

**Proper place to use enum**

→ for example if function takes limited set of values it is better to take enum as arguments rather that using data types(int ,float ...)

→ ex:- function needs to work on 3 values like

fun(int val)

if(val == 1)

// some work

Else if (val == 2)

// some work

Else

//some work

Can be written as

Enum day {sun,mon,tue}

Day d = sun;

fun(d) → void fun (day d)

If (d == sun)

//some work

Else if (d == mon )

// some work

Etc …………

**--------> use enum when you are dealing with limited set of values**

**How to pass multiple values from function**

struct values{

Int a;

Char c;

String s;

};

Values fun()

{

Return values(1,’c’,”abc”)

}

Values v = fun();

**2nd way**

std:tuple<int,char,std::string>fun()

{

Return make\_tuple(1,’c’,”string”);

}

Int num,char ch,std::string s;

tie(num,ch,s) = fun()