Embedded Linux C Programming Notes

Dogus YUKSEL
Second Edition
2023

Notes to readers: I just created this document by using copy and paste method from different kind of sources these can be easily and freely found by googling

- GDB (GNU Debugger)
- VALGRIND
- DMALLOC
- Compiler Steps
  - Preprocessing
  - Compilation
  - Assembly
  - Linking
- SORTINGS
  - O(N2) Sortings
    - Insertion Sort
    - Selection Sort
    - Bubble Sort
  - O(NLogN) Sortings
    - Heap Sort
    - Quick Sort
    - Merge Sort
  - Sub O(NLogN) Sortings
    - Counting Sort
    - Bucket Sort
- DATA STRUCTURES
  - Linked List
  - Stack
  - Queue
  - Tree
    - Tree Terminology
    - Binary Tree
      - Proper Binary Tree
      - Balanced Binary Tree
      - Full Binary Tree
      - Perfect Binary Tree
      - Complete Binary Tree
      - Degenerate or Pathological Tree
      - Skewer Binary Tree
- SEARCHING
    - Linear Search
      - lfind
      - lsearch
      - bsearch
  - Hash Search
    - hcreate
    - hdestroy
    - hsearch

- Tree Search
  - tsearch
  - tfind
  - tdelete
  - tdestroy
  - twalk (there is also twalk_r function for threads)
- PROCESS MEMORY LAYOUT
  - BSS Segment
  - Data Segment
  - Text Segment
  - Stack Segment
  - Heap Segment
    - Allocating Memory on the Stack: alloca()
- KERNEL TASKS
- DAEMON
- PROCESSES
  - Process Priorities (Nice Values)
  - Realtime Scheduling (Hard Realtime)
  - Posix Realtime (Soft Realtime)
  - Preventing Realtime Processes from Locing Up the System
  - CPU Affinity
- THREADS
  - Threads Versus Processes
  - THREAD SYNCHRONIZATION
    - Mutexes
    - Condition Variables
    - Semaphores
  - Thread Safety (Reentrancy)
- NETWORKING
  - OSI Layers
    - Physical Layer
    - Data Link Layer
    - Network Layer
    - Transport Layer
    - Session Layer - Presentation Layer - Application Layer
  - TCP vs UDP
  - IPV4 vs IPV6
    - IPV6 Benefits
    - IPV6 Adverses
    - IP Syntax
    - URL Syntax
    - API Differences
    - Dual Stack
    - IPV6 Address Scopes

        - IPV6 Address Pripritization (Happy Eyeball, RFC 8305)
    - Sockets
        - Socket Options
    - IO Multiplexing
- C Notes
        - file holes
        - volatile
        - To create an image with C
        - getopt_long - optional arguments
        - parameter hiding in C
        - polymorphism in C
        - function pointer example
        - popen example
        - fifo example
        - IO Streams
        - Low-Level IO
        - Signals
        - Timer
            - Periodic Timer Example
            - OneShot Timer Example
        - strtoul and strtol
        - strtok_r
        - Format Specifiers
- CLEAN CODE
    - Meaningful Names
    - Error Handling
    - Functions
    - Formatting
        - Vertical Formatting
    - Hotizontal Formatting
    - Comments
        - Good Comments
        - Bad Comments
- C CODING STANDARTS (custom)
- GNU C CODING STANDARTS
    - Formatting
    - Commenting
    - Clean Use of C Constructs
    - Naming Variables, Functions, and Files
    - Portability between CPUs
- MAKE
    - Basic Topics
        - Targets and Prerequisites
        - Explicit Rules

- Phony Targets
- Variables
  - Variable Types and Assignments
  - MACROS
  - Where Variables Come From
- VPATH
- Pattern Rules
- Managing Libraries
- Using Libraries as Prerequisites
- Conditional Preprocessing
- Built-in Functions

Advanced Topics
- Recursive Make
- Command-Line Options
- Portable Makefiles
- Parallel make
- The eval Function
- MAKE Debugging

Make Detailed Example

DESIGN PATTERNS

Basic Design Workflows

Design Patterns for Accessing Hardware
- Hardware Proxy Pattern
- Hardware Adapter Pattern
- Mediator Pattern
- Observer Pattern
- Debouncing Pattern
- Interrupt Pattern
- Polling Pattern

Design Patterns for Embedding Concurrency and Resource Management
- Scheduling Patterns
  - Cyclic Executive Pattern
  - Static Priority Pattern
- Task Coordination Patterns
  - Critical Region Pattern
  - Guarded Call Pattern
  - Queuing Pattern
  - Rendezvous Pattern
- Deadlock Avoidance Patterns
  - Simultaneous Locking Pattern
  - Ordered Locking

Design Patterns for State Machines
- Single Event Receptor Pattern
- Multiple Event Receptor Pattern

State Table Pattern
State Pattern
Safety and Reliability Patterns
One’s Complement Pattern
CRC Pattern
Smart Data Pattern
Channel Pattern
Protected Single Channel Pattern
Dual Channel Pattern
Doxygen

GDB (GNU Debugger)

- Need to add a -g option to enable built-in debugging support (which gdb needs)
- Open the program with GDB;
    - type **> gdb** and then specify the program like **> file myapp.x**
    - directly start GDB with specified program **> gdb myapp.x**
    - to pass parameter **> gdb --args myapp.x arg1 arg2 arg3**
- To learn more about a command or GDB; type **> help [command]** -> you may not use argument
- **> run** → run the program
- **> break file1.c:6** → Sets a brekpoint at line 6 in file1.c
- **> break my_function** → Break anytime my_function calles.
- **> break file1.c:6 if i >= ARRAYSIZE** → conditional breakpoint.
- **> continue** → proceed onto the next breakpoint
- **> step** → execute the next line and step-into the function
- **> next** → execute the next line and treats the next-line function as one instruction.
- **> print my_var** → print the variable my_var
- **> print/x my_var** → prints the variable in hex format
- **> print e1** → where e1 is a structure
- **> print e1->key** or **> print (*e1).key**
- **> watch my_var** → add my_var to wotch list. When its value changes, gdb prints its old and the new value.
- **> backtrace** → produces a stack trace of the function calls that lead to a seg fault
- **> where** → same as backtrace but it can be used in the middle of the program
- **> finish** → runs until the current function is finished
- **> info breakpoints** → shows information about the all declared breakpoints
- **> delete** → deletes breakpoint
- Basic usage example;
    - > gdb myapp
    - > break my_function
    - > info breakpoints
    - > run
    - > next
    - > next
    - > finish
    - > where
    - > continue

VALGRIND

> valgrind --leak-check=full --show-leak-kinds=all <app executable with args>

DMALLOC

- add **-DDMALLOC** in to the Makefile, **-DDMALLOC_FUNC_CHECK** is optional. This will allow the library to check all of the arguments of a number of common string and utility routines.
- add **-ldmallocth** at the end of library list in Makefile
- Include header
  #ifdef DMALLOC
  #include "dmalloc.h"
  #endif
- add below lines top of the main function
  #ifdef DMALLOC
  dmalloc_debug_setup("log-stats,log-non-free,log-trans,check-fence,check-funcs,log=logfile.log");
  dmalloc_log_stats();
  dmalloc_log_unfreed();
  #endif //DMALLOC
- add below lines at the end of main function;
  #ifdef DMALLOC
  dmalloc_shutdown();
  #endif //DMALLOC
- then check **logfile.log** file content

Compiler Steps

- Rule 1: We only compile source files
- Rule 2: We compile each source file separately
- Do not use preprocessors if you can write it as a function

Preprocessing

- After this step, we get a single piece of code created by copying content of the header files into the source file content.
- This preprocessed piece of code is called a translation unit. A translation unit is a single logical unit of C code generated by the preprocessor, and it is ready to be compiled. A translation unit is sometimes called a compilation unit as well.
- It is possible to ask compilers to dump the translation unit without compiling it further. In the case of gcc, it is enough to pass the -E option

Compilation

- The input to the compilation step is the translation unit, retrieved from the previous step, and the output is the corresponding assembly code
- We can always ask gcc to stop after performing the second step and dump the resulting assembly code by passing the -S
- **Abstraction Syntax Tree:** As we have explained in the previous section, a C compiler frontend should parse the translation unit and create an intermediate data structure. The compiler creates this intermediate data structure by parsing the C source code according to the C grammar and saving the result in a tree-like data structure that is not architecture-dependent. The final data structure is commonly referred to as an AST.

Assembly

- The next step after compilation is assembly. The objective here is to generate the actual machine-level instructions (or machine code) based on the assembly code generated by the compiler in the previous step. Each architecture has its own assembler, which can translate its own assembly code to its own machine code.
- A file containing the machine-level instructions that we are going to assemble in this section is called an object file.
- We have used the -c option to compile a source file and generate its corresponding object file.
- At this point, we need to remind ourselves that relocatable object files are not executable. If a project is going to have an executable file as its final product, we need to use all, or at the very least, some, of the already produced relocatable object files to build the target executable file through the linking step.

Linking

- In Unix-like systems, ld is the default linker.
- Static libraries are usually linked to other executable files, and they then become part of those executable files. They are the simplest and easiest way to encapsulate a piece of logic so that you can use it at a later point. There is an enormous number of static libraries that exist within an operating system, with each of them containing a specific piece of logic that can be used to access a certain functionality within that operating system.
- Shared object files, which have a more complicated structure rather than simply being an archive, are created directly by the linker. They are also used differently; namely, before they are used, they need to be loaded into a running process at runtime.

SORTINGS

- While using **memmove**, or **memcpy** over int array, use **sizeof(int)** while giving size to the function
- **stable sort:** A stable sorting algorithm is one that maintains the original relative positioning of equivalent values

O(N2) Sortings

Insertion Sort

- logic is, take next element and check left hand side sorted elements and insert the next element into the left while shifting 1 all the sorted elements
- pick next item and insert it proper position into the left sorted items. Sorted and unsorted items are seperated with outher for index **i**

Selection Sort

- Find the smallest item and replace it with the beginning of the array
- Incement the index and continue to do the same thing

```
unsigned int i = 0, j = 0;
int min = 0, min_idx = 0, temp = 0;

for (i = 0; i < count; i++) {
        min = numbers[i];
        min_idx = i;
        for (j = i + 1; j < count; j++) {
                if (min > numbers[j]) {
                        min = numbers[j];
                        min_idx = j;
                }
        }

        temp = numbers[i];
        numbers[i] = min;
        numbers[min_idx] = temp;
}
```

Bubble Sort

Bubblesort uses the fairly obvious fact that if an array is not sorted, then it must contain two adjacent elements that are out of order. The algorithm repeatedly passes through the array, swapping items that are out of order, until it can't find any more swaps.

O(NLogN) Sortings

Heap Sort

- Heapsort uses a data structure called a heap, which also demonstrates a useful technique for storing a complete binary tree in an array.
- One useful feature of complete binary trees is that you can easily store them in an array using a simple formula. Start by placing the root node at index 0. Then, for any node with index i , place its children at indices 2 x i + 1 and 2 x i + 2
- If a node has index j , then its parent has index ⌊(j - 1) / 2⌋, where ⌊ ⌋ means to truncate the result to the next-smallest integer. In other words, round down. For example, ⌊ 2.9 ⌋ is 2, and ⌊ 2.1 ⌋ is also 2
- Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree

Quick Sort

- The quicksort algorithm uses a divide-and-conquer strategy. It subdivides an array into two pieces and then calls itself recursively to sort the pieces
- Picking a Dividing Item → selecting fist index generally (last, middle and random indexes are another choices)
- in practice, quicksort is usually faster than heapsort
- another advantage over heapsort: it is parallelizable (divided arrays c anbe sorted on different processors)

```
#include <stdlib.h>
int cmpfunc_ascending (const void \*a, const void \*b) {
        return (\*(int\*)a - \*(int\*)b );
}

 qsort(numbers, count, sizeof(int), cmpfunc_ascending);
```

Merge Sort

- Like quicksort, mergesort uses a divide-and-conquer strategy. mergesort splits the items into two halves holding an equal number of items. It then recursively calls itself to sort the two halves. When the recursive calls to mergesort return, the algorithm merges the two sorted halves into a combined sorted list.
- Like quicksort, mergesort is parallelizable
- Mergesort is easy to implement as a stable sort → **important difference than qsort**

Sub O(NLogN) Sortings

Counting Sort

- Countingsort is a specialized algorithm that works well if the values you are sorting are integers that lie in a relatively small range. For example, if you need to sort 1 million integers with values between 0 and 1,000, countingsort can provide amazingly fast performance
- Logic is to keep the data 101 (eg) in the new array’s 101. index.

Bucket Sort

- The bucketsort algorithm, which is also called binsort, works by dividing items into buckets. It sorts the buckets either by recursively calling bucketsort or by using some other algorithm. It then concatenates the buckets' contents back into the original array in sorted order
- Unlike countingsort and pigeonhole sort, bucketsort's performance does not depend on the range of the values. Instead, it depends on the number of buckets that you use
- Bucket count should be selected wisely.

DATA STRUCTURES

- LinkedLists and Arrays are the base almost all the data structures

Linked List

- Using the first node as a reference for head and tail ptr and also count data is always better choice for simplicity **Using Sentiels**
- While custom linkedlist library design, if beginning (and ending if the linkedlist is double sided) node's address needs to be changed, then double asterix should be used (**) while passing the node to a function.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/queue.h>

struct entry {
    int data;
    TAILQ_ENTRY(entry) entries;             /* Tail queue */
};

TAILQ_HEAD(tailhead, entry);

int
main(void)
{
    struct entry *n1, *n2, *n3, *np;
    struct tailhead head;                   /* Tail queue head */
    int i;

    TAILQ_INIT(&head);                      /* Initialize the queue */

    n1 = malloc(sizeof(struct entry));      /* Insert at the head */
    TAILQ_INSERT_HEAD(&head, n1, entries);

    n1 = malloc(sizeof(struct entry));      /* Insert at the tail */
    TAILQ_INSERT_TAIL(&head, n1, entries);

    n2 = malloc(sizeof(struct entry));      /* Insert after */
    TAILQ_INSERT_AFTER(&head, n1, n2, entries);

    n3 = malloc(sizeof(struct entry));      /* Insert before */
    TAILQ_INSERT_BEFORE(n2, n3, entries);

    TAILQ_REMOVE(&head, n2, entries);       /* Deletion */
```

```
    free(n2);
                                /* Forward traversal */
    i = 0;
    TAILQ_FOREACH(np, &head, entries)
        np->data = i++;
                                /* Reverse traversal */
    TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
        printf("%i\n", np->data);
                                /* TailQ deletion */
    n1 = TAILQ_FIRST(&head);
    while (n1 != NULL) {
        n2 = TAILQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
    }
    TAILQ_INIT(&head);

    exit(EXIT_SUCCESS);
}
```

Stack

- LIFO: Last in First out

Queue

FIFO: First in First out

Tree

Tree Terminology

- **Root**: A root is a node without a parent.
- **Siblings**: Siblings mean that nodes which have the same parent node.
- **Internal Node**: Internal Node means that a node which has at least a single child.
- **External Node**: External Node means that a node which has no children. It is also known as leaf.
- **Ancestors**: Ancestors include the parent, grandparent and so on of a node.
- **Descendants**: Descendants are the opposite of ancestors, It includes the child, grandchild and so on of a node.
- **Edge**: An edge means a connection between one node to another node.
- **Path**: Path is a combination of nodes and edges connected with each other.
- **Depth**: You can calculate depth by the number of edges from node to the root of the tree.

- **Height**: Height is the maximum depth of a node.
- **Level**: Level of a node is equal to depth of the node + 1.

Binary Tree

- A binary tree is an ordered tree in which every node has at most two children.
- Every node has at most two children.
- Each child node is labeled as being either a left child or a right child.
- A left child precedes a right child in the ordering of children of a node.

Proper Binary Tree

- In a proper binary tree, every internal node has exactly wo children. A binary tree that is not proper is improper

Balanced Binary Tree

- A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right (node to leaf, depth is node to root) subtree of any node differ by not more than 1. To do that, all nodes' lef and right subtrees also should be balanced

Full Binary Tree

- every parent node / internal node has either two or no children (it means, if all leafes have siblings)

Perfect Binary Tree

- all internal nodes have two children and all leafes are at same level

Complete Binary Tree

- every level must be completely filled. all the leaf elements must lean towards the left. the last leaf element might not have a right sibling.

Degenerate or Pathological Tree

- having a single children left or right

Skewer Binary Tree

- similar to degenerate but have children only one side

SEARCHING

```
#define _GNU_SOURCE     /* Expose declaration of tdestroy() and also twalk_r()*/
#include <search.h>
```

Linear Search

- Generally searching for a specific element in an array means that potentially all elements must be checked
- the below function is commonly used for the following topics

```
static int compare_ascending(const void *x, const void *y)
{
        return (*(int *)x - *(int *)y);
}
```

lfind

- void * lfind (const void *key, const void *base, size_t *nmemb, size_t size, comparison_fn_t compar)
- The lfind function searches in the array with *nmemb elements of size bytes pointed to by base for an element which matches the one pointed to by key. The function pointed to by compar is used to decide whether two elements match.
- The return value is a pointer to the matching element in the array starting at base if it is found. If no matching element is available NULL is returned

```
result = (int *)lfind(&want_to_find, data, &array_len, sizeof(int), compare_ascending);
if (result) {
        debugf ("Key %d found in linear find\n", want_to_find);
} else {
        debugf ("Key %d not found in linear find\n", want_to_find);
}
```

lsearch

- void * lsearch (const void *key, void *base, size_t *nmemb, size_t size, comparison_fn_t compar)
- similar to lfind but the difference is that if no matching element is found the lsearch function adds the object pointed to by key (with a size of size bytes) at the end of the array and it increments the value of *nmemb to reflect this addition
- the array starting at base must have room for at least size more bytes

```
	array_len_old = array_len;

	result = (int *)lsearch(&want_to_find, data, &array_len, sizeof(int),
compare_ascending);
	if (result && array_len_old == array_len) {
		debugf ("Key %d found in linear search\n", want_to_find);
	} else if (array_len_old < array_len) {
		debugf ("Key %d not found in linear search, added\n", want_to_find);
	}
```

bsearch

- void * bsearch (const void *key, const void *array, size_t count, size_t size, comparison_fn_t compare)
- The bsearch function searches the sorted array array for an object that is equivalent to key. The array contains count elements, each of which is of size size bytes
- The return value is a pointer to the matching array element, or a null pointer if no match is found. If the array contains more than one element that matches, the one that is returned is unspecified.

```
	qsort(dup_array, len, sizeof(int), compare_ascending); //this is important

	//the array should be sorted already in ascending order (for descending order search,
compare_ascending function can be edited)
	result = (int *)bsearch(&want_to_find, dup_array, array_len, sizeof(int),
compare_ascending);
	if (result) {
		debugf ("Key %d found in binary search\n", want_to_find);
	} else {
		debugf ("Key %d not found in binary search\n", want_to_find);
	}
```

Hash Search

hcreate

- int hcreate (size_t nel)
- The hcreate function creates a hashing table which can contain at least nel elements.
- There is no possibility to grow this table so it is necessary to choose the value for nel wisely.
- Hashing tables usually work inefficiently if they are filled 80% or more.
- The weakest aspect of this function is that there can be at most one hashing table used through the whole program. The table is allocated in local memory out of control of the programmer.

- It is possible to use more than one hashing table in the program run if the former table is first destroyed by a call to hdestroy.
- The function returns a non-zero value if successful. If it returns zero, something went wrong. This could either mean there is already a hashing table in use or the program ran out of memory.

hdestroy

- void hdestroy (void)
- The hdestroy function can be used to free all the resources allocated in a previous call of hcreate. After a call to this function it is again possible to call hcreate and allocate a new table with possibly different size
- It is important to remember that the elements contained in the hashing table at the time hdestroy is called are not freed by this function. It is the responsibility of the program code to free those strings

hsearch

- ENTRY *hsearch (ENTRY item, ACTION action)
- To search in a hashing table created using hcreate the hsearch function must be used. This function can perform a simple search for an element
- it can alternatively insert the key element into the hashing table
- Entries are never replaced
- If an entry with a matching key is found the action parameter is irrelevant
- If no matching entry is found and the action parameter has the value FIND the function returns a NULL pointer
- If no entry is found and the action parameter has the value ENTER a new entry is added to the hashing table which is initialized with the parameter item. A pointer to the newly added entry is returned

---

```
unsigned int i = 0;
unsigned int total_data_count = 0;
ENTRY e, *ep;
char *data[] = { "alpha", "bravo", "charlie", "delta",
        "echo", "foxtrot", "golf", "hotel", "india", "juliet",
        "kilo", "lima", "mike", "november", "oscar", "papa",
        "quebec", "romeo", "sierra", "tango", "uniform",
        "victor", "whisky", "x-ray", "yankee", "zulu"
};

total_data_count = sizeof(data)/sizeof(data[0]);

hcreate(2 * total_data_count); //should be long enough, no change to change later

for (i = 0; i < total_data_count; i++) {
        e.key = data[i];
        /* data is just an integer, instead of a
        pointer to something */
```

```
                e.data = (void *) &i;
                ep = hsearch(e, ENTER);
                /* there should be no failures */
                if (ep == NULL) {
                        errorf( "entry failed\n");
                        goto out;
                }
        }

        e.key = data[0];
        ep = hsearch(e, FIND);
        debugf("%9.9s -> %9.9s: %d\n", e.key, ep ? ep->key : "NULL", ep ? *((int
*)(ep->data)) : 0);

        e.key = "dummy";
        ep = hsearch(e, FIND);
        debugf("%9.9s -> %9.9s: %d\n", e.key, ep ? ep->key : "NULL", ep ? *((int
*)(ep->data)) : 0);

out:
        hdestroy();
```

Tree Search

tsearch

- void * tsearch (const void *key, void **rootp, comparison_fn_t compar)
- If the tree does not contain a matching entry the key value will be added to the tree
- tsearch does not make a copy of the object pointed to by key
- The tree is represented by a pointer to a pointer since it is sometimes necessary to change the root node of the tree
- The return value is a pointer to the matching element in the tree
- If a new element was created the pointer points to the new data
- If an entry had to be created and the program ran out of space NULL is returned

tfind

- void * tfind (const void *key, void *const *rootp, comparison_fn_t compar)
- The tfind function is similar to the tsearch function
- But if no matching element is available no new element is entered. Instead the function returns NULL

tdelete

- void * tdelete (const void *key, void **rootp, comparison_fn_t compar)
- To remove a specific element matching key from the tree tdelete can be used
- a pointer to the parent of the deleted node is returned by the function

- If there is no matching entry in the tree nothing can be deleted and the function returns NULL

tdestroy

- void tdestroy (void *vroot, __free_fn_t freefct)
- If the complete search tree has to be removed one can use tdestroy. It frees all resources allocated by the tsearch functions to generate the tree pointed to by vroot.
- For the data in each tree node the function freefct is called. The pointer to the data is passed as the argument to the function. If no such work is necessary freefct must point to a function doing nothing. It is called in any case

twalk (there is also twalk_r function for threads)

- void twalk (const void *root, __action_fn_t action)
- For each node in the tree with a node pointed to by root, the twalk function calls the function provided by the parameter action.
- For leaf nodes the function is called exactly once with value set to leaf. For internal nodes the function is called three times
- Since the functions used for the action parameter to twalk must not modify the tree data, it is safe to run twalk in more than one thread at the same time, working on the same tree
- preorder, postorder, and endorder are known as preorder, inorder, and postorder: before visiting the children, after the first and before the second, and after visiting the children. Thus, the choice of name postorder is rather confusing

---

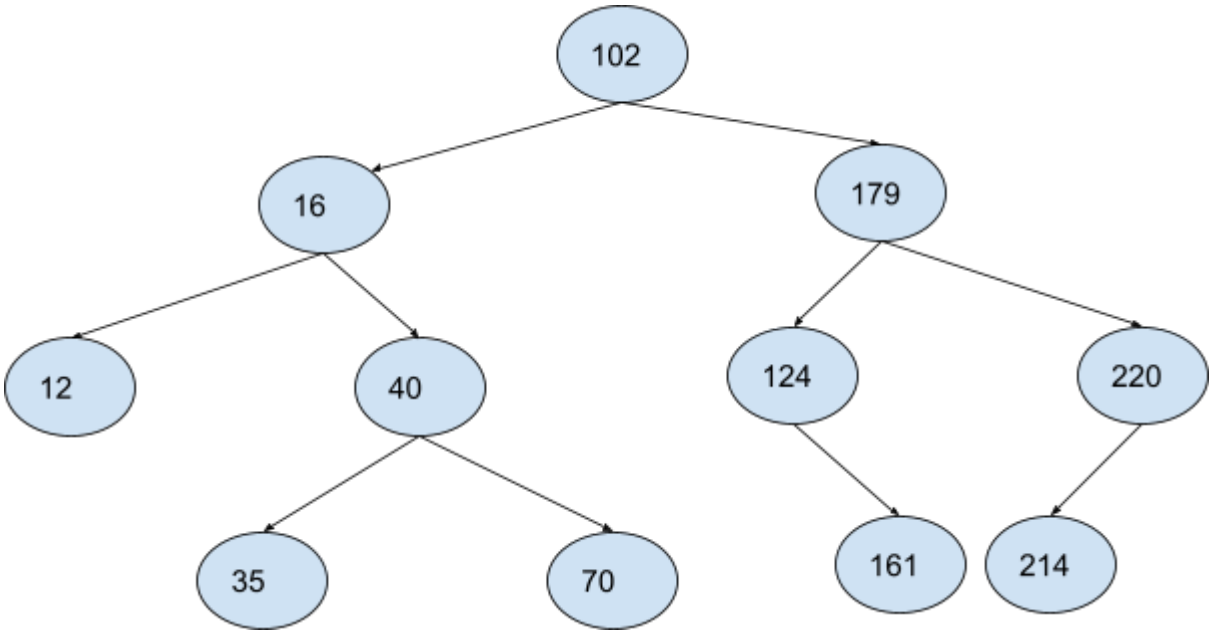
```
#define _GNU_SOURCE     /* Expose declaration of tdestroy() */
#include <search.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

static void *root = NULL;

static void *
xmalloc(size_t n)
{
   void *p;
   p = malloc(n);
   if (p)
      return p;
   fprintf(stderr, "insufficient memory\n");
   exit(EXIT_FAILURE);
}

static int
```

```
compare(const void *pa, const void *pb)
{
    if (*(int *) pa < *(int *) pb)
        return -1;
    if (*(int *) pa > *(int *) pb)
        return 1;
    return 0;
}

static void
action(const void *nodep, VISIT which, int depth)
{
    int *datap;

    switch (which) {
    case preorder:
        break;
    case postorder:
        datap = *(int **) nodep;
        printf("%6d\tdepth: %d\n", *datap, depth);
        break;
    case endorder:
        break;
    case leaf:
        datap = *(int **) nodep;
        printf("%6d\tdepth: %d\n", *datap, depth);
        break;
    }
}

int
main(void)
{
    int **val;
    int dummy[11] = {179, 102, 220, 124, 214, 40, 16, 12, 70, 161, 35};

    for (int i = 0; i < 11; i++) {
        int *ptr = xmalloc(sizeof(*ptr));
        *ptr = dummy[i];
        val = tsearch(ptr, &root, compare);
        if (val == NULL)
            exit(EXIT_FAILURE);
        else if (*val != ptr)
            free(ptr);
    }
    twalk(root, action);
    tdestroy(root, free);
    exit(EXIT_SUCCESS);
```

Output of tsearch example;

```
 12      depth: 2
 16      depth: 1
 35      depth: 3
 40      depth: 2
 70      depth: 3
102      depth: 0
124      depth: 2
161      depth: 3
179      depth: 1
214      depth: 3
220      depth: 2
```

Where the tree seems like below;



PROCESS MEMORY LAYOUT

- Whenever you run an executable file, the operating system creates a new process. A process is a live and running program that is loaded into the memory and has a unique Process Identifier (PID). The operating system is the sole responsible entity for spawning and loading new processes.
- A process remains running until it either exits normally, or the process is given a signal, such as SIGTERM, SIGINT, or SIGKILL, which eventually makes it exit. The SIGTERM and SIGINT signals can be ignored, but SIGKILL will kill the process immediately and forcefully.
- When creating a process, one of the first things that operating systems do is allocate a portion of memory dedicated to the process and then apply a predefined memory layout.
- The memory layout of an ordinary process is divided into multiple parts. Each part is called a segment. Each segment is a region of memory which has a definite task and it is supposed to store a specific type of data. You can see the following list of segments being part of the memory layout of a running process:
    - Uninitialized data segment or Block Started by Symbol (BSS) segment
    - Data segment
    - Text segment or Code segment
    - Stack segment
    - Heap segment
- In the memory layout of a running process, some segments come directly from the base executable object file, and the rest are built dynamically at runtime while the process is being loaded. The former layout is called the static memory layout, and the latter is called the dynamic memory layout.
- Static and dynamic memory layouts both have a predetermined set of segments. The content of the static memory layout is prewritten into the executable object file by the compiler, when compiling the source code. On the other hand, the content of the dynamic memory layout is written by the process instructions allocating memory for variables and arrays, and modifying them according to the program's logic.
- With all that said, we can guess the content of the static memory layout either by just looking at the source code or the compiled object file. But this is not that easy regarding the dynamic memory layout as it cannot be determined without running the program. In addition, different runs of the same executable file can lead to different content in the dynamic memory layout. In other words, the dynamic content of a process is unique to that process and it should be investigated while the process is still running.
- **size** <executable object file>

BSS Segment

- The purpose that we use the BSS segment for; either uninitialized global variables or global variables set to zero.
- These special global variables are part of the static layout and they become preallocated when a process is loading, and they never get deallocated until the process is alive. In other words, they have a static lifetime.

- Because of design concerns, we usually prefer to use local variables in our algorithms. Having too many global variables can increas

Data Segment

- Hold Initialized global variables.
- Other than global variables, we can have some static variables declared inside a function. These variables retain their values while calling the same function multiple times. These variables can be stored either in the Data segment or the BSS segment depending on the platform and whether they are initialized or not.
- We have readelf and objdump commands in order to see the content of ELF files.

Text Segment

- The linker writes the resulting machine-level instructions into the final executable object file. Since the Text segment, or the Code segment, contains all the machine-level instructions of a program, it should be located in the executable object file, as part of its static memory layout.
- The dynamic memory layout is actually the runtime memory of a process, and it exists as long as the process is running. When you execute an executable object file, a program called loader takes care of the execution. It spawns a new process and it creates the initial memory layout which is supposed to be dynamic. To form this layout, the segments found in the static layout will be copied from the executable object file. More than that, two new segments will also be added to it. Only then can the process proceed and become running.

Stack Segment

- As pointed out before, the Stack segment is usually limited in size, and it is not a good place to store big objects. If the Stack segment is full, the process cannot make any further function calls since the function call mechanism relies heavily on the functionality of the Stack segment.
- Used for function returns, function args, fixed lenght local variables

Heap Segment

- Dynamic memory allocation
- The Heap doesn't have any memory blocks that are allocated automatically. Instead, the programmer must use malloc or similar functions to obtain Heap memory blocks, one by one.
- The Heap has a large memory size.
- Memory allocation and deallocation inside Heap memory are managed by the programmer.
- Variables allocated from the Heap do not have any scope, unlike variables in the Stack.
- We can only use pointers to address a Heap memory block.
- Since the Heap segment is private to its owner process, we need to use a debugger to probe it

Allocating Memory on the Stack: **alloca()**

- Like the functions in the malloc package, alloca() allocates memory dynamically. However, instead of obtaining memory from the heap, alloca() obtains memory from the stack by increasing the size of the stack frame.
- Using alloca() to allocate memory has a few advantages over malloc(). One of these is that allocating blocks of memory is faster with alloca() than with malloc(), because alloca() is implemented by the compiler as inline code that directly adjusts the stack pointer. Furthermore, alloca() doesn't need to maintain a list of free blocks.

KERNEL TASKS

- **Process scheduling**: A computer has one or more central processing units (CPUs), which execute the instructions of programs. Like other UNIX systems, Linux is a preemptive multitasking operating system, Multitasking means that multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s). Preemptive means that the rules governing which processes receive use of the CPU and for how long are determined by the kernel process scheduler (rather than by the processes themselves).
- **Memory management**: While computer memories are enormous by the standards of a decade or two ago, the size of software has also correspondingly grown, so that physical memory (RAM) remains a limited resource that the kernel must share among processes in an equitable and efficient fashion. Like most modern operating systems, Linux employs virtual memory management, a technique that confers two main advantages:
    - Processes are isolated from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel.
    - Only part of a process needs to be kept in memory, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute.
- **Provision of a file system**: The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on.
- **Creation and termination of processes**: The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) that it needs in order to run. Such an instance of a running program is termed a process. Once a process has completed execution, the kernel ensures that the resources it uses are freed for subsequent reuse by later programs.
- **Access to devices**: The devices (mice, monitors, keyboards, disk and tape drives, and so on) attached to a computer allow communication of information between the computer and the outside world, permitting input, output, or both. The kernel provides programs with an interface that standardizes and simplifies access to devices, while at the same time arbitrating access by multiple processes to each device.
- **Networking**: The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes routing of network packets to the target system.
- **Provision of a system call application programming interface (API)**: Processes can request the kernel to perform various tasks using kernel entry points known as system calls.

DAEMON

- A daemon is a special-purpose process that is created and handled by the system in the same way as other processes, but which is distinguished by the following characteristics:
- It is long-lived. A daemon process is often started at system boot and remains in existence until the system is shut down.
- It runs in the background, and has no controlling terminal from which it can read input or to which it can write output.

PROCESSES

- The **fork()** system call allows one process, the parent, to create a new process, the child. This is done by making the new child process an (almost) exact duplicate of the parent: the child obtains copies of the parent's stack, data, heap, and text segments
- The **exit(status)** library function terminates a process, making all resources (memory, open file descriptors, and so on) used by the process available for subsequent reallocation by the kernel. The status argument is an integer that determines the termination status for the process. Using the wait() system call, the parent can retrieve this status
- The **wait(&status)** system call has two purposes. First, if a child of this process has not yet terminated by calling exit(), then wait() suspends execution of the process until one of its children has terminated. Second, the termination status of the child is returned in the status argument of wait().
- The **execve(pathname, argv, envp)** system call loads a new program (pathname, with argument list argv, and environment list envp) into a process's memory. The existing program text is discarded, and the stack, data, and heap segments are freshly created for the new program.
- Some other operating systems combine the functionality of fork() and exec() into a single operatio, a so-called **spawn,** that creates a new process that then executes a specified program.
- The key point to understanding fork() is to realize that after it has completed its work, two processes exist, and, in each process, execution continues from the point where fork() returns.
- The two processes are executing the same program text, but they have separate copies of the stack, data, and heap segments. The child's stack, data, and heap segments are initially exact duplicates of the corresponding parts of the parent's memory. After the fork(), each process can modify the variables in its stack, data, and heap segments without affecting the other process.
- It is important to realize that after a fork(), it is indeterminate which of the two processes is next scheduled to use the CPU. In poorly written programs, this indeterminacy can lead to errors known as race conditions. But generally, parent process is scheduled first.
- Open files are shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

- Conceptually, we can consider fork() as creating copies of the parent's text, data, heap, and stack segments. However, actually performing a simple copy of the parent's virtual memory pages into the new child process would be wasteful for a number of reasons—one being that a fork() is often followed by an immediate exec(), which replaces the process's text with a new program and reinitializes the process's data, heap, and stack segments. Most modern UNIX implementations, including Linux, use two techniques to avoid such wasteful copying:
    - The kernel marks the text segment of each process as read-only, so that a process can't modify its own code. This means that the parent and child can share the same text segment. The fork() system call creates a text segment for the child by building a set of per-process page-table entries that refer to the same physical memory page frames already used by the parent.
    - For the pages in the data, heap, and stack segments of the parent process, the kernel employs a technique known as copy-on-write. Initially, the kernel sets things up so that the page-table entries for these segments refer to the same physical memory pages as the corresponding page-table entries in the parent, and the pages themselves are marked read-only. After the fork(), the kernel traps any attempts by either the parent or the child to modify one of these pages, and makes a duplicate copy of the about-to-be-modified page. This new page copy is assigned to the faulting process, and the corresponding page-table entry for the other process is adjusted appropriately. From this point on, the parent and child can each modify their private copies of the page, without the changes being visible to the other process.
- Like fork(), **vfork()** is used by the calling process to create a new child process. However, vfork() is expressly designed to be used in programs where the child performs an immediate exec() call.
- The parent that is, by default, run first after a fork(). This default can be changed by assigning a nonzero value to the Linux-specific **/proc/sys/kernel/sched_child_runs_first** file.
- A process may terminate in two general ways. One of these is abnormal termination, caused by the delivery of a signal whose default action is to terminate the process (with or without a core dump). Alternatively, a process can terminate normally, using the _exit() system call.During both normal and abnormal termination of a process, the following actions occur:
    - Open file descriptors, directory streams, message catalog descriptors, and conversion descriptors (see the iconv_open(3) manual page) are closed
    - As a consequence of closing file descriptors, any file locks held by this process are released.
    - Any attached System V shared memory segments are detached, and the shm_nattch counter corresponding to each segment is decremented by one
    - If this is the controlling process for a controlling terminal, then the SIGHUP signal is sent to each process in the controlling terminal's foreground process group, and the terminal is disassociated from the session
    - Any POSIX named semaphores that are open in the calling process are closed as though sem_close() were called.
    - Any POSIX message queues that are open in the calling process are closed as though mq_close() were called.

    - If, as a consequence of this process exiting, a process group becomes orphaned and there are any stopped processes in that group, then all processes in the group are sent a SIGHUP signal followed by a SIGCONT signal.
    - Any memory locks established by this process using mlock() or mlockall() are removed.
    - Any memory mappings established by this process using mmap() are unmapped.
- An exit handler is a programmer-supplied function that is registered at some point during the life of the process and is then automatically called during normal process termination via **exit()**
    - **atexit()** used to register an exit handler
- **waitpid()**
    - If a parent process has created multiple children, it is not possible to wait() for the completion of a specific child; we can only wait for the next child that terminates.
    - If no child has yet terminated, wait() always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.
    - Using wait(), we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as SIGSTOP or SIGTTIN) or when a stopped child is resumed by delivery of a SIGCONT signal.
- wait or waitpid approaches are inconvenient. On the one hand, we may not want the parent to be blocked waiting for a child to terminate. To get around these problems, we can employ a handler for the **SIGCHLD** signal.
- The SIGCHLD signal is sent to a parent process whenever one of its children terminates. By default, this signal is ignored, but we can catch it by installing a signal handler. Within the signal handler, we can use wait() (or similar) to reap the zombie child. However, there is a subtlety to consider in this approach.

Process Priorities (Nice Values)

- On Linux, as with most other UNIX implementations, the default model for scheduling processes for use of the CPU is round-robin time-sharing. Under this model, each process in turn is permitted to use the CPU for a brief period of time, known as a time slice or quantum. Round-robin time-sharing satisfies two important requirements of an interactive multitasking system:
    - **Fairness**: Each process gets a share of the CPU.
    - **Responsiveness**: A process doesn't need to wait for long periods before it receives use of the CPU.
- Under the **round-robin time-sharing algorithm**, processes can't exercise direct control over when and for how long they will be able to use the CPU. By default, each process in turn receives use of the CPU until its time slice runs out or it voluntarily gives up the CPU (for example, by putting itself to sleep or performing a disk read). If all processes attempt to use the CPU as much as possible (i.e., no process ever sleeps or blocks on an I/O operation), then they will receive a roughly equal share of the CPU.

- However, one process attribute, the nice value, allows a process to indirectly influence the kernel's scheduling algorithm. Each process has a nice value in the range –20 (high priority) to +19 (low priority); the default is 0 (refer to Figure 35-1). In traditional UNIX implementations, only privileged processes can assign themselves (or other processes) a negative (high) priority. (We'll explain some Linux differences in Section 35.3.2.) Unprivileged processes can only lower their priority, by assuming a nice value greater than the default of 0. By doing this, they are being "nice" to other processes, and this fact gives the attribute its name.

Realtime Scheduling (Hard Realtime)

- A realtime application must provide a guaranteed maximum response time for external inputs. In many cases, these guaranteed maximum response times must be quite small (e.g., of the order of a fraction of a second). For example, a slow response by a vehicle navigation system could be catastrophic. To satisfy this requirement, the kernel must provide the facility for a high-priority process to obtain control of the CPU in a timely fashion, preempting any process that may currently be running.
    - A high-priority process should be able to maintain exclusive access to the CPU until it completes or voluntarily relinquishes the CPU.
    - A realtime application should be able to control the precise order in which its component processes are scheduled

Posix Realtime (Soft Realtime)

- the POSIX realtime process scheduling API doesn't satisfy all of these requirements. In particular, it provides no way for an application to guarantee response times for handling input.
- To make such guarantees requires operating system features that are not part of the mainline Linux kernel (nor most other standard operating systems).
- The POSIX API merely provides us with so-called soft realtime, allowing us to control which processes are scheduled for use of the CPU.

Preventing Realtime Processes from Locing Up the System

- Establish a suitably low soft CPU time resource limit (RLIMIT_CPU) using setrlimit(). If the process consumes too much CPU time, it will be sent a SIGXCPU signal, which kills the process by default.
- Set an alarm timer using alarm(). If the process continues running for a wall clock time that exceeds the number of seconds specified in the alarm() call, then it will be killed by a SIGALRM signal.
- Create a **watchdog** process that runs with a high realtime priority

CPU Affinity

- When a process is rescheduled to run on a multiprocessor system, it doesn't necessarily run on the same CPU on which it last executed. The usual reason it may run on another CPU is that the original CPU is already busy.
- When a process changes CPUs, there is a performance impact: in order for a line of the process's data to be loaded into the cache of the new CPU, it must first be invalidated (i.e., either discarded if it is unmodified, or flushed to main memory if it was modified), if present in the cache of the old CPU. (To prevent cache inconsistencies, multiprocessor architectures allow data to be kept in only one CPU cache at a time.) This invalidation costs execution time. Because of this performance impact, the Linux (2.6) kernel tries to ensure soft CPU affinity for a process—wherever possible, the process is rescheduled to run on the same CPU.

//Here is an example program showing how you might write a function similar to the built-in system. It executes its command argument using the equivalent of 'sh -c command'.

```
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Execute the command using this shell program.  */
#define SHELL "/bin/sh"

int my_system (const char *command)
{
  int status;
  pid_t pid;

  pid = fork ();
  if (pid == 0) {
      /* This is the child process.  Execute the shell command. */
      execl (SHELL, SHELL, "-c", command, NULL);
      _exit (EXIT_FAILURE);
    }
  else if (pid < 0)
    /* The fork failed.  Report failure.  */
    status = -1;
  else
    /* This is the parent process.  Wait for the child to complete.  */
    if (waitpid (pid, &status, 0) != pid)
      status = -1;
  return status;
}
```

There are a couple of things you should pay attention to in this example.

- Remember that the first argv argument supplied to the program represents the name of the program being executed. That is why, in the call to execl, SHELL is supplied once to name the program to execute and a second time to supply a value for argv[0].
- The execl call in the child process doesn't return if it is successful. If it fails, you must do something to make the child process terminate. Just returning a bad status code with return would leave two processes running the original program. Instead, the right behavior is for the child process to report failure to its parent process.
- Call _exit to accomplish this. The reason for using _exit instead of exit is to avoid flushing fully buffered streams such as stdout. The buffers of these streams probably contain data that was copied from the parent process by the fork, data that will be output eventually by the parent process. Calling exit in the child would output the data twice.

THREADS

- Include **#include <pthread.h>** and link -**lpthread**
- In the traditional UNIX API, errno is a global integer variable. However, this doesn't suffice for threaded programs. If a thread made a function call that returned an error in a global errno variable, then this would confuse other threads that might also be making function calls and checking errno. In other words, race conditions would result. Therefore, in threaded programs, each thread has its own errno value.
- Sharing information between threads is easy and fast. It is just a matter of copying data into shared (global or heap) variables.
- Thread creation is faster than process creation—typically, ten times faster or better.
- The pthread_join() function waits for the thread identified by thread to terminate. (If that thread has already terminated, pthread_join() returns immediately.)
- Calling pthread_join() for a thread ID that has been previously joined can lead to unpredictable behavior
- If a thread is not detached, then we must join with it using pthread_join(). If we fail to do this, then, when the thread terminates, it produces the thread equivalent of a zombie process.
- The task that pthread_join() performs for threads is similar to that performed by waitpid() for processes. However, there are some notable differences:
    - Threads are peers. Any thread in a process can use pthread_join() to join with any other thread in the process. For example, if thread A creates thread B, which creates thread C, then it is possible for thread A to join with thread C, or vice versa. This differs from the hierarchical relationship between processes. When a parent process creates a child using fork(), it is the only process that can wait() on that child. There is no such relationship between the thread that calls pthread_create() and the resulting new thread.
    - There is no way of saying "join with any thread" (for processes, we can do this using the call waitpid(–1, &status, options)); nor is there a way to do a nonblocking join.
- **Detaching a Thread**: By default, a thread is joinable, meaning that when it terminates, another thread can obtain its return status using pthread_join(). Sometimes, we don't care about the thread's return status; we simply want the system to automatically clean up and remove the thread when it terminates. In this case, we can mark the thread as detached, by making a call to pthread_detach() specifying the thread's identifier in thread. Once a thread has been detached, it is no longer possible to use pthread_join() to obtain its return status, and the thread can't be made joinable again.

Threads Versus Processes

- Sharing data between threads is easy. By contrast, sharing data between processes requires more work (e.g., creating a shared memory segment or using a pipe).
- Thread creation is faster than process creation; context-switch time may be lower for threads than for processes.
- Using threads can have some disadvantages compared to using processes:

- When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner. Multiprocess applications don't need to be concerned with this.
- A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. By contrast, processes are more isolated from one another.
- Each thread is competing for use of the finite virtual address space of the host process. In particular, each thread's stack and thread-specific data (or thread-local storage) consumes a part of the process virtual address space, which is consequently unavailable for other threads. Although the available virtual address space is large (e.g., typically 3 GB on x86-32), this factor may be a significant limitation for processes employing large numbers of threads or threads that require large amounts of memory. By contrast, separate processes can each employ the full range of available virtual memory (subject to the limitations of RAM and swap space).

THREAD SYNCHRONIZATION

Mutexes

- Protecting Accesses to Shared Variables
- A mutex has two states: locked and unlocked. At any moment, at most one thread may hold the lock on a mutex. Attempting to lock a mutex that is already locked either blocks or fails with an error, depending on the method used to place the lock.
- Sometimes, a thread needs to simultaneously access two or more different shared resources, each of which is governed by a separate mutex. When more than one thread is locking the same set of mutexes, deadlock situations can arise. Figure 30-3 shows an example of a **deadlock** in which each thread successfully locks one mutex, and then tries to lock the mutex that the other thread has already locked. Both threads will remain blocked indefinitely.
    - The simplest way to avoid such deadlocks is to define a mutex hierarchy. When threads can lock the same set of mutexes, they should always lock them in the same order.
    - An alternative strategy that is less frequently used is "try, and then back off."

```
// int pthread_mutex_init (pthread_mutex_t *__mutex,
                           const pthread_mutexattr_t *__mutexattr)
//int pthread_mutex_destroy(pthread_mutex_t *mutex)
// int pthread_mutex_lock (pthread_mutex_t *__mutex)
// int pthread_mutex_trylock (pthread_mutex_t *__mutex)
// int pthread_mutex_timedlock (pthread_mutex_t *__restrict __mutex,
                                const struct timespec *__restrict
                                __abstime)
// int pthread_mutex_unlock (pthread_mutex_t *__mutex)
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
	pthread_mutex_lock(&lock);

	unsigned long i = 0;
	counter += 1;
	printf("\n Job %d has started\n", counter);

	for (i = 0; i < (0xFFFFFFFF); i++)
		;

	printf("\n Job %d has finished\n", counter);

	pthread_mutex_unlock(&lock);

	return NULL;
}

int main(void)
{
	int i = 0;
	int error;

	if (pthread_mutex_init(&lock, NULL) != 0) {
		printf("\n mutex init has failed\n");
		return 1;
	}

	while (i < 2) {
		error = pthread_create(&(tid[i]),
				NULL,
				&trythis, NULL);
		if (error != 0)
			printf("\nThread can't be created :[%s]",
				strerror(error));
		i++;
```

```
        }

        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
        pthread_mutex_destroy(&lock);

        return 0;
}
```

→ **Output**;
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished

→ **Output before mutex**
Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished

Condition Variables

- The condition variable is used to signal changes in the variable's state
- The principal condition variable operations are signal and wait. The signal operation is a notification to one or more waiting threads that a shared variable's state has changed. The wait operation is the means of blocking until such a notification is received.
- They are used with mutexes

```
// C program to implement cond(), signal()
// and wait() functions
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

// Declaration of thread condition variable
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;

// declaring mutex
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int done = 1;

// Thread function
```

```
void* foo()
{

	// acquire a lock
	pthread_mutex_lock(&lock);
	if (done == 1) {

		// let's wait on condition variable cond1
		done = 2;
		printf("Waiting on condition variable cond1\n");
		pthread_cond_wait(&cond1, &lock);
	}
	else {

		// Let's signal condition variable cond1
		printf("Signaling condition variable cond1\n");
		pthread_cond_signal(&cond1);
	}

	// release lock
	pthread_mutex_unlock(&lock);

	printf("Returning thread\n");

	return NULL;
}

// Driver code
int main()
{
	pthread_t tid1, tid2;

	// Create thread 1
	pthread_create(&tid1, NULL, foo, NULL);

	// sleep for 1 sec so that thread 1
	// would get a chance to run first
	sleep(1);

	// Create thread 2
	pthread_create(&tid2, NULL, foo, NULL);

	// wait for the completion of thread 2
	pthread_join(tid2, NULL);

	return 0;
}
```

Output →
Waiting on condition variable cond1
Signaling condition variable cond1
Returning thread
Returning thread

Semaphores

- There are two basic sorts of semaphores: **binary semaphores**, which never take on values other than zero or one, and **counting semaphores**, which can take on arbitrary nonnegative values.
- A binary semaphore is logically just like a mutex.
- Counting semaphore is used for multi process systems for example.

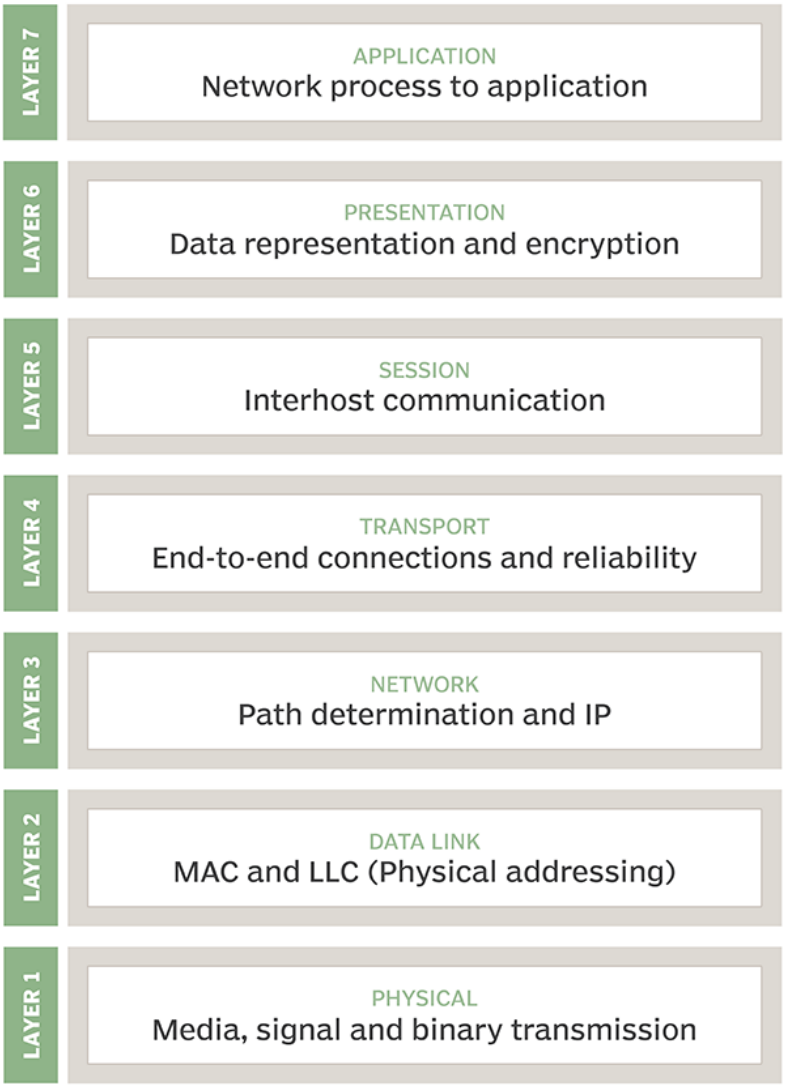
```
#include <semaphore.h>
sem_t SEM_INFORM;
res = sem_init (&SEM_INFORM, 0, 0);
sem_post(&SEM_INFORM);
sem_wait(&SEM_INFORM);
sem_trywait (sem_t *__sem)
sem_destroy (sem_t *__sem)
```

Thread Safety (Reentrancy)

- Use **reentrant** functions inside the thread
- There are various methods of rendering a function thread-safe. One way is to associate a mutex with the function (or perhaps with all of the functions in a library, if they all share the same global variables), lock that mutex when the function is called, and unlock it when the function returns. This approach has the virtue of simplicity. On the other hand, it means that only one thread at a time can execute the function—we say that access to the function is serialized. If the threads spend a significant amount of time executing this function, then this serialization results in a loss of concurrency, because the threads of a program can no longer execute in parallel.
- A more sophisticated solution is to associate the mutex with a shared variable. We then determine which parts of the function are critical sections that access the shared variable, and acquire and release the mutex only during the execution of these critical sections. This allows multiple threads to execute the function at the same time and to operate in parallel, except when more than one thread needs to execute a critical section.

NETWORKING

OSI Layers



Physical Layer

The lowest layer of the OSI Model is concerned with electrically or optically transmitting raw unstructured data bits across the network from the physical layer of the sending device to the physical layer of the receiving device. It can include specifications such as voltages, pin layout, cabling, and radio frequencies. At the physical layer, one might find “physical” resources such as network hubs, cabling, repeaters, network adapters or modems.

Data Link Layer

At the data link layer, directly connected nodes are used to perform node-to-node data transfer where data is packaged into frames. The data link layer also corrects errors that may have occurred at the physical layer.

The data link layer encompasses two sub-layers of its own. The first, media access control (**MAC**), provides flow control and multiplexing for device transmissions over a network. The second, the logical link control (**LLC**), provides flow and error control over the physical medium as well as identifies line protocols.

Network Layer

The network layer is responsible for receiving frames from the data link layer, and delivering them to their intended destinations among based on the addresses contained inside the frame. The network layer finds the destination by using logical addresses, such as **IP** (internet protocol). At this layer, routers are a crucial component used to quite literally route information where it needs to go between networks.

Transport Layer

The transport layer manages the delivery and error checking of data packets. It regulates the size, sequencing, and ultimately the transfer of data between systems and hosts. One of the most common examples of the transport layer is **TCP** or **UDP**.

Session Layer - Presentation Layer - Application Layer

We are tolking about sockets and ports here.

TCP vs UDP

Basis	Transmission control protocol (TCP)	User datagram protocol (UDP)
Type of Service	TCP is a connection-oriented protocol. Connection-orientation means that the communicating devices should establish a connection before transmitting data and should close the connection after transmitting the data.	UDP is the Datagram-oriented protocol. This is because there is no overhead for opening a connection, maintaining a connection, and terminating a connection. UDP is efficient for broadcast and multicast types of network transmission.
Reliability	TCP is reliable as it guarantees the delivery of data to the destination router.	The delivery of data to the destination cannot be guaranteed in UDP.
Error checking mechanism	TCP provides extensive error-checking mechanisms. It is because it provides flow control and acknowledgment of data.	UDP has only the basic error checking mechanism using checksums.
Acknowledgment	An acknowledgment segment is present.	No acknowledgment segment.

Sequence (Ordering)	Sequencing of data is a feature of Transmission Control Protocol (TCP). This means that packets arrive in order at the receiver.	There is no sequencing of data in UDP. If the order is required, it has to be managed by the application layer.
Speed	TCP is comparatively slower than UDP.	UDP is faster, simpler, and more efficient than TCP.
Retransmission	Retransmission of lost packets is possible in TCP, but not in UDP.	There is no retransmission of lost packets in the User Datagram Protocol (UDP).
Broadcasting	TCP doesn't support Broadcasting.	UDP supports Broadcasting.
Protocols	TCP is used by HTTP, HTTPs, FTP, SMTP and Telnet.	UDP is used by DNS, DHCP, TFTP, SNMP, RIP, and VoIP.
Stream Type	The TCP connection is a byte stream.	UDP connection is message stream.

IPV4 vs IPV6

IPV6 Benefits

- No more NAT (Network Address Translation)
- No more private address collisions
- Insane amount of IP addresses (340 undecillion addresses) --> visualizer
- Improved security
- Better response time (more efficient routing)
- Easier administration (self-configuration mechanism)
- True quality of service (QoS), also called "flow labeling"

IPV6 Adverses

- Doesn't include a checksum in the header. IPv6 no longer has a header
- Checksum to protect the IP header, meaning that when a packet header is
- Corrupted by transmission errors, the packet may be delivered incorrectly.
- Pv4 and IPv6 machines cannot communicate directly to each other.
- The process of making the switch to IPv6 from IPv4 is slow and tedious.
- Understanding IPv6 subnetting can be difficult on its own. (Learning Curve)

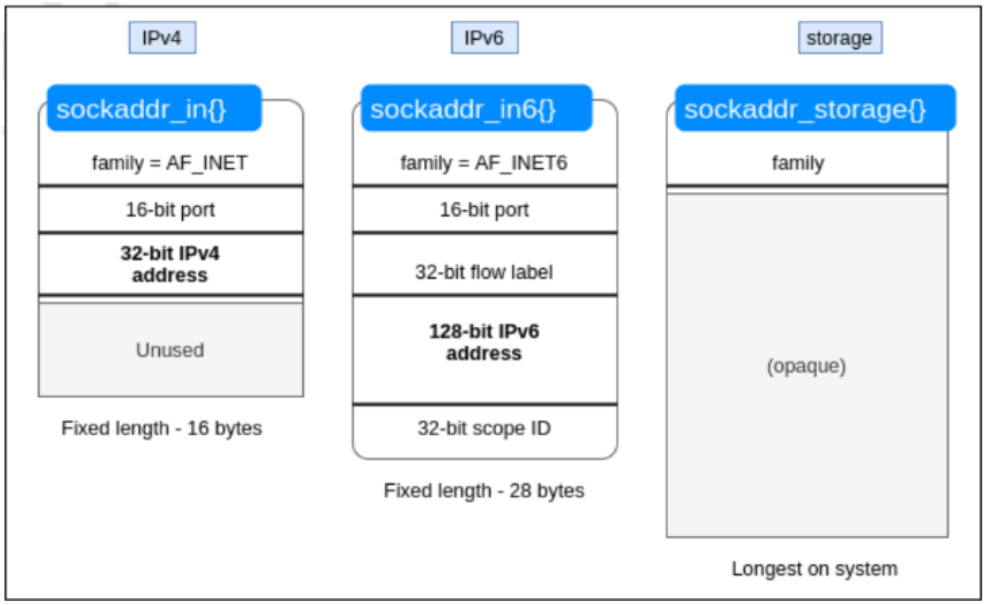
IP Syntax

- IPv6 addresses range;
    - **from** 0000:0000:0000:0000:0000:0000:0000:0000
    - **to** ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
    - Eg: 2004:0cb8:82a3:08d3:1319:8a2e:0370:7334
- **Rule1**: Omit leading zeros: Specify IPv6 addresses by omitting leading zeros.
    - Eg: 1050:0000:0000:0000:0005:0600:300c:326b **can be written as** 1050:0:0:0:5:600:300c:326b
- **Rule2**: Double colon: Specify IPv6 addresses by using double colons (::) in place of a series of zeros.
    - Eg: ff06:0:0:0:0:0:0:c3 **can be written as** ff06::c3

URL Syntax

- IPV4 Style--> http://192.168.100.100:8080
- IPV6 Style--> http://[fe80::f617:b8ff:feca:1995]:7547

API Differences



All This Structures are castable to generic socket address: **sockaddr**

NOTE THAT, No vice versa

So, **read, write, listen, accept, sendto, recvfrom, send, recv, connect** APIs are same (struct sockaddr * casting is required)

```
1  struct sockaddr_in address;
2
3  socket(AF_INET, SOCK_DGRAM, 0);
4
5  (void)memset((void *)&address, 0, sizeof(struct sockaddr_in));
6  address.sin_family = AF_INET;
7  address.sin_port = htons(PORT);
8  address.sin_addr.s_addr = inet_addr(LOCALHOST); //htonl(
   INADDR_ANY);
9
10 sendto(fd, (const char *)message_buffer, strlen(message_buffer),
11              MSG_CONFIRM, (struct sockaddr *) &address,
12              sizeof(address));
```

```
1  struct sockaddr_in6 address;
2
3  socket(AF_INET6, SOCK_DGRAM, 0);
4
5  bzero(&address, sizeof (struct sockaddr_in6));
6  address.sin6_family = AF_INET6;
7  address.sin6_port = htons(PORT);
8  inet_pton(AF_INET6, LOCALHOST, &address.sin6_addr);
9
10 sendto(fd, (const char *)message_buffer, strlen(message_buffer),
11              MSG_CONFIRM, (struct sockaddr *) &address,
12              sizeof(address));
```

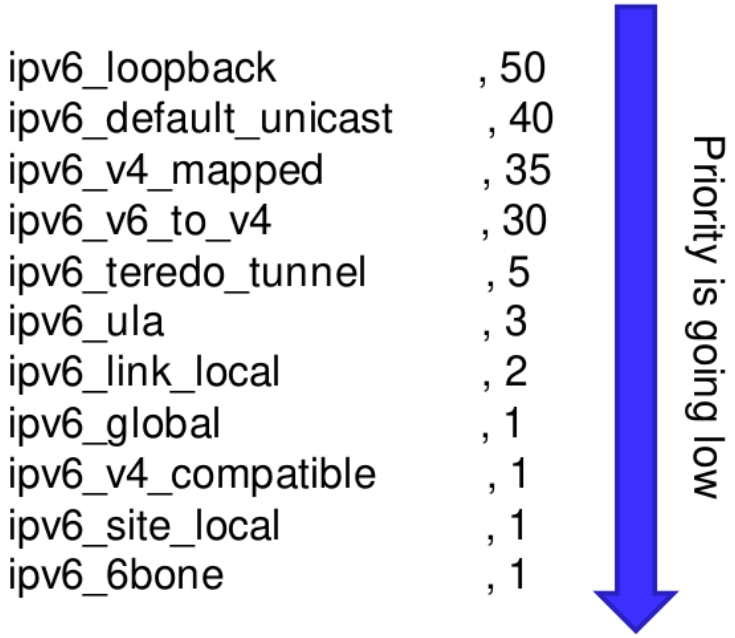
Dual Stack

- For servers, one option is creating both IPV4 and IPV6 sockets and maintain them simultaneously.
- Other option is using generic APIs to create a single process that maintain all socket types.
- Only criteria for a server to support dual-stack implementation, they must listen IPV6 WildCard IP address (::) and "IPV6_V6ONLY" socket option must be disabled.
- Imagine a server that serves only the clients in the same host, listening wildcard IP address causes a security risk because these kinds of servers can accept any connection from any interface, not just localhost.
- So, there are there are two options here;
    - Get the client address and filter it. If it does not belong to localhost, close the connection.
    - Or, set the socket option "SO_BINDTODEVICE". If a socket is bound to an interface, only packets received from that particular interface are processed by the socket.

IPV6 Address Scopes

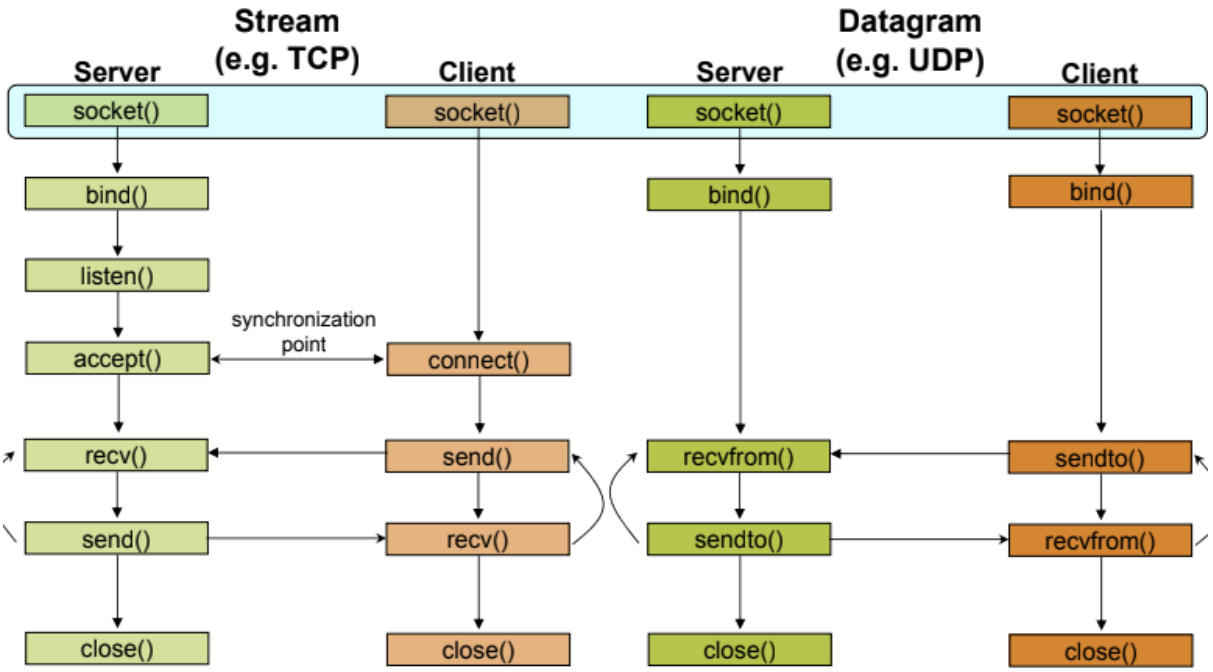
IP	Scopes
::1/128	Loopback
::/0	Wildcard
::ffff:0:0/96	IPv4-mapped IPv6 address
2002::/16	6to4
2001::/32	Teredo tunneling
FC00::/7	Unique local address
::/96	IPv4-compatible addresses (deprecated)
fec0::/10	Site-local address (deprecated)
3ffe::/16	6bone (returned)
ffc0 and fe80	Link-local
others	Global

IPV6 Address Pripritization (Happy Eyeball, RFC 8305)



- Bigger precedence IP adres is more prior than others.
- Second element in the list **must be IPV4**

Sockets



Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

- **bind**: associates and reserves a port for use by the socket
- **inet_pton**: string to network address conversion
    - **inet_pton**(**AF_INET**, addr, &(((struct **sockaddr_in** *)converted_addr)->sin_addr)
    - **inet_pton**(**AF_INET6**, addr, &(((struct **sockaddr_in6** *)converted_addr)->sin6_addr)
- inet_ntop: network to string address conversion

    - **inet_ntop**(**AF_INET**, &(((struct **sockaddr_in** *)addr)->sin_addr), conv_addr, **INET_ADDRSTRLEN**)
    - **inet_ntop**(**AF_INET6**, &(((struct **sockaddr_in6** *)addr)->sin6_addr), conv_addr, **INET6_ADDRSTRLEN**)
- **getaddrinfo**() returns one or more addrinfo structures by using IP port data, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2)
    - hints.ai_family = AF_UNSPEC;   /* Allow IPv4 or IPv6 */
    - hints.ai_flags = AI_PASSIVE;   /* For wildcard IP address */
    - hints.ai_protocol = 0;   /* Any protocol */
- **getnameinfo**() function is the inverse of getaddrinfo(3): it converts a socket address to a corresponding host and service

Socket Options

- SO_BINDTODEVICE
- SO_KEEPALIVE
- SO_LINGER
- SO_REUSEADDR
- SO_REUSEPORT

IO Multiplexing

- **select** and **poll** methos can be used.
- **poll** is **better** choice but select can be used because of **portability**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/poll.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT  12345

#define TRUE             1
#define FALSE            0

main (int argc, char *argv[])
{
  int    len, rc, on = 1;
  int    listen_sd = -1, new_sd = -1;
  int    desc_ready, end_server = FALSE, compress_array = FALSE;
  int    close_conn;
```

```
char   buffer[80];
struct sockaddr_in6   addr;
int    timeout;
struct pollfd fds[200];
int    nfds = 1, current_size = 0, i, j;

/*************************************************************/
/* Create an AF_INET6 stream socket to receive incoming      */
/* connections on                                            */
/*************************************************************/
listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
if (listen_sd < 0)
{
  perror("socket() failed");
  exit(-1);
}

/*************************************************************/
/* Allow socket descriptor to be reuseable                   */
/*************************************************************/
rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
                (char *)&on, sizeof(on));
if (rc < 0)
{
  perror("setsockopt() failed");
  close(listen_sd);
  exit(-1);
}

/*************************************************************/
/* Set socket to be nonblocking. All of the sockets for      */
/* the incoming connections will also be nonblocking since   */
/* they will inherit that state from the listening socket.   */
/*************************************************************/
rc = ioctl(listen_sd, FIONBIO, (char *)&on);
if (rc < 0)
{
  perror("ioctl() failed");
  close(listen_sd);
  exit(-1);
}

/*************************************************************/
/* Bind the socket                                           */
/*************************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin6_family      = AF_INET6;
memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
```

```
addr.sin6_port       = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
  perror("bind() failed");
  close(listen_sd);
  exit(-1);
}

/*************************************************************/
/* Set the listen back log                                   */
/*************************************************************/
rc = listen(listen_sd, 32);
if (rc < 0)
{
  perror("listen() failed");
  close(listen_sd);
  exit(-1);
}

/*************************************************************/
/* Initialize the pollfd structure                           */
/*************************************************************/
memset(fds, 0 , sizeof(fds));

/*************************************************************/
/* Set up the initial listening socket                       */
/*************************************************************/
fds[0].fd = listen_sd;
fds[0].events = POLLIN;
/*************************************************************/
/* Initialize the timeout to 3 minutes. If no                */
/* activity after 3 minutes this program will end.           */
/* timeout value is based on milliseconds.                   */
/*************************************************************/
timeout = (3 * 60 * 1000);

/*************************************************************/
/* Loop waiting for incoming connects or for incoming data   */
/* on any of the connected sockets.                          */
/*************************************************************/
do
{
  /***********************************************************/
  /* Call poll() and wait 3 minutes for it to complete.      */
  /***********************************************************/
  printf("Waiting on poll()...\n");
```

```
rc = poll(fds, nfds, timeout);

/*************************************************************/
/* Check to see if the poll call failed.                     */
/*************************************************************/
if (rc < 0)
{
  perror("  poll() failed");
  break;
}

/*************************************************************/
/* Check to see if the 3 minute time out expired.            */
/*************************************************************/
if (rc == 0)
{
  printf("  poll() timed out.  End program.\n");
  break;
}


/*************************************************************/
/* One or more descriptors are readable.  Need to            */
/* determine which ones they are.                            */
/*************************************************************/
current_size = nfds;
for (i = 0; i < current_size; i++)
{
  /***********************************************************/
  /* Loop through to find the descriptors that returned      */
  /* POLLIN and determine whether it's the listening         */
  /* or the active connection.                               */
  /***********************************************************/
  if(fds[i].revents == 0)
    continue;

  /***********************************************************/
  /* If revents is not POLLIN, it's an unexpected result,    */
  /* log and end the server.                                 */
  /***********************************************************/
  if(fds[i].revents != POLLIN)
  {
    printf("  Error! revents = %d\n", fds[i].revents);
    end_server = TRUE;
    break;

  }
  if (fds[i].fd == listen_sd)
```

```
{
  /*************************************************/
  /* Listening descriptor is readable.             */
  /*************************************************/
  printf("  Listening socket is readable\n");

  /*************************************************/
  /* Accept all incoming connections that are      */
  /* queued up on the listening socket before we   */
  /* loop back and call poll again.                */
  /*************************************************/
  do
  {
    /***********************************************/
    /* Accept each incoming connection. If         */
    /* accept fails with EWOULDBLOCK, then we      */
    /* have accepted all of them. Any other        */
    /* failure on accept will cause us to end the  */
    /* server.                                     */
    /***********************************************/
    new_sd = accept(listen_sd, NULL, NULL);
    if (new_sd < 0)
    {
      if (errno != EWOULDBLOCK)
      {
        perror("  accept() failed");
        end_server = TRUE;
      }
      break;
    }

    /***********************************************/
    /* Add the new incoming connection to the      */
    /* pollfd structure                            */
    /***********************************************/
    printf("  New incoming connection - %d\n", new_sd);
    fds[nfds].fd = new_sd;
    fds[nfds].events = POLLIN;
    nfds++;

    /***********************************************/
    /* Loop back up and accept another incoming    */
    /* connection                                  */
    /***********************************************/
  } while (new_sd != -1);
}

/*************************************************/
```

```
/* This is not the listening socket, therefore an        */
/* existing connection must be readable                  */
/*********************************************************/

else
{
  printf("  Descriptor %d is readable\n", fds[i].fd);
  close_conn = FALSE;
  /*******************************************************/
  /* Receive all incoming data on this socket            */
  /* before we loop back and call poll again.            */
  /*******************************************************/

  do
  {
    /*****************************************************/
    /* Receive data on this connection until the         */
    /* recv fails with EWOULDBLOCK. If any other         */
    /* failure occurs, we will close the                 */
    /* connection.                                       */
    /*****************************************************/
    rc = recv(fds[i].fd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
      if (errno != EWOULDBLOCK)
      {
        perror("  recv() failed");
        close_conn = TRUE;
      }
      break;
    }

    /*****************************************************/
    /* Check to see if the connection has been           */
    /* closed by the client                              */
    /*****************************************************/
    if (rc == 0)
    {
      printf("  Connection closed\n");
      close_conn = TRUE;
      break;
    }

    /*****************************************************/
    /* Data was received                                 */
    /*****************************************************/
    len = rc;
    printf("  %d bytes received\n", len);
```

```
      /*****************************************************/
      /* Echo the data back to the client                  */
      /*****************************************************/
      rc = send(fds[i].fd, buffer, len, 0);
      if (rc < 0)
      {
        perror("  send() failed");
        close_conn = TRUE;
        break;
      }

    } while(TRUE);

    /*****************************************************/
    /* If the close_conn flag was turned on, we need     */
    /* to clean up this active connection. This          */
    /* clean up process includes removing the            */
    /* descriptor.                                       */
    /*****************************************************/
    if (close_conn)
    {
      close(fds[i].fd);
      fds[i].fd = -1;
      compress_array = TRUE;
    }


  }  /* End of existing connection is readable             */
} /* End of loop through pollable descriptors              */

/***********************************************************/
/* If the compress_array flag was turned on, we need       */
/* to squeeze together the array and decrement the number  */
/* of file descriptors. We do not need to move back the    */
/* events and revents fields because the events will always*/
/* be POLLIN in this case, and revents is output.          */
/***********************************************************/
if (compress_array)
{
  compress_array = FALSE;
  for (i = 0; i < nfds; i++)
  {
    if (fds[i].fd == -1)
    {
      for(j = i; j < nfds; j++)
      {
        fds[j].fd = fds[j+1].fd;
```

```
        }
        i--;
        nfds--;
      }
    }
  }

 } while (end_server == FALSE); /* End of serving running.    */

 /*************************************************************/
 /* Clean up all of the sockets that are open                 */
 /*************************************************************/
 for (i = 0; i < nfds; i++)
 {
   if(fds[i].fd >= 0)
     close(fds[i].fd);
 }
}
```

C Notes

file holes

What happens if a program seeks past the end of a file, and then performs I/O?

volatile

prevents compiler optimizations of arithmetic operations on 'glob'

To create an image with C

pgm format can be used

getopt_long - optional arguments

can be used but while typing it via console, there should not be an empty space. This is confusing. To overcome it, the below code snip could be used;

```
#define OPTIONAL_ARGUMENT_IS_PRESENT \
    ((optarg == NULL && optind < argc && argv[optind][0] != '-') \
     ? (bool) (optarg = argv[optind++]) \
     : (optarg != NULL))

case 'd':
        if (OPTIONAL_ARGUMENT_IS_PRESENT) {
                //do something with optarg
        }
        break;
```

parameter hiding in C

could be done by using **static** keyword for a variable

polymorphism in C

could be done by using **variadic** functions. With the below example, you may call the function with 2 arguments or three arguments.

```
int sipc_send_bradcast_data(void *data, unsigned int len, ...)
{
        va_list args;
        const char *fmt = "%d";
        char buffer[BUFFER_SIZE];
        char *ptr = NULL;
        unsigned long timeout = 0;

        if (!data || !len) {
```

```
                errorf("args cannot be NULL\n");
                return NOK;
        }

        va_start(args, fmt);
        vsnprintf(buffer, sizeof(buffer) - 1, fmt, args);
        va_end(args);

        timeout = strtoul(buffer, &ptr, 10);

        //do something with timeout
}
```

function pointer example

```
int sipc_register(char *title, int (*callback)(void *, unsigned int), ...)
{
        return callback((void *)”deneme”, 6);
}

int my_callback(void *prm, unsigned int len)
{
        printf("here is %s: %s len: %d\n", __func__, (char *)prm, len);
        return OK;
}

int main (void)
{
        sipc_register("App_A_Registered_title", &my_callback, 10)
}
```

popen example

```
        FILE *p;
        size_t sz = 0;
        char buffer[128] = {0};

        p = popen("ifconfig eth0", "r");
        if (!p) {
                errorf("Failed popen");
                return false;
        }

        sz = fread (buffer, sizeof(char), sizeof(buffer) - 1, p);
        if (ferror(p)) {
                errorf("pipe reading error");
                pclose(p);
```

```
        return false;
    }

    buffer[sz] = '\0';

    pclose(p);

    if (strstr(buffer, "MAC")) {
        // do sth
    }
```

fifo example

- a FIFO special file is entered into the file system by calling mkfifo.
- Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

```
const char *annexf_fifo = "/tmp/annexf_fifo";
int fd_fifo;

fd_fifo = open(annexf_fifo, O_RDONLY);
if (fd_fifo == -1) {
    //error
}

unlink(annexf_fifo);
if (mkfifo(annexf_fifo, 0666) == -1) { //could be O_WRONLY | O_APPEND | O_CREAT, 0666
    //Error
}

if (read(fd_fifo, buffer, n) != n) {
    //error
}

close(fd_fifo);
```

IO Streams

- FILE * **stdin** : The standard input stream, which is the normal source of input for the program.
- FILE * **stdout** : The standard output stream, which is used for normal output from the program.
- FILE * **stderr** : The standard error stream, which is used for error messages and diagnostics issued by the program.
- FILE * **fopen** (const char *filename, const char *opentype)
    - ‘r’ Open an existing file for reading only.

    - ‘w’ Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise a new file is created.
    - ‘a’ Open a file for append access; that is, writing at the end of file only. If the file already exists, its initial contents are unchanged and output to the stream is appended to the end of the file. Otherwise, a new, empty file is created.
    - ‘r+’ Open an existing file for both reading and writing. The initial contents of the file are unchanged and the initial file position is at the beginning of the file.
    - ‘w+’ Open a file for both reading and writing. If the file already exists, it is truncated to zero length. Otherwise, a new file is created.
    - ‘a+’ Open or create file for both reading and appending. If the file exists, its initial contents are unchanged. Otherwise, a new file is created. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.
- int **fclose** (FILE *stream) : closing stream
- int **fcloseall** (void): This function causes all open streams of the process to be closed and the connections to corresponding files to be broken. All buffered data is written and any buffered input is discarded.
- Streams on threads
    - void **flockfile** (FILE *stream)
    - int **ftrylockfile** (FILE *stream)
    - void **funlockfile** (FILE *stream)
- int **fputc** (int c, FILE *stream)
- int **fputs** (const char *s, FILE *stream)
- size_t **fwrite** (const void *data, size_t size, size_t count, FILE *stream)
- int **fgetc** (FILE *stream)
- size_t **fread** (void *data, size_t size, size_t count, FILE *stream)
- int **fflush** (FILE *stream)

Low-Level IO

- int **open** (const char *filename, int flags[, mode_t mode])
    - flags
        - O_RDONLY
        - O_WRONLY
        - O_RDWR
    - mode examples
        - 0666
        - 0777
- int **close** (int filedes)
- ssize_t **read** (int filedes, void *buffer, size_t size)
- ssize_t **pread** (int filedes, void *buffer, size_t size, off_t offset)
- ssize_t **write** (int filedes, const void *buffer, size_t size)
- ssize_t **pwrite** (int filedes, const void *buffer, size_t size, off_t offset)
- off_t **lseek** (int filedes, off_t offset, int whence)
    - SEEK_SET
    - SEEK_CUR
    - SEEK_END

- void * **mmap** (void *address, size_t length, int protect, int flags, int filedes, off_t offset)
- int **munmap** (void *addr, size_t length)
- int **msync** (void *address, size_t length, int flags)
- void * **mremap** (void *address, size_t length, size_t new_length, int flag)

Signals

- Error Signals
  - **SIGFPE** : The SIGFPE signal reports a fatal arithmetic error.
  - **SIGILL**: The name of this signal is derived from “illegal instruction”; it usually means your program is trying to execute garbage or a privileged instruction
  - **SIGSEGV** : This signal is generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read.
  - **SIGBUS** : This signal is generated when an invalid pointer is dereferenced
  - **SIGABRT** : This signal indicates an error detected by the program itself and reported by calling abort
  - **SIGTRAP** : Generated by the machine’s breakpoint instruction, and possibly other trap instructions. This signal is used by debuggers
  - **SIGEMT** : Emulator trap; this results from certain unimplemented instructions which might be emulated in software, or the operating system’s failure to properly emulate them
  - **SIGSYS** : Bad system call; that is to say, the instruction to trap to the operating system was executed, but the code number for the system call to perform was invalid
- Termination Signals
  - **SIGTERM** : The SIGTERM signal is a generic signal used to cause program termination. Unlike SIGKILL, this signal can be blocked, handled, and ignored
  - **SIGINT** : The SIGINT (“program interrupt”) signal is sent when the user types the INTR character (normally C-c).
  - SIGQUIT [: The SIGQUIT signal is similar to SIGINT, except that it’s controlled by a different key—the QUIT character, usually C-\—and produces a core dump when it terminates the process, just like a program error signal.
  - **SIGKILL** : The SIGKILL signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal.
  - **SIGHUP** : The SIGHUP (“hang-up”) signal is used to report that the user’s terminal is disconnected, perhaps because a network or telephone connection was broken
- Alarm Signals
  - **SIGALRM** : This signal typically indicates expiration of a timer that measures real or clock time. It is used by the alarm function, for example
  - **SIGVTALRM** : This signal typically indicates expiration of a timer that measures CPU time used by the current process. The name is an abbreviation for “virtual time alarm”.
  - **SIGPROF** : This signal typically indicates expiration of a timer that measures both CPU time used by the current process, and CPU time expended on

behalf of the process by the system. Such a timer is used to implement code profiling facilities, hence the name of this signal.

- Asynchronous I/O Signals
  - **SIGIO** : This signal is sent when a file descriptor is ready to perform input or output.
  - **SIGURG** : This signal is sent when “urgent” or out-of-band data arrives on a socket.
  - **SIGPOLL** : This is a System V signal name, more or less similar to SIGIO. It is defined only for compatibility.
- Job Control Signals
  - **SIGCHLD** : This signal is sent to a parent process whenever one of its child processes terminates or stops.
  - **SIGCONT** : You can send a SIGCONT signal to a process to make it continue. This signal is special—it always makes the process continue if it is stopped, before the signal is delivered
  - **SIGSTOP** : The SIGSTOP signal stops the process. It cannot be handled, ignored, or blocked.
  - **SIGTSTP** : The SIGTSTP signal is an interactive stop signal. Unlike SIGSTOP, this signal can be handled and ignored
  - **SIGTTIN** : A process cannot read from the user’s terminal while it is running as a background job. When any process in a background job tries to read from the terminal, all of the processes in the job are sent a SIGTTIN signal.
  - **SIGTTOU** : This is similar to SIGTTIN, but is generated when a process in a background job attempts to write to the terminal or set its modes.
- int **sigaction** (int signum, const struct sigaction *restrict action, struct sigaction *restrict old-action)
  - The action argument is used to set up a new action for the signal signum, while the old-action argument is used to return information about the action previously associated with this signal
  - **struct sigaction**
    - **sighandler_t sa_handler**
      - This is used in the same way as the action argument to the signal function
    - **sigset_t sa_mask**
      - This specifies a set of signals to be blocked while the handler runs
    - **int sa_flags**
      - This specifies various flags which can affect the behavior of the signal
- int **kill** (pid_t pid, int signum)
  - The kill function sends the signal signum to the process or process group specified by pid.

```
//It retrieves information about the current action for SIGINT without changing that action.

struct sigaction query_action;

if (sigaction (SIGINT, NULL, &query_action) < 0)
  /* sigaction returns -1 in case of error. */
else if (query_action.sa_handler == SIG_DFL)
  /* SIGINT is handled in the default, fatal manner. */
else if (query_action.sa_handler == SIG_IGN)
  /* SIGINT is ignored. */
else
  /* A programmer-defined signal handler is in effect. */
```

```
#include <signal.h>

void
termination_handler (int signum)
{
  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink (p->name);
}

int
main (void)
{
  …
  struct sigaction new_action, old_action;

  /* Set up the structure to specify the new action. */
  new_action.sa_handler = termination_handler;
  sigemptyset (&new_action.sa_mask);
  new_action.sa_flags = 0;

  sigaction (SIGINT, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGINT, &new_action, NULL);
  sigaction (SIGHUP, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGHUP, &new_action, NULL);
  sigaction (SIGTERM, NULL, &old_action);
```

```
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGTERM, &new_action, NULL);
  …
}
```

Timer

- The **alarm** and **setitimer** functions provide a mechanism for a process to interrupt itself in the future. They do this by setting a timer; when the timer expires, the process receives a signal.

- Each process has three independent interval timers available:
    - A real-time timer that counts elapsed time. This timer sends a **SIGALRM** signal to the process when it expires.
    - A virtual timer that counts processor time used by the process. This timer sends a **SIGVTALRM** signal to the process when it expires.
    - A profiling timer that counts both processor time used by the process, and processor time spent in system calls on behalf of the process. This timer sends a **SIGPROF** signal to the process when it expires.

      This timer is useful for profiling in interpreters. The interval timer mechanism does not have the fine granularity necessary for profiling native code.
- You can only have one timer of each kind set at any given time. If you set a timer that has not yet expired, that timer is simply reset to the new value.
- You should establish a handler for the appropriate alarm signal using signal or sigaction before issuing a call to setitimer or alarm
- To be able to use the alarm function to interrupt a system call which might block otherwise indefinitely it is important to not set the SA_RESTART flag when registering the signal handler using sigaction
- int **setitimer** (int which, const struct itimerval *new, struct itimerval *old)
- int **getitimer** (int which, struct itimerval *old)

Periodic Timer Example

```
#include <sys/time.h>          /* for setitimer */
#include <unistd.h>            /* for pause */
#include <signal.h>            /* for signal */

#define INTERVAL 500                /* number of milliseconds to go off */

/* function prototype */
void DoStuff(void);

int main(int argc, char *argv[]) {

  struct itimerval it_val;          /* for setting itimer */
```

```
  /* Upon SIGALRM, call DoStuff().
   * Set interval timer.  We want frequency in ms,
   * but the setitimer call needs seconds and useconds. */
  if (signal(SIGALRM, (void (*)(int)) DoStuff) == SIG_ERR) {
    perror("Unable to catch SIGALRM");
    exit(1);
  }
  it_val.it_value.tv_sec =     INTERVAL/1000;
  it_val.it_value.tv_usec =    (INTERVAL*1000) % 1000000;
  it_val.it_interval = it_val.it_value;
  if (setitimer(ITIMER_REAL, &it_val, NULL) == -1) {
    perror("error calling setitimer()");
    exit(1);
  }

  //do your other stuff

}

/*
 * DoStuff
 */
void DoStuff(void) {

  printf("Timer went off.\n");
}
```

OneShot Timer Example

```
#include <sys/time.h>          /* for setitimer */
#include <unistd.h>            /* for pause */
#include <signal.h>            /* for signal */

#define INTERVAL 500                /* number of milliseconds to go off */

/* function prototype */
void DoStuff(void);

int main(int argc, char *argv[]) {
  if (signal(SIGALRM, DoStuff) == SIG_ERR) {
    perror("Unable to catch SIGALRM");
    exit(1);
  }
  alarm(10); //set oneshot alarm to 10 sec later

  //do your other stuff
```

```
}

/*
 * DoStuff
 */
void DoStuff(int sig) {

        if (sig == SIGALRM) {
                //timer expired
        }

}
```

---

strtoul and strtol

```
int base;
        char *endptr, *str;
        long val;

        if (argc < 2) {
            fprintf(stderr, "Usage: %s str [base]\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        str = argv[1];
        base = (argc > 2) ? atoi(argv[2]) : 0;

        errno = 0;    /* To distinguish success/failure after call */
        val = strtol(str, &endptr, base);

        /* Check for various possible errors. */

        if (errno != 0) {
            perror("strtol");
            exit(EXIT_FAILURE);
        }

        if (endptr == str) {
            fprintf(stderr, "No digits were found\n");
            exit(EXIT_FAILURE);
        }

        /* If we got here, strtol() successfully parsed a number. */
```

```
        printf("strtol() returned %ld\n", val);

        if (*endptr != '\0')        /* Not necessarily an error... */
            printf("Further characters after number: \"%s\"\n", endptr);

        exit(EXIT_SUCCESS);
```

---

output

```
$ ./a.out 123
        strtol() returned 123
        $ ./a.out '    123'
        strtol() returned 123
        $ ./a.out 123abc
        strtol() returned 123
        Further characters after number: "abc"
        $ ./a.out 123abc 55
        strtol: Invalid argument
        $ ./a.out ''
        No digits were found
        $ ./a.out 4000000000
        strtol: Numerical result out of range
```

strtok_r

```
char *str1, *str2, *token, *subtoken;
    char *saveptr1, *saveptr2;
    int j;

   if (argc != 4) {
        fprintf(stderr, "Usage: %s string delim subdelim\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

   for (j = 1, str1 = argv[1]; ; j++, str1 = NULL) {
        token = strtok_r(str1, argv[2], &saveptr1);
        if (token == NULL)
            break;
        printf("%d: %s\n", j, token);

       for (str2 = token; ; str2 = NULL) {
            subtoken = strtok_r(str2, argv[3], &saveptr2);
            if (subtoken == NULL)
                break;
            printf(" --> %s\n", subtoken);
        }
```

```
    }

    exit(EXIT_SUCCESS);
```

---

output

```
$ ./a.out 'a/bbb///cc;xxx:yyy:' ':;' '/'
1: a/bbb///cc
        --> a
        --> bbb
        --> cc
2: xxx
        --> xxx
3: yyy
        --> yyy
```

Format Specifiers

Format Specifier	Type
%c	Character
%d	Signed integer
%e or %E	Scientific notation of floats
%f	Float values
%g or %G	Similar as %e or %E
%hi	Signed integer (short)
%hu	Unsigned Integer (short)
%i	Unsigned integer
%l or %ld or %li	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or unsigned long
%lli or %lld	Long long

%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x or %X	Hexadecimal representation
%n	Prints nothing
%%	Prints % character

- printf("%8d", x); → add space before it
- printf("%08d", x); → add zero before it
- printf("%.2f", x); → specifies precision

CLEAN CODE

Meaningful Names

- Use Intention-Revealing Names
    - int elapsedTimeInDays
- Avoid Disinformation
    - do not use accountList if the container is not a List
    - use accounts or bunchOfAccunts
- Make Meaningful Distinctions
    - imagine classes *Product* and *ProductInfo*. Distinction is not clear
- Use Pronounceable Names
    - use *generationTimestamp* instead of *genymdhms*
- Use Searchable Names
    - single-letter names and numeric constants are problematic
- Avoid Encodings
    - it is hard to change a variable type because you need to change encoding or prefix
- Avoid Mental Mapping
    - similar to single-letter variable, eg: *r* for *url*
- Class Names
    - sould be noun or nun phrase like *Account* or *AddressParser*, should not be a verb
- Method Names
    - should be verb or ver phrase like *postPayment* or *deletePage*
- Pick One Word per Concept
    - confusing to have *fetch*, *retrive* and *get* in same scope
- Do not Pun
    - avoid using same word for two different purposes
- Use Solution Domain Names
    - use algorithm names, pattern names if needed
- Use Problem Domain Names
- Add Meaninful Context
- Do not Add Gratuitous Context

Error Handling

- Use Exceptions Rather Than Return Codes
- Write *try/catch* Statement First
- Use Unchecked Exceptions
- Do not Return NULL
- Do not Pass NULL

Functions

- Small
  - a function should be no longer than a screen-ful
- Blocks and Indenting
  - one block should be one size long and indent level of a function should not be greater than one or two
- Do One Thing
  - do one thing, do it well, do it only
- Sections within Functions
  - functions that do one thing cannot be reasonably divided into sections
- One Level of Abstraction per Function
  - in order to make sure the function is doing *one thing*, we need to make sure that the statemens withing our function are all at the same level ofastraction
- Reading Code from Top to Bottom: *The Stepdown Ruke*
- Swithc Statement
  - use them in single block or function. use polymorphism for switch statement in order not to repeat it again and again
- Use Descriptive Names
  - a long descriptive name is better than a short enigmatic name
- Function Arguments
  - more than three args function should not beused anywhere
- Common Monadic Forms
  - sigle argument functions. reason to use this kind of functions are;
    - asking question about the arument
    - operating on that argument
    - *event* purpose
- Falg Argument
  - ugly practice. eg: *render(bool isSuite)
  - instead, use *render()* and *renderForSuite*
- Dyadic Functions
  - hard to understand than monodic functions
- Triads
  - functions that take three arguments
- Argument Objects
  - if necessary to pass more than one argument, then use class of structure
- Verbs and Keywords
  - choice nice verb/noun pair. eg: use *writeField(name)* instead *write(name)*
- Have No Side Effect
  - the fucntion should not do anything then we cannot inference from its name
- Output Arguments
  - in general, output arguments should be avoided (it is not in the case in C language I believe)
- Command Query Seperation
  - functions should either do something or answer something
- Prefer Exceptions to Return Error Codes
- Extract Try/Catch Blocks
  - Do not Repeat Yourself

Formatting

Vertical Formatting

- Vertical Openness Between Concepts
    - each line represents an expression or
    - a clause, and each group of lines represents a complete thought
- Vertical Density
    - lines of code that are tightly related should appear vertically dense
- Vertical Distance
    - concepts that are closely related should be kept vertically close to each other
- Variable Declaration
    - variables should be declared as close to their usage as possible
- Dependent Functions
    - they should be vertically close
- Conceptual Affinity
    - all destroyer functions should be close
- Vertical Ordering
    - a function that is called should be below a function that does the calling (or vice versa)

Hotizontal Formatting

- Horizontal Alignment
    - unaligned
    - declarations and assignments, because they point out an important deficiency
- Indentation
    - avoid collapsing scopes down to one line
    - eg: *public String render() throws Exception {return ""; }*

Comments

- Comments Do Not Make Up for Bad Code
- Explain Yourself in Code

Good Comments

- Legal Comments
    - Copyright and License
- Informative Comments
    - it is sometimes useful to provide basic information with a comment
- Explenation of Intent
    - sometimes a comment goes beyond just useful information about the implementation and provides the intent ehind a decision
- Clarification
    - sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that is readable
    - eg: *assertTrue(a.compareTo(a) == 0)* //a == a

- Warning of Consequences
    - eg: *//do not run unless you will face crash*
- TODO Comments
- Amplification
    - a comment may be used to amplify the importance of something
    - eg: *//this trim is really important because ..*

Bad Comments

- Mumbling
    - if you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write
    - Redundant Comments
    - Misleading Comments
    - Mandated Comments
        - it is just plain silly to have a rule that says that every function must have a javadoc
    - Sometimes people add a comment to the start of a module every time they edit it
    - Noise Comment
        - sometimes you see comments that are nothing but noise
    - Position Marker
        - eg: *// Actions ////////////////////////////////////
    - Closing Brace Comments
        - instead, try to shorten your function
    - Attributions and Bylines
        - eg: * // added by Rick // *
    - Commented-Out Code
    - Nonlocal Information
        - if you must write a comment, then make sure it describes the code it appears near. don’t
        - offer systemwide information in the context of a local comment
    - Too Much Information
    - Inobvious Connection
        - array that is big enough to hold all the pixels (plus filter bytes)
        - what is *filter bytes*
    - Function Headers
        - short functions don’t need much description. a well-chosen name for a small function that
        - does one thing is usually better than a comment header.

C CODING STANDARTS (custom)

- Always use braces including 1-line-if-statement
- Always use paranthesis in complex expressions or macro definitions
- Line length limit should be max 80 characters
- Use block comments
    - for every file, at the beginning
    - before every function in header
    - before every static function in source code
- for short commands, use tab to separate it from code statement. if multiple commands should be used, then align them
- use rvalue at left in if conditions (if (3 == i) eg)
- keep assignments outside of the if statement
- perform explicit test to determine success (use if(foo() != 0) instead of if (foo()))
- use as less vertical distance as possible for stronger affinity
- set errno to zero before calling C standart library functions
- use tab or 8 chars empty space for indention
- Within a switch statement, case statements should not be indented, however, their contents should be intented
- successive define constants should be alligned
- successive assignments should be alligned
- Pointer alignment should be always right
- Each concept should be separated by blank lines. Those are functions, declarations, definitions and statements (if-else, loops, switch, return).
- operators should be on the new line in long statement
    - eg:
      if (!aa && bb
      || cc)
- Local variables should be at the top of each function
- When declaring variables in structures, declare them organized by use in a manner to attempt to minimize memory wastage because of compiler alignment issues,
- Each variable declaration should have its own line.
- Avoid any local variable declarations that override declarations at higher levels.
- All extern variables defined in a module, but that referenced in other modules, must be declared in a separate header file
- When a function takes no argument, it is better to explicitly specify void parameter
- All private functions should be defined static
- All functions should have prototypes
- All variables should be initialized before its use.
- Initilization should be placed as close as possible to the code that uses them
- Global variables are initialized by the compiled code at zeros. These variables can be used without explicitly initializing them.
- More than one statements in a single line is not allowed
- use for(;;) for infinite loop

- There should be a break for each case and one default within a switch and it should be indented to align with the associated case, rather than with the contents of the case code block. By aligning the case and the break keywords it is possible to spot missing breaks
    - eg
      switch(err) {
      case ERR A:
      statement A;
      break;
      case ERR B:
      statement B;
      break;
      default:
      ...
      break;
      }

      The last break is redundant, but it prevents a fall-through error if another case is added the last one
- ” is” prefix can be added to indicate which value is true or false.
- Meaningful names can be useful in terms of consistency throughout the program
- Avoid abbreviations that form letter combinations that may suggest unintended meanings. For example, the name “inch” is a misleading abbreviation for “input character.” The name “in char” would be better
- Identifiers in mutually visible scopes must be named carefully to prevent confusion
- The names of all public functions should be prefixed with their module name and an underscore
- No variable name should have a same name with a variable name from the C standart library such as errno
- No variable name should be longer than 31 characters
- In header files, this is the organization
    - At the beginning, block comment describing the contents of the file for readers ant the Doxygen documentation
    - ifndef: file guard mechanism to prevent multiple inclusion
    - Header include files
    - Constant definitions,
    - Macros,
    - Type definitions,
    - Struct definitions,
    - Global variable declarations with extern keyword
    - Functions protoypes (with the above block comments for Doxygen) and lastly,
    - Closing match to the inclusion guar
- In source codes
    - Beginning Comment
    - Header include files
    - Constant Definitions,
    - Macros,
    - Private Data Types

        - Type Definitions
        - Struct Definitions
    - Declarations
        - Global Data Declarations
        - Private Data Declarations
    - Static Function Prototypes
    - Function Definitions
        - Global Function Definitions
        - Static Function Definitions

GNU C CODING STANDARTS

- keep the length of source lines to **79** characters or less, for maximum readability in the widest range of environments.
- function definitions to start the name of the function in column one

Formatting

```
static char *
concat (char *s1, char *s2)
{
  …
}
```

- if the arguments don’t fit nicely on one line, split it like this

```
int
lots_of_args (int an_integer, long a_long, short a_short,
              double a_double, float a_float)
…
```

- For struct and enum types, likewise put the braces in column one, unless the whole contents fits on one line:

```
struct foo
{
  int a, b;
}
or
struct foo { int a, b; }
```

- When you split an expression into multiple lines, split it before an operator, not after one. Here is the right way:

```
if (foo_this_is_long && bar > win (x, y, z)
    && remaining_condition)
```

- Try to avoid having two operators of different precedence at the same level of indentation. For example, don’t write this:

```
mode = (inmode[j] == VOIDmode
        || GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])
        ? outmode[j] : inmode[j]);
```

Commenting

- Every program should start with a comment saying briefly what it is for. Example: ‘fmt - filter for simple filling of text’. This comment should be at the top of the source file containing the ‘main’ function of the program.
- Also, please write a brief comment at the start of each source file, with the file name and a line or two about the overall purpose of the file.
- Please put a comment on each function saying what the function does, what sorts of arguments it gets, and what the possible values of arguments mean and are used for.
- Also explain the significance of the return value, if there is one.
- Every ‘#endif’ should have a comment, except in the case of short conditionals (just a few lines) that are not nested.

Clean Use of C Constructs

- Please explicitly declare the types of all objects.
- Declarations of external functions and functions to appear later in the source file should all go in one place near the beginning of the file (somewhere before the first function definition in the file), or else should go in a header file. Don’t put extern declarations inside functions.
- It used to be common practice to use the same local variables (with names like tem) over and over for different values within one function. Instead of doing this, it is better to declare a separate local variable for each distinct purpose, and give it a name which is meaningful.
- Don’t use local variables or parameters that shadow global identifiers. GCC’s ‘-Wshadow’ option can detect this problem.
- Don’t declare multiple variables in one declaration that spans lines. Start a new declaration on each line, instead. write either this:

```
int foo, bar;
```

or this

```
int foo;
int bar;
```

- If the above variables are global variables, each should have a comment preceding it anyway.
- Try to avoid assignments inside if-conditions (assignments inside while-conditions are ok). For example, don’t write this:

```
if ((foo = (char *) malloc (sizeof *foo)) == NULL)
  fatal ("virtual memory exhausted");
```

Naming Variables, Functions, and Files

- The names of global variables and functions in a program serve as comments of a sort. So don't choose terse names—instead, look for names that give useful information about the meaning of the variable or function.
- Local variable names can be shorter, because they are used only within one context, where (presumably) comments explain their purpose.
- Try to limit your use of abbreviations in symbol names. It is ok to make a few abbreviations, explain what they mean, and then use them frequently, but don't use lots of obscure abbreviations.
- Please use underscores to separate words in a name
- When you want to define names with constant integer values, use enum rather than '#define'. GDB knows about enumeration constants.
- Some GNU programs were designed to limit themselves to file names of 14 characters or less, to avoid file name conflicts if they are read into older System V systems.

Portability between CPUs

- We don't support 16-bit machines in GNU.
- You need not cater to the possibility that long will be smaller than pointers and size_t. We know of one such platform: 64-bit programs on Microsoft Windows. If you care about making your package run on Windows using Mingw64, you would need to deal with 8-byte pointers and 4-byte long,
- Don't assume that the address of an int object is also the address of its least-significant byte. This is false on big-endian machines. Thus, don't make the following mistake:

Do This

```
int c;
while ((c = getchar ()) != EOF)
  {
    unsigned char u = c;
    write (file_descriptor, &u, 1);
  }
```

instead of this

```
int c;
…
while ((c = getchar ()) != EOF)
  write (file_descriptor, &c, 1);
```

- Avoid casting pointers to integers if you can. Such casts greatly reduce portability, and in most programs they are easy to avoid.

MAKE

- The make program is intended to automate the mundane aspects of transforming source code into an executable.
- The advantages of make over scripts is that you can specify the relationships between the elements of your program to make, and it knows through these relationships and timestamps exactly what steps need to be redone to produce the desired program each time.
- Using this information, make can also optimize the build process avoiding unnecessary steps

makefile example 1

```
hello: hello.c
    gcc hello.c -o hello
```

- If a target is included as a c ommand-line argument, that target is updated. If no command-line targets are given, then the first target in the file is used, called the default goal.
- Modifying any of the source files and reinvoking make will cause some, but usually not all, of these commands to be repeated so the source code changes are properly incorporated into the executable.
- make can perform the minimum amount of work necessary to update the executable

Basic Topics

Targets and Prerequisites

- Essentially a makefile contains a set of rules used to build an application. The first rule seen by make is used as the default rule. A rule consists of three parts: the target, its prerequisites, and the command(s) to perform:

```
target: prereq1 prereq2
        commands
```

example: Here is a rule for compiling a C file, foo.c, into an object file, foo.o:

```
foo.o: foo.c foo.h
    gcc -c foo.c
```

- If any of the prerequisites has an associated rule, make attempts to update those first.

example

```
count_words: count_words.o lexer.o -lfl
    gcc count_words.o lexer.o -lfl -o count_words
```

```
count_words.o: count_words.c
	gcc -c count_words.c

lexer.o: lexer.c
	gcc -c lexer.c

lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```

output

```
gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -o count_words
```

- As you look at the makefile and sample execution, you may notice that the order in which commands are executed by make are nearly the opposite to the order they occur in the makefile. This top-down style is common in makefiles. Usually the most general form of target is specified first in the makefile and the details are left for later.
- **-c** is used for converting source file to object file
- **-l<NAME>** links the lib<NAME>.so that should be in the directory of make searches
- If you specify a target that is not in the makefile and for which there is no implicit rule (discussed in Chapter 2), make will respond with:

  ```
  $make non-existent-target
  make: *** No rule to make target `non-existent-target'.  Stop.
  ```

- **—just-print** (or **-n**) which tells make to display the commands it would execute for a particular target without actually executing them
- Each command must begin with a **tab** character. This (obscure) syntax tells make that the characters that follow the tab are to be passed to a subshell for execution
- The **comment** character for make is the hash or pound sign, . All text from the pound sign to the end of line is ignored.
- **Long lines** can be continued using the standard Unix escape character backslash **(\)**.

Explicit Rules

- A rule can have more than one target. This means that each target has the same set of prerequisites as the others. If the targets are out of date, the same set of actions will be performed to update each one. For instance:

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
```

- a variable is either a dollar sign followed by a single character or a dollar sign followed by a word in parentheses → make variables

```
# Compile vpath.c with special flags.
vpath.o: vpath.c
	$(COMPILE.c) $(RULE_FLAGS) $(OUTPUT_OPTION) $<
```

- **wildcards**
  - *.* → expands all the files containing a period
  - ? → represents andy single character
  - [...] → represents a character class
  - [^...] → represents opposite of character class
  - ~ → represents current user's home directory

example

```
prog: *.c
	$(CC) -o $@ $^
```

Phony Targets

- Targets that do not represent files are known as phony targets

example

```
clean:
	rm -f *.o lexer.c
```

- Since most phony targets do not have prerequisites, the clean target would always be considered up to date and would never execute.
- To avoid this problem, GNU make includes a special target, .PHONY, to tell make that a target is not a real file.
- more or less standard phony targets are like below

Target	**Function**
all	Perform all tasks to build the application
install	Create an installation of the application from the compiled binaries
clean	Delete the binary files generated from sources

distclean	Delete all the generated files that were not in the original source distribution
TAGS	Create a tags table for use by editors
info	Create GNU info files from their Texinfo sources
check	Run any tests associated with this application

example
```
.PHONY: clean
clean:
	rm -f *.o lexer.c
```

- Phony targets can also be used to improve the “user interface” of a makefile. Often targets are complex strings containing directory path elements, additional filename components (such as version numbers) and standard suffixes.

Variables

- The simplest ones have the syntax:

$(variable-name)

- In general, a variable name must be surrounded by $( ) or ${ } to be recognized by make. As a special case, a single character variable name does not require the parentheses.
- A makefile will typically define many variables, but there are also many special variables defined automatically by make. Some can be set by the user to control make’s behavior while others are set by make to communicate with the user’s makefile.
- Automatic variables are set by make after a rule is matched. They provide access to elements from the target and prerequisite lists so you don’t have to explicitly specify any filenames.
- There are seven “core” automatic variables:
  - **$@** The filename representing the target.
  - **$%** The filename element of an archive member specification.
  - **$<** The filename of the first prerequisite.
  - **$?** The names of all prerequisites that are newer than the target, separated by spaces.

    - **$^** The filenames of all the prerequisites, separated by spaces. This list has duplicate filenames removed since for most uses, such as compiling, copying, etc., duplicates are not wanted.
    - **$+** Similar to $^, this is the names of all the prerequisites separated by spaces, except that $+ includes duplicates. This variable was created for specific situations such as arguments to linkers where duplicate values have meaning.
    - **$*** The stem of the target filename. A stem is typically a filename without its suffix. (We'll discuss how stems are computed later in Section 2.4.) Its use outside of pattern rules is discouraged.
- In addition, each of the above variables has two variants for compatibility with other makes.
    - One variant returns only the directory portion of the value. This is indicated by appending a "D" to the symbol, $(@D), $(<D), etc
    - The other variant returns only the file portion of the value. This is indicated by appending an F to the symbol, $(@F), $(<F), etc
- the variables are only available in the command script of a rule

example

```
count_words: count_words.o counter.o lexer.o -lfl
	gcc $^ -o $@

count_words.o: count_words.c
	gcc -c $<

counter.o: counter.c
	gcc -c $<

lexer.o: lexer.c
	gcc -c $<

lexer.c: lexer.l
	flex -t $< > $@
```

Variable Types and Assignments

- There are two types of variables in make: simply expanded variables and recursively expanded variables. A simply expanded variable (or a simple variable) is defined using the **:=**
- The second type of variable is called a recursively expanded variable. A recursively expanded variable (or a recursive variable) is defined using the **=** assignment operator. It is called "recursively expanded" because its righthand side is simply slurped up by make and stored as the value of the variable without evaluating or expanding it in any way
- The **?=** operator is called the conditional variable assignment operator. That's quite a mouth-full so we'll just call it conditional assignment. This operator will perform the requested variable assignment only if the variable does not yet have a value.

```
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

- Here we set the output directory variable, OUTPUT_DIR, **only if** it hasn't been set earlier.
- **+=** , is usually referred to as append.

MACROS

- The define directive is followed by the macro name and a newline. The body of the macro includes all the text up to the endef keyword, which must appear on a line by itself. A macro created with define is expanded pretty much like any other variable, except that when it is used in the context of a command script, each line of the macro has a tab prepended to the line

```
define create-jar
 @echo Creating $@...
 $(RM) $(TMP_JAR_DIR)
 $(MKDIR) $(TMP_JAR_DIR)
 $(CP) -r $^ $(TMP_JAR_DIR)
 cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
 $(JAR) -ufm $@ $(MANIFEST)
 $(RM) $(TMP_JAR_DIR)
endef

$(UI_JAR): $(UI_CLASSES)
      $(create-jar)
```

Where Variables Come From

1. File
    a. a file could be included
2. Command Line
    a. while calling make
3. environment
    a. export is used
4. automatic
    a. hidden make rules

VPATH

- The value of the make variable **VPATH** specifies a list of directories that make should search. Most often, the directories are expected to contain prerequisite files that are not in the current directory; however, make uses VPATH as a search list for both prerequisites and targets of rules.

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
      cc -c $(CFLAGS) $< -o $@
```

- If a file of the same name exists in multiple places in the VPATH list, make grabs the first one. Sometimes this can be a problem. The **vpath** directive is a more precise way to achieve our goals.

vpath %.c src
vpath %.l src
vpath %.h include

- **-I** → include option to search header files. !!! what is the difference between -I and vpath???
- -I option can be added to the compiler command or **CPPFLAGS** flag can be edited like CPPFLAGS = -I include

Pattern Rules

- Many programs that read one file type and output another conform to standard conventions. For instance, all C compilers assume that files that have a .c suffix contain C source code and that the object filename can be derived by replacing the .c suffix with .o (or .obj for some Windows compilers).
- So, we can write this;

%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<

- there is a special rule to generate a file with no suffix (always an executable) from a .c file:

%: %.o
    $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@

NOTE: The include directive should always be placed after the hand-written dependencies so that the default goal is not hijacked by some dependency file.

Managing Libraries

- An archive library, usually called simply a library or archive, is a special type of file containing other files called members. Archives are used to group related object files into more manageable units

example usage → **ar rv** libcounter.a counter.o lexer.o

- The options rv indicate that we want to replace members of the archive with the object files listed and that ar should verbosely echo its actions. We can use the replace option even though the archive doesn't exist. The first argument after the

options is the archive name followed by a list of object files. (Some versions of ar also require the “c” option, for create, if the archive does not exist but GNU ar does not.)

- both add and replace options are handled by “ar” command

Using Libraries as Prerequisites

- When the **-l** syntax is used, the prerequisites aren’t proper files at all:

```
xpong: $(OBJECTS) -lX11 -lXaw
    $(LINK) $^ -o $@
```

- if the library is not under the search path, then an environment should be edited before using it in the same session;

export LD_LIBRARY_PATH=<full-library-path>

Conditional Preprocessing

example

```
ifdef COMSPEC
  PATH_SEP := ;
  EXE_EXT  := .exe
else
  PATH_SEP := :
  EXE_EXT  :=
endif
```

example

```
ifeq (a, a)
  # These are equal
endif

ifneq ( b, b )
  # These are not equal - ' b' != 'b '
endif
```

Built-in Functions

- $(**filter** pattern . . . ,text)

example

```
words := he the hen other the%
get-the:
    @echo he matches: $(filter he, $(words))
    @echo %he matches: $(filter %he, $(words))
```

```
	@echo he% matches: $(filter he%, $(words))
	@echo %he% matches: $(filter %he%, $(words))
```

output

```
$ make
he matches: he
%he matches: he the
he% matches: he hen
%he% matches: the%
```

- $(**findstring** string,text)

example

```
find-tree:
	# PWD = $(PWD)
	# $(findstring /test/book/admin,$(PWD))
	# $(findstring /test/book/bin,$(PWD))
	# $(findstring /test/book/dblite_0.5,$(PWD))
	# $(findstring /test/book/examples,$(PWD))
	# $(findstring /test/book/out,$(PWD))
	# $(findstring /test/book/text,$(PWD))
```

output

```
$make
# PWD = /test/book/out/ch03-findstring-1
#
#
#
#
# /test/book/out
#
```

- $(**subst** search-string,replace-string,text)

example

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c, .o, $(sources))
```

(notice the space after each comma), the value of $(objects) would have been:

```
count_words .o counter .o lexer .o
```

- $(**words** text)
  - This returns the number of words in text.
- $(**word** n,text)
  - This returns the n th word in text
- $(**firstword** text)
  - This returns the first word in text. This is equivalent to $(word 1, text).
- $(**wordlist** start,end,text)
  - This returns the words in text from start to end, inclusive

- $(**sort** list)
    - The sort function sorts its list argument and removes duplicates. The resulting list contains all the unique words in lexicographic order, each separated by a single space. In addition, sort strips leading and trailing blanks.

example
```
$ make -f- <<< 'x:;@echo =$(sort    d  b s   d      t   )='
```

output
```
=b d s t=
```

- $(**shell** command)
    - The shell function accepts a single argument that is expanded (like all arguments) and passed to a subshell for execution

example
```
stdout := $(shell echo normal message)
stderr := $(shell echo error message 1>&2)
shell-value:
    # $(stdout)
    # $(stderr)
```

- $(**wildcard** pattern . . .) → example: sources := $(wildcard *.c *.h)
- $(**dir** list . . .)
    - The dir function returns the directory portion of each word in list.
- $(**notdir** name . . .)
    - The notdir function returns the filename portion of a file path
- $(**suffix** name . . .)
    - The suffix function returns the suffix of each word in its argument.
- $(**basename** name . . .)
    - The basename function is the complement of suffix. It returns the filename without its suffix
- $(**addsuffix** suffix,name . . .)
    - The addsuffix function appends the given suffix text to each word in name
- $(**addprefix** prefix,name . . .)
    - The addprefix function is the complement of addsuffix
- $(**join** prefix-list,suffix-list)
    - The join function is the complement of dir and notdir. It accepts two lists and concatenates the first element from prefix-list with the first element from suffix-list, then the second element from prefix-list with the second element from suffix-list and so on
- $(**error** text)
    - The error function is used for printing fatal error messages. After the function prints its message, make terminates with an exit status of 2
- $(**warning** text)
    - The warning function is similar to the error function except that it does not cause make to exit
- $(**foreach** variable,list,body)

example
```
VARIABLE_LIST := SOURCES OBJECTS HOME
$(foreach i,$(VARIABLE_LIST), \
  $(if $($i),,              \
    $(shell echo $i has no value > /dev/stderr)))
```

- $(**strip** text)
  - The strip function removes all leading and trailing whitespace from text and replaces all internal whitespace with a single space.
- $(**origin** variable)
  - The origin function returns a string describing the origin of a variable.
  - The possible return values of origin are:
    - undefined
    - default
    - environment
    - environment override
    - file
    - command line
    - override
    - automatic

Advanced Topics

Recursive Make

- The motivation behind recursive make is simple: make works very well within a single directory (or small set of directories) but becomes more complex when the number of directories grows. So, we can use make to build a large project by writing a simple, self-contained makefile for each directory, then executing them all individually.
- We could use a scripting tool to perform this execution, but it is more effective to use make itself since there are also dependencies involved at the higher level.

example
```
lib_codec := lib/codec
lib_db    := lib/db
lib_ui    := lib/ui
libraries := $(lib_ui) $(lib_db) $(lib_codec)
player    := app/player

.PHONY: all $(player) $(libraries)
all: $(player)
for d in $(player) $(libraries); \
       do                        \
         $(MAKE) --directory=$$d;       \
       done

$(player) $(libraries):
```

```
    $(MAKE) --directory=$@

$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

Command-Line Options

- When make starts, it automatically appends most command-line options to **MAKEFLAGS**
- The only exceptions are the options
    - **—directory (-C)**
    - —file (-f)
    - —old-file (-o)
    - —new-file (-W)

Portable Makefiles

- here are some common portability problems that every makefile must deal with sooner or later
    - Program names
    - Paths
    - Options
    - Shell features
    - Program behavior
    - Operating system

Parallel make

- —jobs =2 (or -j 2)

The eval Function

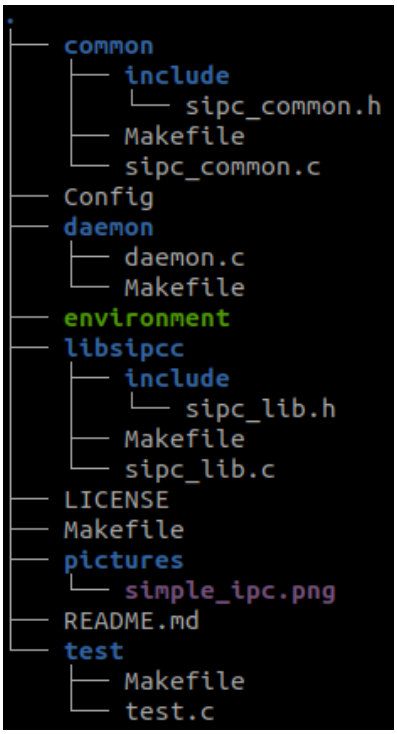
The eval function is very special: it allows you to define new makefile constructs that are not constant; which are the result of evaluating other variables and functions. The argument to the eval function is expanded, then the results of that expansion are parsed as makefile syntax. The expanded results can define new make variables, targets, implicit or explicit rules, etc.

MAKE Debugging

- Use **warning** function inside the Makefile
- Command-Line Options
    - —just-print (-n)
        - This causes make to read the makefile and print every command it would normally execute to update the target but without executing them.
    - —print-data-base

        - It will dump its internal database.
    - —warn-undefined-variables
        - This option causes make to display a warning whenever an undefined variable is expanded
    - —debug option.
        - This provides the most detailed information available other than by running a debugger. There are five debugging options and one modifier: **basic, verbose, implicit, jobs, all**, and **makefile**, respectively.

Make Detailed Example



→ **cat Config**

```
COMMON_PATH=common
LIBSIPCC_PATH=libsipcc
LIBRARY_NAME=sipcc
OPEN_DEBUG=y
```

→ **cat environment**

```
#!/bin/bash
source ./Config

OUTPUT=$(env | grep LD_LIBRARY_PATH | cut -d ";" -f 2)
PWD=$(pwd)

VARIABLE_PART=`echo $OUTPUT | cut -d "=" -f 2`
LEN=`echo $VARIABLE_PART | awk '{print length}'`
```

```
if [ "$LEN" -eq "0" ]; then
   export LD_LIBRARY_PATH=$PWD/$LIBSIPCC_PATH
else
    export LD_LIBRARY_PATH=$PWD/$LIBSIPCC_PATH:$VARIABLE_PART
fi
```

→ **cat Makefile**

```
include Config

ifeq (${OPEN_DEBUG},y)
CFLAGS += -DOPEN_DEBUG
endif

CC = gcc
RM = rm -rf
CFLAGS = -Wall -Wextra -Werror -g3 -O2 -fPIC
LDFLAGS = -lpthread

COMMON_INCDIR="$(shell echo ${PWD}/${COMMON_PATH}/include)"
LIB_INCDIR="$(shell echo ${PWD}/${LIBSIPCC_PATH}/include)"
LIB_DIR="$(shell echo ${PWD}/${LIBSIPCC_PATH})"
SO_NAME=lib${LIBRARY_NAME}.so
LIB_NAME=${LIBRARY_NAME}

SUBDIRS = common \
          libsipcc \
          daemon \
          test

.PHONY: all clean

all:
          echo "_-_-_-_- make start _-_-_-_-"
          for dir in $(SUBDIRS); do \
                    make -C $$dir all; \
          done
          echo "_-_-_-_- make end _-_-_-_-"

clean:
          echo "_-_-_-_- clean start _-_-_-_-"
          for dir in $(SUBDIRS); do \
                    make -C $$dir clean; \
          done
          echo "_-_-_-_- clean end _-_-_-_-"

.EXPORT_ALL_VARIABLES:
```

→ **cat common/Makefile**
```
COMMON_INCDIR=./include

C_SRCS = \
sipc_common.c

OBJS += \
./sipc_common.o

.PHONY: all clean

all:
	$(CC) $(CFLAGS) $(LDFLAGS) -I$(COMMON_INCDIR) -c $(C_SRCS)

clean:
	$(RM) $(OBJS)
```

→ **cat daemon/Makefile**
```
EXECUTABLE_NAME=sipcd

CFLAGS += -Wno-stringop-truncation

C_SRCS = \
daemon.c \
../common/sipc_common.o

OBJS += \
./daemon.o

.PHONY: all clean

all:
	$(CC) -o ./$(EXECUTABLE_NAME) $(C_SRCS) $(CFLAGS) $(LDFLAGS)
-I$(COMMON_INCDIR)

clean:
	$(RM) $(OBJS) ./$(EXECUTABLE_NAME)
```

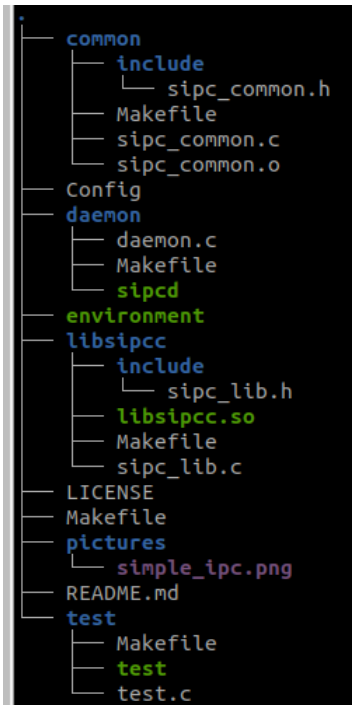
→ cat libsipcc/Makefile
```
CFLAGS += -Wno-varargs

C_SRCS = \
sipc_lib.c \
../common/sipc_common.o
```

```
LIBSIPCC_INCDIR=-I ./include -I ../common/include

OBJS += \
./sipc_lib.o

.PHONY: all clean

all:
	$(CC) $(C_SRCS) $(CFLAGS) $(LDFLAGS) $(LIBSIPCC_INCDIR) -o
./$(SO_NAME) -shared

clean:
	$(RM) $(OBJS) ./$(SO_NAME)
```

→ **cat test/Makefile**

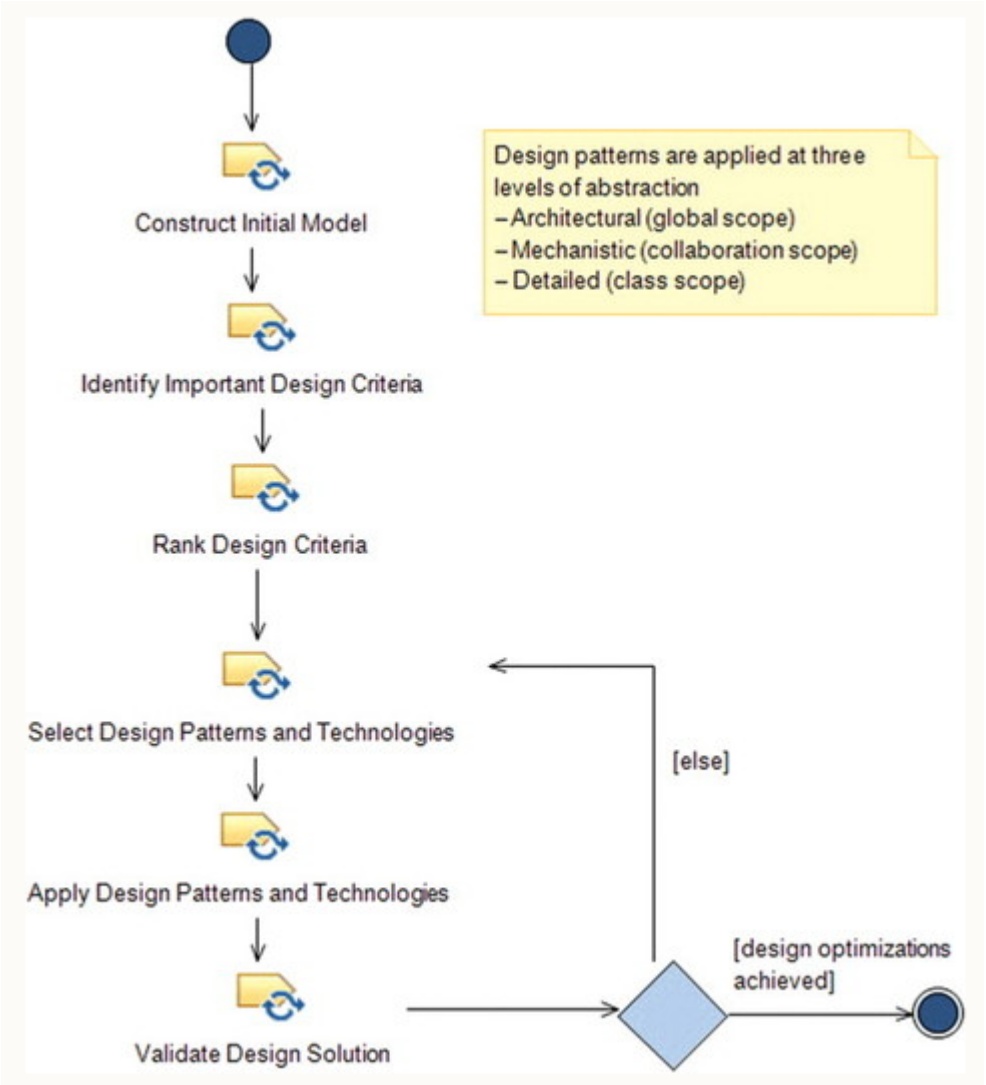
```
TEST_EXECUTABLE_NAME=test

LDFLAGS += -l${LIB_NAME}

TEST_INCDIR=-I ${LIB_INCDIR} -I ${COMMON_INCDIR}
TEST_LIBDIR=-L ${LIB_DIR}

C_SRCS = \
test.c

OBJS += \
./test.o

.PHONY: all clean

all:
	$(CC) -o ./$(TEST_EXECUTABLE_NAME) $(C_SRCS) $(CFLAGS) $(LDFLAGS)
$(TEST_LIBDIR) $(TEST_INCDIR)

clean:
	$(RM) $(OBJS) ./$(TEST_EXECUTABLE_NAME)
```

result


.
common
include
sipc_common.h
Makefile
sipc_common.c
sipc_common.o
Config
daemon
daemon.c
Makefile
sipcd
environment
libsipcc
include
sipc_lib.h
libsipcc.so
Makefile
sipc_lib.c
LICENSE
Makefile
pictures
simple_ipc.png
README.md
test
Makefile
test
test.c

DESIGN PATTERNS

Basic Design Workflows



- Construct Initial Modal
    - working model and code base is created. By “working” I mean not only that it compiles and runs, but also that it is robust and functionally correct.
- Identify Important Design Criteria
    - This is another commonly omitted step. It is important to remember that whenever some system aspects are optimized, it is almost always true that some other aspects are deoptimized. In order to select appropriate design patterns and technologies to achieve optimization, we need to first understand what are the important aspects to optimize, so we can trade them off appropriately.
    - Some common design optimization criteria are:
        - Performance
            - Worst case

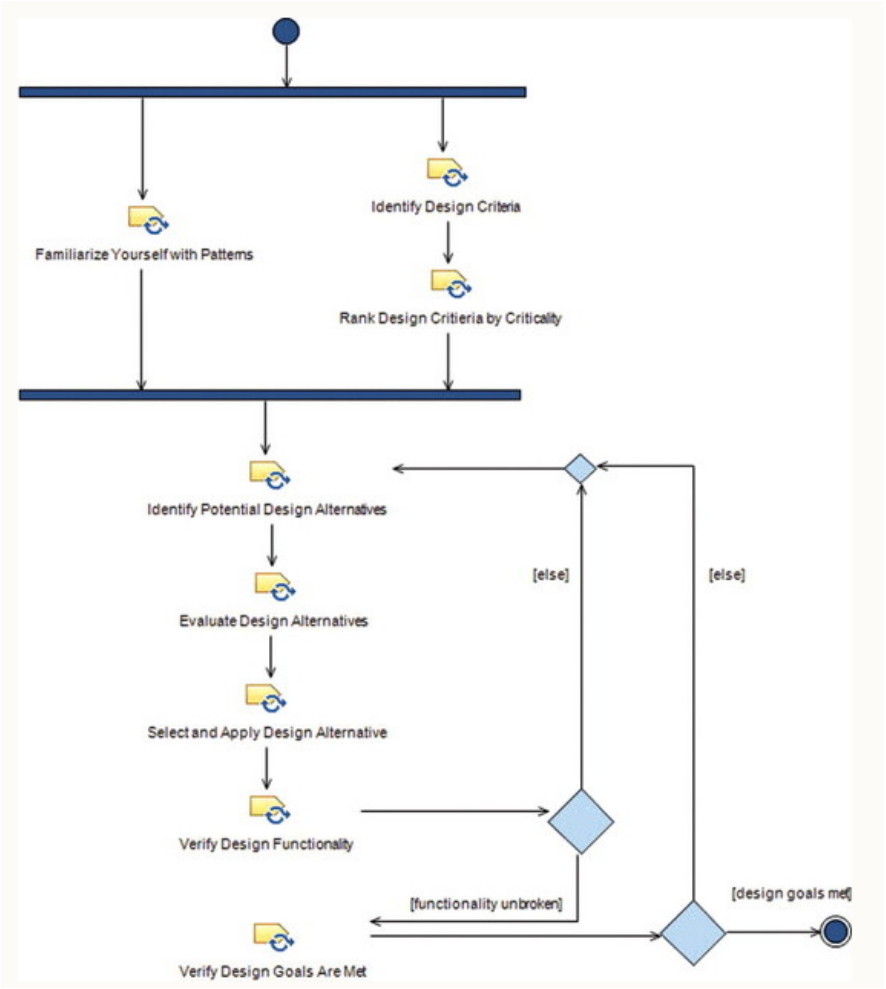
        - Average case
    - Predictability
    - Schedulability
    - Throughput
        - Average
        - Sustained
        - Burst
    - Reliability
        - With respect to errors
        - With respect to failures
    - Safety
    - Reusability
    - Distributability
    - Portability
    - Maintainability
    - Scalability
    - Complexity
    - Resource usage, e.g., memory
    - Energy consumption
    - Recurring cost, i.e., hardware
    - Development effort and cost
- Rank Design Criteria
    - The ranking of the criteria is necessary for us to make appropriate tradeoffs when we select design patterns. Some patterns will optimize worst-case performance at the expense of using more memory; others will optimize robustness at the expense of average case performance. Others may optimize energy economy at the expense of responsiveness. The ranking of the design criteria makes their relative importance explicit, so that we can reason about which design solutions are best.
- Selecting Design Pattern

Design Solution	Design Criteria					Total Weighted Score
	Criteria 1	Criteria 2	Criteria 3	Criteria 4	Criteria 5	
	Weight = 7	Weight = 5	Weight = 3	Weight = 2	Weight = 1.5	
	Score	Score	Score	Score	Score	
Alternative 1	7	3	6	9	4	106
Alternative 2	4	8	5	3	4	95
Alternative 3	10	2	4	8	8	**120**
Alternative 4	2	4	9	7	6	84

- Apply Design Patterns and Technologies
    - This step refers to the application of the selected design alternative. In design pattern terms, this is known as pattern instantiation
- Validate Design Solution
    - The validation step ensures that the solution works and checks two different facets of the solution.
        - First, it ensures that the functionality of the system continues to be correct; that is, adding in the design solution didn't break the running model and code.

    - Secondly, since we selected the design patterns to optimize the system, we must ensure that we've achieved our optimization goals

- **A design pattern** is "a generalized solution to a commonly occurring problem." To be a pattern, the problem must recur often enough to be usefully generalizable and the solution must be general enough to be applied in a wide set of application domains.
- If the solution only applies to a single application domain, then it is probably an **analysis pattern**. Analysis patterns define ways for organizing problem-specific models and code for a particular application domain.
- Remember that analysis is driven by what the system must do while design is driven by how well the system must achieve its requirements. A design pattern is a way of organizing a design that improves the optimality of a design with respect to one or a small set of design criteria, such as qualities of service (QoS).
- A design pattern always has a focused purpose – which is to take one or a small set of these QoS properties and optimize them at the expense of the others. Each pattern optimizes some aspect of a system at the cost of deoptimizing some other aspects. These constitute a set of pros and cons.
- A good design is one composed of a set of design patterns applied to a piece of functional software that achieves a balanced optimization of the design criteria and incurs an acceptable cost in doing so.



Design Patterns for Accessing Hardware

Hardware Proxy Pattern

- **PROBLEM**: If every client accesses a hardware device directly, problems due to hardware changes are exacerbated. If the bit encoding, memory address, or connection technology changes, then every client must be tracked down and modifie
- The Hardware Proxy Pattern creates a software element responsible for access to a piece of hardware and encapsulation of hardware compression and coding implementation.
- The Hardware Proxy Pattern uses a class (or struct) to encapsulate all access to a hardware device, regardless of its physical interface.
- he proxy publishes services that allow values to be read from and written to the device, as well as initialize, configure, and shut down the device as appropriate.
- The proxy provides an encoding and connection-independent interface for clients and so promotes easy modification should the nature of the device interface or connection change.
- **CONSEQUENCES**: This can, however, have a negative impact on run-time performance. It may sometimes be more time efficient for the clients to know the encoding details and manipulate the data in their native format.
- Collaboration Roles
    - Hardware Proxy Functions
        - **access**
        - **configure**
        - **disable**
        - **initialize**
        - **marshal** → performs any required encryption, compression, or bit-packing required to send the data to the device
        - **unmarshal**
        - **mutate** → This function writes data values to the device

Hardware Adapter Pattern

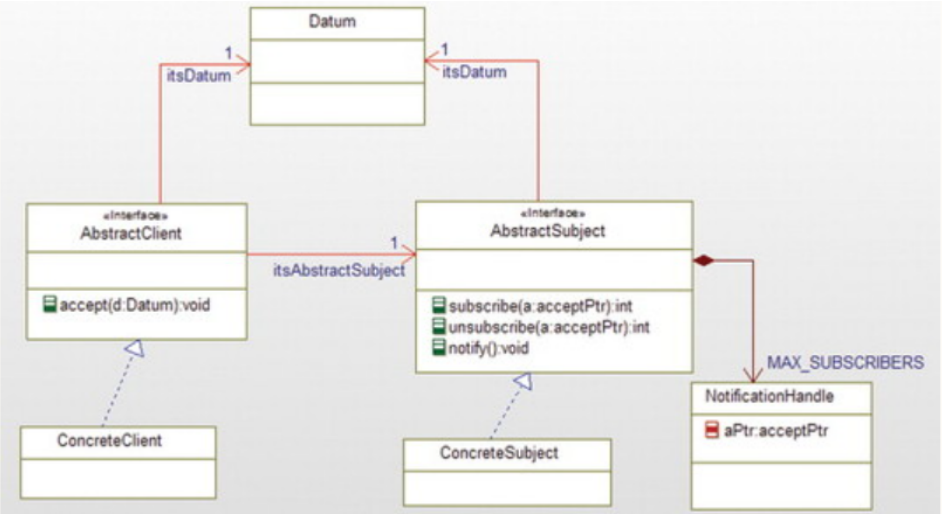
- **PROBLEM**: when you have hardware that meets the semantic need of the system but that has an incompatible interface.
- The Hardware Adapter Pattern provides a way of adapting an existing hardware interface into the expectations of the application. This pattern is a straightforward derivative of the Adapter Pattern.
- The Hardware Adapter Pattern is useful when the application requires or uses one interface, but the actual hardware provides another. The pattern creates an element that converts between the two interfaces.
- **CONSEQUENCES**: The cost of using the pattern is that it adds a level of indirection and therefore decreases run-time performance slightly.

Mediator Pattern

- **PROBLEM**: Many embedded applications control sets of actuators that must work in concert to achieve the desired effect. For example, to achieve a coordinated movement of a multi-joint robot arm, all the motors must work together to provide the desired arm movement. Similarly, using reaction wheels or thrusters in a spacecraft in three dimensions requires many different such devices acting at precisely the right time and with the right amount of force to achieve attitude stabilization.
- The Mediator Pattern provides a means of coordinating a complex interaction among a set of elements.
- The Mediator Pattern is particularly useful for managing different hardware elements when their behavior must be coordinated in well-defined but complex ways
- It is particularly useful for C applications because it doesn't require a lot of specialization (subclassing), which can introduce its own complexities into the implementation.
- The Mediator Pattern uses a mediator class to coordinate the actions of a set of collaborating devices to achieve the desired overall effect
- The Mediator class coordinates the control of the set of multiple Specific Collaborators. Each Specific Collaborator must be able to contact the Mediator when an event of interest occurs
- **CONSEQUENCES**: Since many embedded systems must react with high time fidelity, delays between the actions may result in unstable or undesirable effects. It is important that the mediator class can react within those time constraints

Observer Pattern

- The Observer Pattern (also known as the "Publish-Subscribe Pattern") provides notification to a set of interested clients that relevant data have changed.
- he clients simply offer a subscription function that allows clients to dynamically add (and remove) themselves to the notification list. The data server can then enforce whatever notification policy it desires.
- In a naïve situation, each client can request data periodically from a data server in case the data have changed, but that is wasteful of compute and communication resources as the clients generally cannot know when new data are available.
- The Observer Pattern addresses this concern by adding subscription and unsubscription services to the data server. Thus a client can dynamically add itself to the notification list without a priori knowledge of the client on the part of the server. On the server side, the server can enforce the appropriate update policy to the notification of its interested clients.
- When the AbstractSubject decides to notify its clients, the notify() function walks through the client list, calling the pointed-to function and passing the relevant data.
- The AbstractClient provides the accept(Datum) function to receive and process the incoming data.



Debouncing Pattern

- **PROBLEM**: Many input devices for embedded systems use metal-on-metal contact to indicate events of interest, such as button presses, switch movement, and activating or deactivating relays. As the metal moves into contact, physical deformation occurs resulting in an intermittent bouncing contact until the vibrations dampen down. This results in an intermediate contact profile
- This simple pattern is used to reject multiple false events arising from intermittent contact of metal surfaces.
- Push buttons, toggle switches, and electromechanical relays are input devices for digital systems that share a common problem – as metal connections make contact, the metal deforms or “bounces”, producing intermittent connections during switch open or closure.
- used **timer** and check **previous state** and decides event occurred or not, just that.

Interrupt Pattern

- **PROBLEM**: If the event is urgent, this pattern pausing the current processing, handling the incoming event
- The Interrupt Pattern is a way of structuring the system to respond appropriately to incoming events. It does require some processor- and compiler-specific services but most embedded compilers, and even some operating systems provide services for just this purpose. Once initialized, an interrupt will pause its normal processing, handle the incoming event, and then return the system to its original computations.
- **CONSEQUENCES**: Problems arise with this pattern when the ISR processing takes too long, when an implementation mistake leaves interrupts disabled, or race conditions or deadlocks occur on shared resources.

Polling Pattern

- **PROBLEM**: Polling is useful when the data or signals are not so urgent that they cannot wait until the next polling period to be received or when the hardware isn’t capable of generating interrupts when data or signals become available.

- Another common pattern for getting sensor data or signals from hardware is to check periodically, a process known as polling.
- The Polling Pattern addresses the concern of getting new sensor data or hardware signals into the system as it runs when the data or events are not highly urgent and the time between data sampling can be guaranteed to be fast enough
- **CONSEQUENCES**: If data arrive faster than the poll time, then data will be lost

Design Patterns for Embedding Concurrency and Resource Management

Scheduling Patterns

- **Deadline**: a point in time at which the completion of an action or action sequence becomes incorrect or irrelevant

Cyclic Executive Pattern

- **PROBLEM**: Many embedded systems are tiny applications that have extremely tight memory and time constraints and cannot possibly host even a scaled-down microkernel RTOS. The Cyclic Executive Pattern provides a low-resource means to accomplish this goa
- The Cyclic Executive Pattern has the advantage of being a very simple approach to scheduling multiple tasks but isn't very flexible or responsive to urgent events
- The Cyclic Executive Pattern implements a fairness doctrine for scheduling that allows all tasks an equal opportunity to run
- the deadline for each task must be greater than or equal to the sum of all the worst-case execution times for all the tasks plus the loop overhead
- The Cyclic Executive Pattern is used primarily in two situations.
    - First, for very small embedded applications, it allows multiple tasks to be run pseudoconcurrently without the overhead of a complex scheduler or RTOS.
    - Secondly, in high-safety relevant systems, the Cyclic Executive Pattern is easy to certify, hence its prevalence in avionics and flight management systems
- the primary benefit of this pattern is its simplicity.
- **CONSEQUENCES**: If an output of one task is needed by another, the data must be retained in global memory or a shared resource until that task runs

Static Priority Pattern

- **PROBLEM**: The Static Priority Pattern provides a simple approach to scheduling tasks based on priority. This pattern addresses systems with more tasks than the Cyclic Executive Pattern and also emphasizes responsiveness to urgent events over fairness.
- The Static Priority Pattern is a very common pattern in embedded systems because of its strong RTOS support, ease of use, and good responsiveness to urgent events
- The most common way to assign priorities is on the basis of deadline duration (that is, task urgency); the shorter the deadline, the higher the priority.
- **CONSEQUENCES**: It is common to use blocking to serialize access to resources

Task Coordination Patterns

Critical Region Pattern

- **PROBLEM**: The process may be accessing a resource that may not be safely accessed by multiple clients simultaneously
- The critical region pattern is the simplest pattern for task coordination around.
- It consists of disabling task switching during a region where it is important that the current task executes without being preempted.
- This can be because it is accessing a resource that cannot be simultaneously accessed safely, or because it is crucial that the task executes for a portion of time without interruption.
- In a preemptive multitasking environment (see the Static Priority Pattern, above), there may be periods of time during which a task must not be interrupted or preempted. The Critical Region Pattern address these intervals by disabling task switching or even by disabling interrupts
- **CONSEQUENCES**: Care must be taken when nesting calls to functions with their own critical regions. If such a function is called within the critical region of another, the nest function call will reenable task switching and end the critical region of the caller function.

Guarded Call Pattern

- **PROBLEM**: The problem this pattern addresses is the need for a timely synchronization or data exchange between threads
- The Guarded Call Pattern serializes access to a set of services that could potentially interfere with each other in some way if called simultaneously by multiple callers
- The Guarded Call Pattern uses semaphores to protect a related set of services (functions) from simultaneous access by multiple clients in a preemptive multitasking environment, in a process known as mutual exclusion
- **CONSEQUENCES**: This pattern provides a timely access to a resource and at the same time prevents multiple simultaneous accesses that can result in data corruption and system misbehavior. If the resource is not locked, then access proceeds without delay. If the resource is currently locked, then the caller must block until the lock on the resource is released

Queuing Pattern

- **PROBLEM**: In multithreaded systems, tasks must synchronize and share information with others. Two primary things must be accomplished. First, the tasks must synchronize to permit sharing of the information. Secondly, the information must be shared in such a way that there is no chance of corruption or race conditions. This pattern addresses such task interaction
- It provides a simple means of communication between tasks that are uncoupled in time. The Queuing Pattern accomplishes this communication by storing messages in a queue, a nominally first-in-first-out data structure
- t also provides a simple means of serializing access to a shared resource. The access messages are queued and handled at a later time and this avoids the mutual exclusion problem common to sharing resources.

- **CONSEQUENCES**: The number of elements held by the queue can be quite large; this is useful as the products of the elements run in a “bursty” fashion, allowing for the consumer of the elements to catch up later in time. Poorly sized queues can lead to data loss (if too small) or wasted memory (if too large). Very small queues (that can hold one or two elements) can be used for simple asynchronous data exchange when the problem of data loss isn’t present.

Rendezvous Pattern

- **PROBLEM**: The Rendezvous Pattern solves the problems when a set of tasks must synchronize in a complex fashion
- Tasks must synchronize in various ways. The Queuing and Guarded Call Patterns provide mechanisms when the synchronization occurs around a simple function call, sharing a single resource, or passing data. But what if the conditions required for synchronization are more complex? The Rendezvous Pattern reifies the strategy as an object itself and therefore supports even those most complex needs for task synchronization.
- he basic behavioral model is that as each thread becomes ready to rendezvous, it registers with the Rendezvous class, and then blocks until the Rendezvous class releases it to run. Once the set of preconditions is met, then the registered tasks are released to run using whatever scheduling policy is currently in force.
- **CONSEQUENCES**: The pattern is most appropriate when the tasks must halt at their synchronization point; that is, the pattern implements a blocking rendezvous. If a nonblocking rendezvous is desired, then this pattern can be mixed with a Queuing Pattern or the Observer Pattern with callbacks to a notify() function within the SynchronizingThread.

Deadlock Avoidance Patterns

- Deadlock is a situation in which multiple clients are simultaneously waiting for conditions that cannot, in principle, occur.

Simultaneous Locking Pattern

- Either all needed resources are locked at once or none are locked.
- This pattern prevents deadlock by removing required condition – some resources are locked while others are requested – by locking either all of the resources needed at once, or none of them
- **CONSEQUENCES**: However, while the pattern removes the possibility of deadlock, it may increase the delay of the execution of other tasks.

Ordered Locking

- he Ordered Locking Pattern is another way to ensure that deadlock cannot occur, this time by preventing circular waiting from occurring. It does this by ordering the resources and requiring that they always be locked by a client in that specified order.

Design Patterns for State Machines

- A synchronous event is activated by a call to an operation on the target state machine and the caller waits (blocked) until the state machine completes its response to the event. In contrast, an asynchronous event is sent via a send-and-forget mechanism by the sender and the sender may continue on regardless of the actions of the target state machine.
- We really want to be able to have the system respond in the same way to the same sequence of events; such a system is said to have deterministic semantics.

Single Event Receptor Pattern

- **PROBLEM**: This pattern addresses the problem of providing simple implementation of a state machine that applies to both synchronous and asynchronous event delivery.
- Single event receptor state machines (henceforth known as SERSMs) can be used for both synchronous and asynchronous events.

Multiple Event Receptor Pattern

- **PROBLEM**: This pattern addresses the problem of providing a simple robust implementation for synchronous state machines.
- Multiple event receptor finite state machines (MERSMs) are generally only used for synchronous state machines because the client is often aware of the set of events that it might want to send to the server state machine.
- In this pattern, there is a single event receptor for each event sent from the client.
- The MERSM approach for state machine implementation is probably the most common implementation pattern for synchronous state machines. It is usually the simplest because each event receptor concerns itself only with the handling of a single event and the execution of the associated actions.

State Table Pattern

- **PROBLEM**: The problem addressed by the State Table Pattern is to implement a state machine for potentially large state spaces that are flat or can easily be made so.
- The State Table Pattern is a create pattern for large, flat (i.e., without nesting or AND-states) state machines. It gives good performance (O(constant)) in use although it requires more initialization time before it becomes available.
- The State Table Pattern uses a two-dimensional array to hold the state transition information. This table is usually constructed as a state x event table; the state variable is the first index and the received event ID is the second index. The contents of the table is a structure containing links to the appropriate actions and guards and the new state, should the transition be taken.
- Also, this pattern is applicable when you want the performance of the state machine, in terms of the response time to a given event, independent of the size of the state space but you are unconcerned about the time necessary to initialize the state machine.

State Pattern

- **PROBLEM**: This pattern provides a means for implementing state machines that simplifies the stateful class by creating a set of objects, each of which implements the behavior of a single state. It makes the states explicit, rather easy to modify, and enables the extension of the state space by adding new state objects as needed.
- The State Pattern implements a state machine through the creation of state objects, one per state. The class owning the state machine, known as the Context, maintains a list of these objects with an internal variable that identifies which of these is the current state. All event receptors are passed to the currently active state object.

Safety and Reliability Patterns

One's Complement Pattern

- **PROBLEM**: This pattern addresses the problem that variables may be corrupted by a variety of causes such as environmental factors (e.g., EMI, heat, radiation), hardware faults (e.g., power fluctuation, memory cell faults, address line shorts), or software faults (other software erroneously modifying memory)
- The One's Complement Pattern is useful to detect when memory is corrupted by outside influences or hardware faults.
- The One's Complement Pattern provides a detailed design pattern for the identification of single or multiple memory bit corruption. This can occur due to EMI (Electromagnetic interference), heat, hardware faults, software faults, or other external causes. The pattern works by storing critical data twice – once in normal form and once in one's complement (bit-wise inverted) form. When the data are read, the one's complement form is reinverted and then compared to the value of the normal form. If the values match, then that value is returned, otherwise error processing ensues.

CRC Pattern

- **PROBLEM**: This pattern addresses the problem that variables may be corrupted by a variety of causes such as environmental factors (e.g., EMI, heat, radiation), hardware faults (e.g., power fluctuation, memory cell faults, address line shorts), or software faults (other software erroneously modifying memory). This pattern addresses the problem of data corruption in large data sets.
- The Cyclic Redundancy Check (CRC) Pattern calculates a fixed-size error-detection code based on a cyclic polynomial that can detect corruption in data sets far larger than the size of the code

Smart Data Pattern

- One of the biggest problems I see with embedded C code in actual application is that functions all have preconditions for their proper execution but these functions don't explicitly check that those preconditons are actually satisfied. This falls under the heading of what is commonly known as "defensive design" – a development

paradigm in which we design with proactive run-time defenses in place to detect problems. The Smart Data Pattern codifies that paradigm for scalar data elements.
- The most common idiom for error checking of data ranges in C is for functions that have no other return value: return a 0 value for function success and a -1 value for failure and set errno to an error code.
- The problem is, of course, that the most common way to “handle” the return values is to ignore them, resulting in difficult-to-debug systems. Some people even put error checking into the system during development but remove such checks in the final release. I would argue that the time when you really want to know that the altitude determination program in your aircraft isn’t producing the right value is when you are in the flying aircraft.
- The problem this pattern addresses is to build functions and data types that essentially check themselves and provide error detection means that cannot be easily ignored.

Channel Pattern

- **PROBLEM:** This pattern provides a basic element of large-scale redundancy that can be used in different ways to address safety and reliability concerns.
- A channel is an architectural structure that contains software (and possibly hardware as well) that performs end-to-end processing; that is, it processes data from raw acquisition, through a series of data processing steps, to physical actuation in the real world. The advantage of a channel is that it provides an independent, self-contained unit of functionality that can be replicated in different ways to address different safety and reliability concerns.

Protected Single Channel Pattern

- The Protected Single Channel Pattern is a simple elaboration of the Channel pattern

Dual Channel Pattern

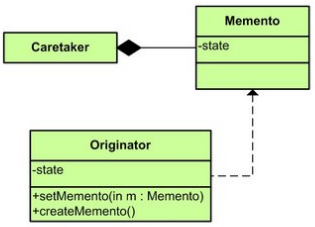
- The Dual Channel Pattern provides architectural redundancy to address safety and reliability concerns. It does this by replicating channels and embedding logic that manages them and determines when the channels will be “active.”

C Abstract Factory
S Adapter
S Bridge
C Builder
B Chain of Responsibility
B Command
S Composite
S Decorator
S Facade
C Factory Method
S Flyweight
B Interpreter
B Iterator
B Mediator
B Memento
C Prototype
S Proxy
B Observer
C Singleton
B State
B Strategy
B Template Method
B Visitor

Memento

Type: Behavioral

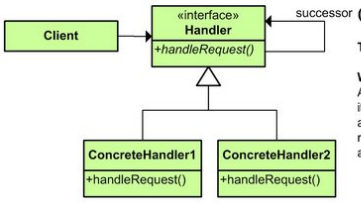
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

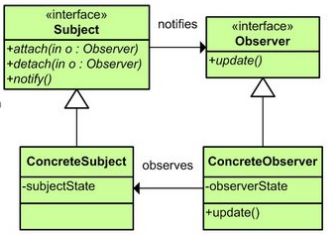
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Observer

Type: Behavioral

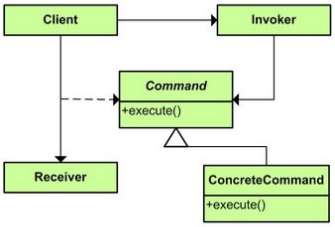
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Command

Type: Behavioral

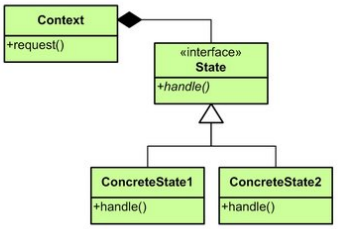
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



State

Type: Behavioral

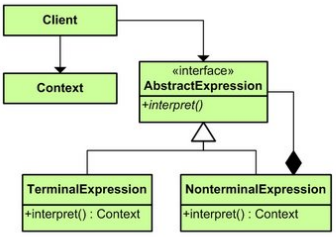
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Interpreter

Type: Behavioral

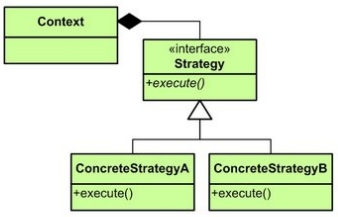
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Strategy

Type: Behavioral

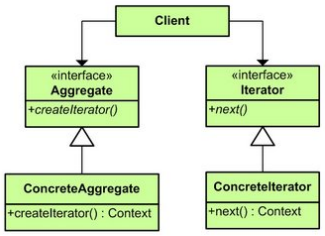
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Iterator

Type: Behavioral

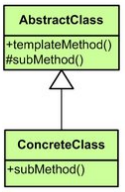
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Template Method

Type: Behavioral

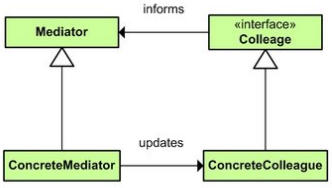
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Mediator

Type: Behavioral

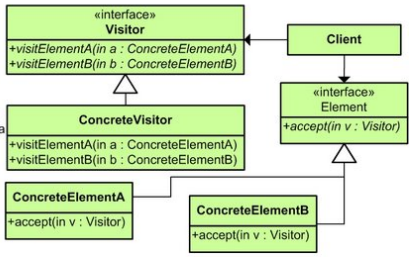
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.



Visitor

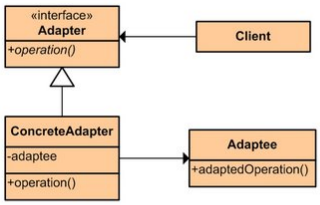
Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



Copyright © 2007 Jason S. McDonald
http://McDonaldLand.wordpress.com

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..



Adapter

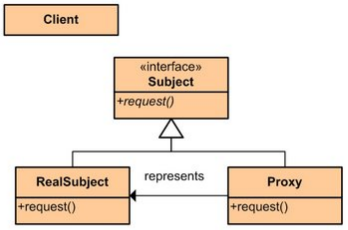
Type: Structural

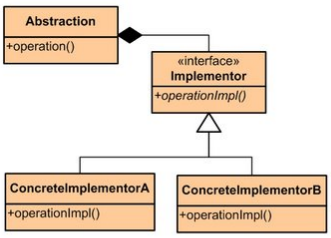
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.





Bridge

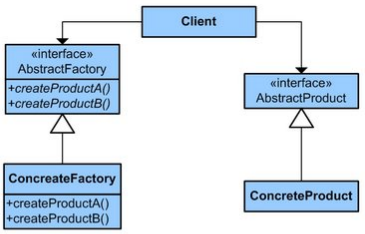
Type: Structural

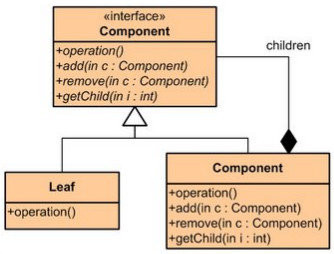
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.





Composite

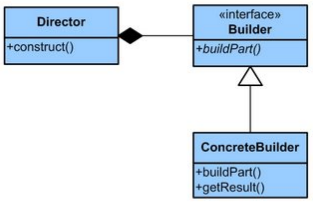
Type: Structural

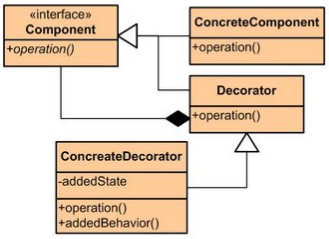
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.





Decorator

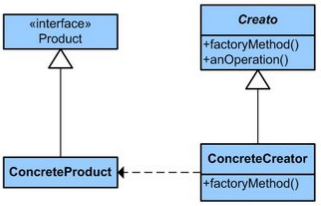
Type: Structural

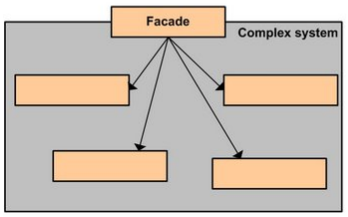
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.





Facade

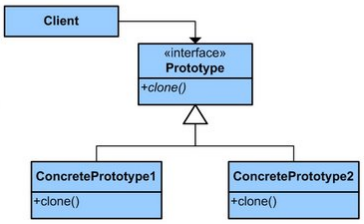
Type: Structural

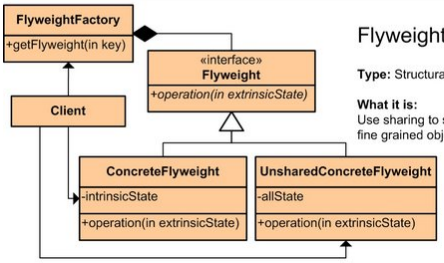
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Prototype

Type: Creational

What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.





Flyweight

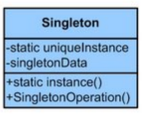
Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



Copyright © 2007 Jason S. McDonald
http://McDonaldLand.wordpress.com

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

Doxygen

```
/**
 *  @defgroup stack_functions Stack Functions Group
 *  stack functions group
 */

/** @struct StackNode
 *  @brief Stack Node element
 *  Detailed explenation
 *  @var int data::data itself
 *  @var StackNode next::next element pointer
 */

struct StackNode
{
          int data;
       struct StackNode *next;
};

/**
 * @ingroup stack_functions
 * @brief return element count in the stack
 * @param void
 * @return unsigned int returns element count
*/
unsigned int stackCount(void);

#define INITIAL_SIZE                    100     ///< Initial array size
#define SIZE_INCREMENT_COUNT            10      ///< If current size is not enough, size will
be incremented with this parameter
```

- Create your config file by using Doxy Wizard
- then type **doxygen mydoxy.conf** where myconfig is your config file.

Here is my config file;

```
# Doxyfile 1.9.1

#---------------------------------------------------------------------------
# Project related configuration options
#---------------------------------------------------------------------------
DOXYFILE_ENCODING      = UTF-8
PROJECT_NAME           = DataStructures
```

```
PROJECT_NUMBER         =
PROJECT_BRIEF          =
PROJECT_LOGO           =
OUTPUT_DIRECTORY       = /home/dodo/Workspace/AlgorithmsAndDataStructures
CREATE_SUBDIRS         = NO
ALLOW_UNICODE_NAMES    = NO
OUTPUT_LANGUAGE        = English
OUTPUT_TEXT_DIRECTION  = None
BRIEF_MEMBER_DESC      = YES
REPEAT_BRIEF           = YES
ABBREVIATE_BRIEF       = "The $name class" \
                         "The $name widget" \
                         "The $name file" \
                         is \
                         provides \
                         specifies \
                         contains \
                         represents \
                         a \
                         an \
                         the
ALWAYS_DETAILED_SEC    = NO
INLINE_INHERITED_MEMB  = NO
FULL_PATH_NAMES        = YES
STRIP_FROM_PATH        =
STRIP_FROM_INC_PATH    =
SHORT_NAMES            = NO
JAVADOC_AUTOBRIEF      = NO
JAVADOC_BANNER         = NO
QT_AUTOBRIEF           = NO
MULTILINE_CPP_IS_BRIEF = NO
PYTHON_DOCSTRING       = YES
INHERIT_DOCS           = YES
SEPARATE_MEMBER_PAGES  = NO
TAB_SIZE               = 4
ALIASES                =
OPTIMIZE_OUTPUT_FOR_C  = NO
OPTIMIZE_OUTPUT_JAVA   = NO
OPTIMIZE_FOR_FORTRAN   = NO
OPTIMIZE_OUTPUT_VHDL   = NO
OPTIMIZE_OUTPUT_SLICE  = NO
EXTENSION_MAPPING      =
MARKDOWN_SUPPORT       = YES
TOC_INCLUDE_HEADINGS   = 5
AUTOLINK_SUPPORT       = YES
BUILTIN_STL_SUPPORT    = NO
CPP_CLI_SUPPORT        = NO
SIP_SUPPORT            = NO
```

```
IDL_PROPERTY_SUPPORT   = YES
DISTRIBUTE_GROUP_DOC   = NO
GROUP_NESTED_COMPOUNDS = NO
SUBGROUPING            = YES
INLINE_GROUPED_CLASSES = NO
INLINE_SIMPLE_STRUCTS  = NO
TYPEDEF_HIDES_STRUCT   = NO
LOOKUP_CACHE_SIZE      = 0
NUM_PROC_THREADS       = 1
#---------------------------------------------------------------------------
# Build related configuration options
#---------------------------------------------------------------------------
EXTRACT_ALL            = YES
EXTRACT_PRIVATE        = NO
EXTRACT_PRIV_VIRTUAL   = NO
EXTRACT_PACKAGE        = NO
EXTRACT_STATIC         = NO
EXTRACT_LOCAL_CLASSES  = YES
EXTRACT_LOCAL_METHODS  = NO
EXTRACT_ANON_NSPACES   = NO
RESOLVE_UNNAMED_PARAMS = YES
HIDE_UNDOC_MEMBERS     = NO
HIDE_UNDOC_CLASSES     = NO
HIDE_FRIEND_COMPOUNDS  = NO
HIDE_IN_BODY_DOCS      = NO
INTERNAL_DOCS          = NO
CASE_SENSE_NAMES       = NO
HIDE_SCOPE_NAMES       = NO
HIDE_COMPOUND_REFERENCE= NO
SHOW_INCLUDE_FILES     = YES
SHOW_GROUPED_MEMB_INC  = NO
FORCE_LOCAL_INCLUDES   = NO
INLINE_INFO            = YES
SORT_MEMBER_DOCS       = YES
SORT_BRIEF_DOCS        = NO
SORT_MEMBERS_CTORS_1ST = NO
SORT_GROUP_NAMES       = NO
SORT_BY_SCOPE_NAME     = NO
STRICT_PROTO_MATCHING  = NO
GENERATE_TODOLIST      = YES
GENERATE_TESTLIST      = YES
GENERATE_BUGLIST       = YES
GENERATE_DEPRECATEDLIST= YES
ENABLED_SECTIONS       =
MAX_INITIALIZER_LINES  = 30
SHOW_USED_FILES        = YES
SHOW_FILES             = YES
SHOW_NAMESPACES        = YES
```

```
FILE_VERSION_FILTER    =
LAYOUT_FILE            =
CITE_BIB_FILES         =
#---------------------------------------------------------------------------
# Configuration options related to warning and progress messages
#---------------------------------------------------------------------------
QUIET                  = NO
WARNINGS               = YES
WARN_IF_UNDOCUMENTED   = YES
WARN_IF_DOC_ERROR      = YES
WARN_NO_PARAMDOC       = NO
WARN_AS_ERROR          = NO
WARN_FORMAT            = "$file:$line: $text"
WARN_LOGFILE           =
#---------------------------------------------------------------------------
# Configuration options related to the input files
#---------------------------------------------------------------------------
INPUT                  = /home/dodo/Workspace/AlgorithmsAndDataStructures
INPUT_ENCODING         = UTF-8
FILE_PATTERNS          = *.c \
                         *.cc \
                         *.cxx \
                         *.cpp \
                         *.c++ \
                         *.java \
                         *.ii \
                         *.ixx \
                         *.ipp \
                         *.i++ \
                         *.inl \
                         *.idl \
                         *.ddl \
                         *.odl \
                         *.h \
                         *.hh \
                         *.hxx \
                         *.hpp \
                         *.h++ \
                         *.cs \
                         *.d \
                         *.php \
                         *.php4 \
                         *.php5 \
                         *.phtml \
                         *.inc \
                         *.m \
                         *.markdown \
                         *.md \
```

```
                         *.mm \
                         *.dox \
                         *.py \
                         *.pyw \
                         *.f90 \
                         *.f95 \
                         *.f03 \
                         *.f08 \
                         *.f18 \
                         *.f \
                         *.for \
                         *.vhd \
                         *.vhdl \
                         *.ucf \
                         *.qsf \
                         *.ice
RECURSIVE              = YES
EXCLUDE                =
EXCLUDE_SYMLINKS       = NO
EXCLUDE_PATTERNS       =
EXCLUDE_SYMBOLS        =
EXAMPLE_PATH           =
EXAMPLE_PATTERNS       = *
EXAMPLE_RECURSIVE      = NO
IMAGE_PATH             =
INPUT_FILTER           =
FILTER_PATTERNS        =
FILTER_SOURCE_FILES    = NO
FILTER_SOURCE_PATTERNS =
USE_MDFILE_AS_MAINPAGE =
#---------------------------------------------------------------------------
# Configuration options related to source browsing
#---------------------------------------------------------------------------
SOURCE_BROWSER         = NO
INLINE_SOURCES         = NO
STRIP_CODE_COMMENTS    = YES
REFERENCED_BY_RELATION = NO
REFERENCES_RELATION    = NO
REFERENCES_LINK_SOURCE = YES
SOURCE_TOOLTIPS        = YES
USE_HTAGS              = NO
VERBATIM_HEADERS       = YES
CLANG_ASSISTED_PARSING = NO
CLANG_ADD_INC_PATHS    = YES
CLANG_OPTIONS          =
CLANG_DATABASE_PATH    =
#---------------------------------------------------------------------------
# Configuration options related to the alphabetical class index
```

```
#---------------------------------------------------------------------------
ALPHABETICAL_INDEX     = YES
IGNORE_PREFIX          =
#---------------------------------------------------------------------------
# Configuration options related to the HTML output
#---------------------------------------------------------------------------
GENERATE_HTML          = YES
HTML_OUTPUT            = html
HTML_FILE_EXTENSION    = .html
HTML_HEADER            =
HTML_FOOTER            =
HTML_STYLESHEET        =
HTML_EXTRA_STYLESHEET  =
HTML_EXTRA_FILES       =
HTML_COLORSTYLE_HUE    = 220
HTML_COLORSTYLE_SAT    = 100
HTML_COLORSTYLE_GAMMA  = 80
HTML_TIMESTAMP         = NO
HTML_DYNAMIC_MENUS     = YES
HTML_DYNAMIC_SECTIONS  = NO
HTML_INDEX_NUM_ENTRIES = 100
GENERATE_DOCSET        = NO
DOCSET_FEEDNAME        = "Doxygen generated docs"
DOCSET_BUNDLE_ID       = org.doxygen.Project
DOCSET_PUBLISHER_ID    = org.doxygen.Publisher
DOCSET_PUBLISHER_NAME  = Publisher
GENERATE_HTMLHELP      = NO
CHM_FILE               =
HHC_LOCATION           =
GENERATE_CHI           = NO
CHM_INDEX_ENCODING     =
BINARY_TOC             = NO
TOC_EXPAND             = NO
GENERATE_QHP           = NO
QCH_FILE               =
QHP_NAMESPACE          = org.doxygen.Project
QHP_VIRTUAL_FOLDER     = doc
QHP_CUST_FILTER_NAME   =
QHP_CUST_FILTER_ATTRS  =
QHP_SECT_FILTER_ATTRS  =
QHG_LOCATION           =
GENERATE_ECLIPSEHELP   = NO
ECLIPSE_DOC_ID         = org.doxygen.Project
DISABLE_INDEX          = NO
GENERATE_TREEVIEW      = NO
ENUM_VALUES_PER_LINE   = 4
TREEVIEW_WIDTH         = 250
EXT_LINKS_IN_WINDOW    = NO
```

```
HTML_FORMULA_FORMAT    = png
FORMULA_FONTSIZE       = 10
FORMULA_TRANSPARENT    = YES
FORMULA_MACROFILE      =
USE_MATHJAX            = NO
MATHJAX_FORMAT         = HTML-CSS
MATHJAX_RELPATH        = https://cdn.jsdelivr.net/npm/mathjax@2
MATHJAX_EXTENSIONS     =
MATHJAX_CODEFILE       =
SEARCHENGINE           = YES
SERVER_BASED_SEARCH    = NO
EXTERNAL_SEARCH        = NO
SEARCHENGINE_URL       =
SEARCHDATA_FILE        = searchdata.xml
EXTERNAL_SEARCH_ID     =
EXTRA_SEARCH_MAPPINGS  =
#---------------------------------------------------------------------------
# Configuration options related to the LaTeX output
#---------------------------------------------------------------------------
GENERATE_LATEX         = NO
LATEX_OUTPUT           = latex
LATEX_CMD_NAME         =
MAKEINDEX_CMD_NAME     = makeindex
LATEX_MAKEINDEX_CMD    = makeindex
COMPACT_LATEX          = NO
PAPER_TYPE             = a4
EXTRA_PACKAGES         =
LATEX_HEADER           =
LATEX_FOOTER           =
LATEX_EXTRA_STYLESHEET =
LATEX_EXTRA_FILES      =
PDF_HYPERLINKS         = YES
USE_PDFLATEX           = YES
LATEX_BATCHMODE        = NO
LATEX_HIDE_INDICES     = NO
LATEX_SOURCE_CODE      = NO
LATEX_BIB_STYLE        = plain
LATEX_TIMESTAMP        = NO
LATEX_EMOJI_DIRECTORY  =
#---------------------------------------------------------------------------
# Configuration options related to the RTF output
#---------------------------------------------------------------------------
GENERATE_RTF           = NO
RTF_OUTPUT             = rtf
COMPACT_RTF            = NO
RTF_HYPERLINKS         = NO
RTF_STYLESHEET_FILE    =
RTF_EXTENSIONS_FILE    =
```

```
RTF_SOURCE_CODE        = NO
#---------------------------------------------------------------------------
# Configuration options related to the man page output
#---------------------------------------------------------------------------
GENERATE_MAN           = NO
MAN_OUTPUT             = man
MAN_EXTENSION          = .3
MAN_SUBDIR             =
MAN_LINKS              = NO
#---------------------------------------------------------------------------
# Configuration options related to the XML output
#---------------------------------------------------------------------------
GENERATE_XML           = NO
XML_OUTPUT             = xml
XML_PROGRAMLISTING     = YES
XML_NS_MEMB_FILE_SCOPE = NO
#---------------------------------------------------------------------------
# Configuration options related to the DOCBOOK output
#---------------------------------------------------------------------------
GENERATE_DOCBOOK       = NO
DOCBOOK_OUTPUT         = docbook
DOCBOOK_PROGRAMLISTING = NO
#---------------------------------------------------------------------------
# Configuration options for the AutoGen Definitions output
#---------------------------------------------------------------------------
GENERATE_AUTOGEN_DEF   = NO
#---------------------------------------------------------------------------
# Configuration options related to Sqlite3 output
#---------------------------------------------------------------------------
#---------------------------------------------------------------------------
# Configuration options related to the Perl module output
#---------------------------------------------------------------------------
GENERATE_PERLMOD       = NO
PERLMOD_LATEX          = NO
PERLMOD_PRETTY         = YES
PERLMOD_MAKEVAR_PREFIX =
#---------------------------------------------------------------------------
# Configuration options related to the preprocessor
#---------------------------------------------------------------------------
ENABLE_PREPROCESSING   = YES
MACRO_EXPANSION        = NO
EXPAND_ONLY_PREDEF     = NO
SEARCH_INCLUDES        = YES
INCLUDE_PATH           =
INCLUDE_FILE_PATTERNS  =
PREDEFINED             =
EXPAND_AS_DEFINED      =
SKIP_FUNCTION_MACROS   = YES
```

```
#---------------------------------------------------------------------------
# Configuration options related to external references
#---------------------------------------------------------------------------
TAGFILES               =
GENERATE_TAGFILE       =
ALLEXTERNALS           = NO
EXTERNAL_GROUPS        = YES
EXTERNAL_PAGES         = YES
#---------------------------------------------------------------------------
# Configuration options related to the dot tool
#---------------------------------------------------------------------------
CLASS_DIAGRAMS         = YES
DIA_PATH               =
HIDE_UNDOC_RELATIONS   = YES
HAVE_DOT               = YES
DOT_NUM_THREADS        = 0
DOT_FONTNAME           = Helvetica
DOT_FONTSIZE           = 10
DOT_FONTPATH           =
CLASS_GRAPH            = YES
COLLABORATION_GRAPH    = YES
GROUP_GRAPHS           = YES
UML_LOOK               = NO
UML_LIMIT_NUM_FIELDS   = 10
DOT_UML_DETAILS        = NO
DOT_WRAP_THRESHOLD     = 17
TEMPLATE_RELATIONS     = NO
INCLUDE_GRAPH          = YES
INCLUDED_BY_GRAPH      = YES
CALL_GRAPH             = NO
CALLER_GRAPH           = NO
GRAPHICAL_HIERARCHY    = YES
DIRECTORY_GRAPH        = YES
DOT_IMAGE_FORMAT       = png
INTERACTIVE_SVG        = NO
DOT_PATH               =
DOTFILE_DIRS           =
MSCFILE_DIRS           =
DIAFILE_DIRS           =
PLANTUML_JAR_PATH      =
PLANTUML_CFG_FILE      =
PLANTUML_INCLUDE_PATH  =
DOT_GRAPH_MAX_NODES    = 50
MAX_DOT_GRAPH_DEPTH    = 0
DOT_TRANSPARENT        = NO
DOT_MULTI_TARGETS      = NO
GENERATE_LEGEND        = YES
DOT_CLEANUP            = YES
```