Memory Leak Detector

For

C/C++

Project : Design and Implement Memory Leak Detector

- Since the advent of C/C++ Programming language, Memory management is one of the responsibility which the developer has to deal with

- C/C++ Softwares often suffers from Two Memory related Problems like
  - Memory corruption
  - Memory leak

- Unlike Java, C/C++ do not have the luxury for automatic garbage collection

- Java do not allow programmer to access the physical memory directly, but C/C++ does. Therefore Java applications do not suffer from Memory corruption either, but C/C++ do

- In this project, we will design and implement memory leak detector (MLD) tool for C programs, easily extendible to C++ as well

- Million $$ Question is : If designing MLD tool was that easy, why we don't have it integrate with C programming language already ?? Complete this course to get the answer ✌

Project : Design and Implement Memory Leak Detector

Tools you need for the project :

1. Any Version of Linux/Unix OS
2. Compiler used : gcc

Pre-requisites :

Good knowledge of C programming knowledge, and Pointers
Elementary knowledge of OS Memory Management

If you are doing this project on Windows, please use Windows compatible compiler like CLANG, Code-blocks at your own Responsibility

System Programming in the Industry is done on linux platforms, so, leave the habit of using Windows platform, Sooner the better

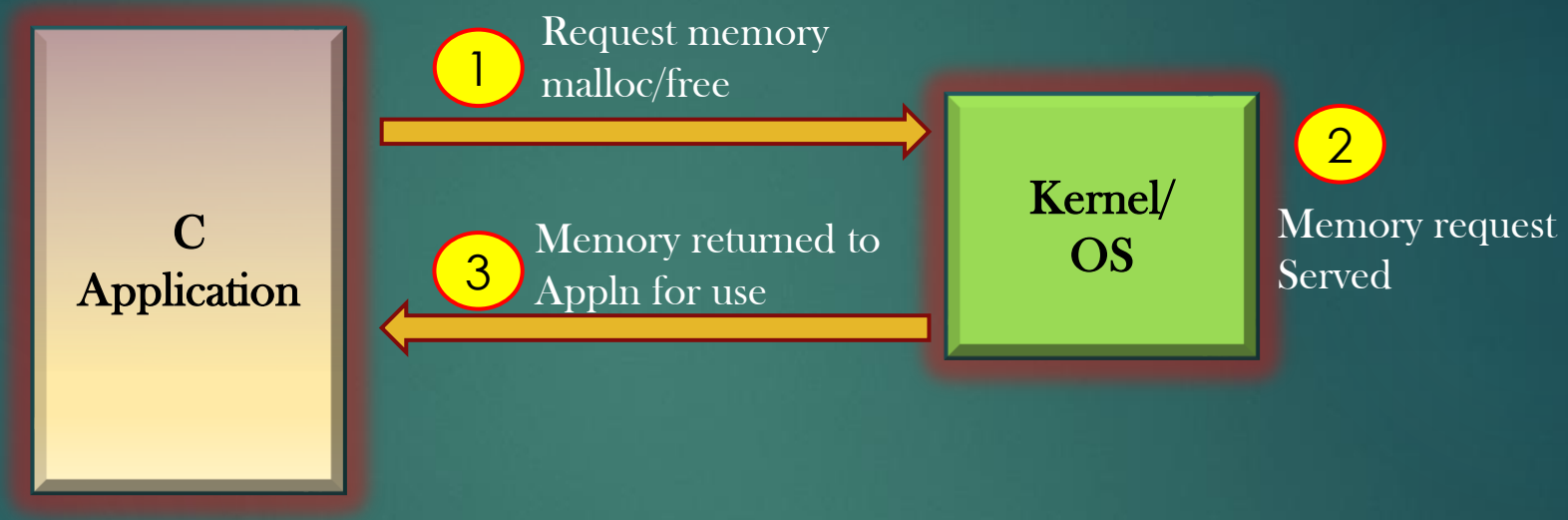
Code Samples/Examples : https://github.com/sachinites/MemoryLeakDetector

Project : Design and Implement Memory Leak Detector

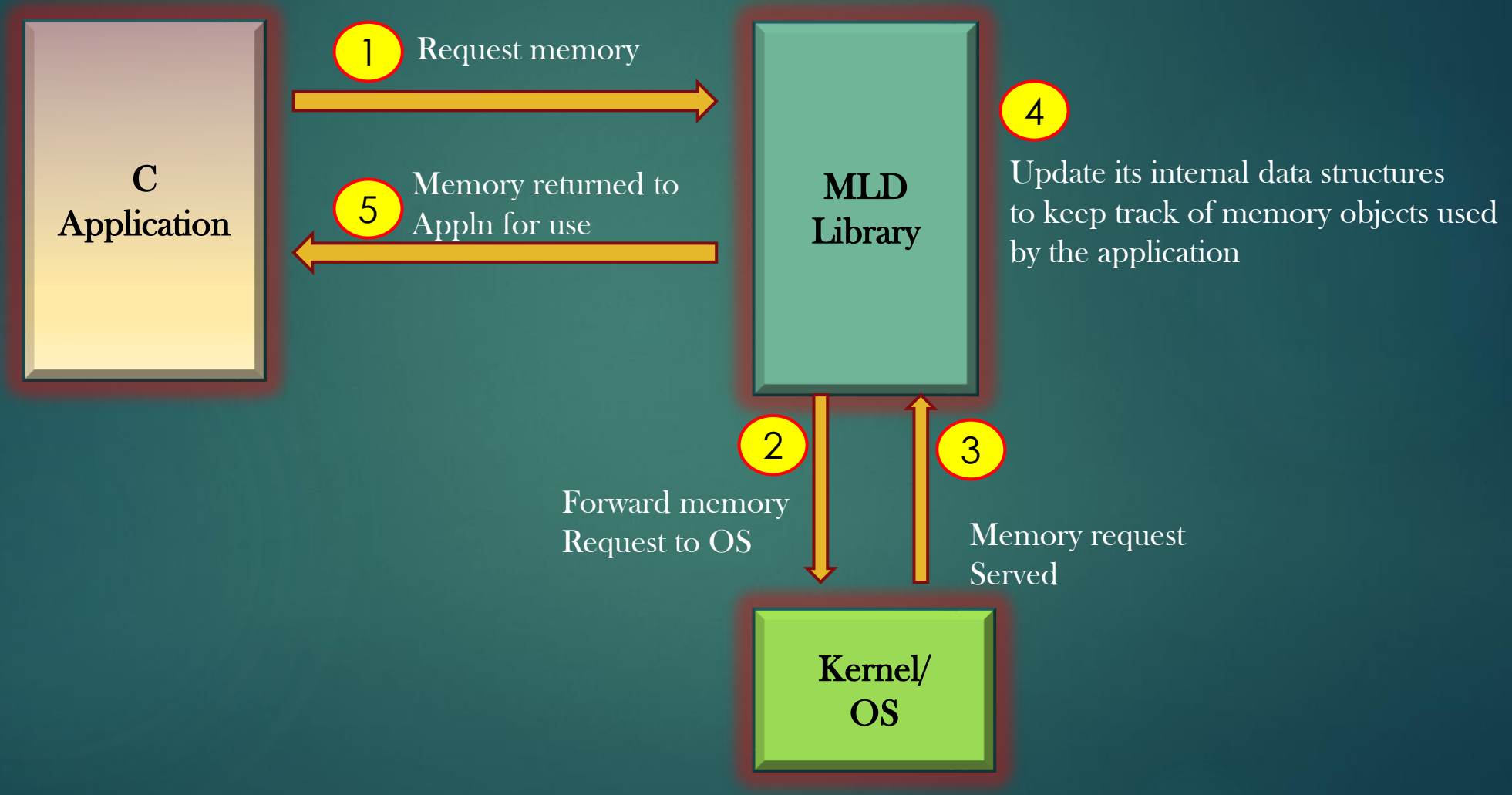
- I Assume, you already familiar with what Memory Leak is !!
- In this Project we shall going to create a Tool called Memory Leak Detector (MLD) tool
- We shall implement this tool as the library, and your C/C++ application will use this library for memory management
- MLD library will keep track of all the Heap Objects the application has created, and how various heap objects holds references to one another
- MLD library will show you the leaked objects, if any
- Show this project on your resume with proud ! ☺
- Advise : Create your github account to maintain all your codes

Project : Design and Implement Memory Leak Detector

Normal Scenario

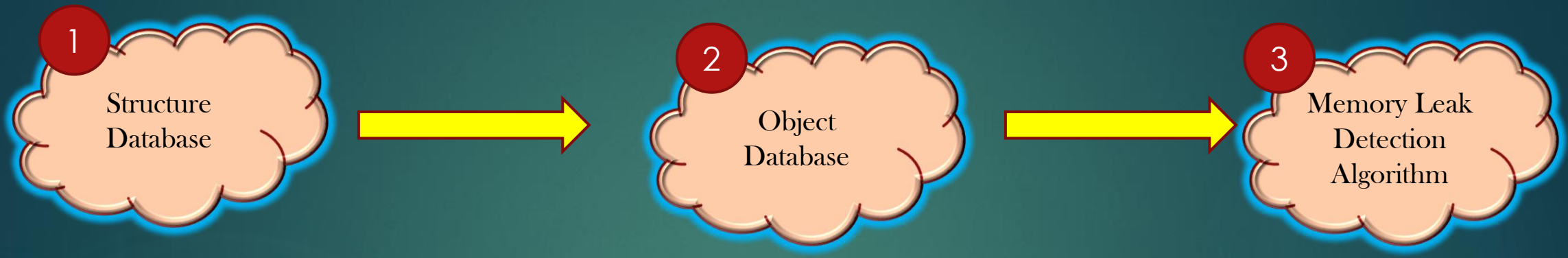


Our Project – High Level Block Diagram



Project : Design and Implement Memory Leak Detector

Project Development Phases



Phase 1 : MLD library will maintain the information about all structures which the application is using

Phase 2 : MLD library will maintain the information about all objects malloc'd by the application

Phase 3 : MLD Library triggers Memory Leak Detection Algorithm on Object database to find leaked objects

Project : Design and Implement Memory Leak Detector

MLD Library Database

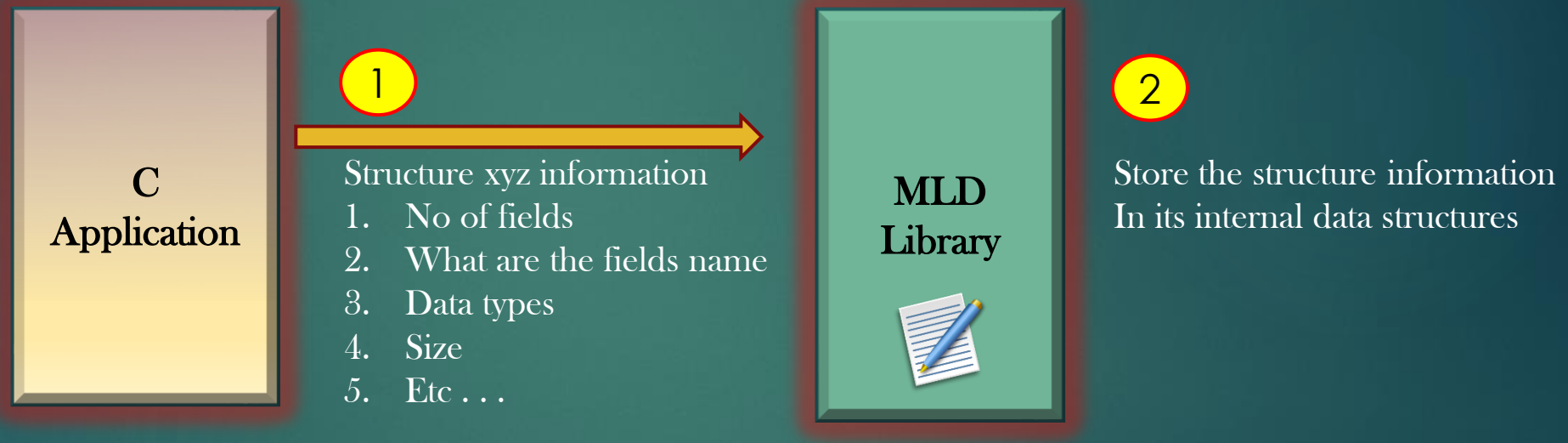
Project Development Phase : 1
Designing MLD Structure Database

Goals :

- MLD library must know the information about all structures being used by the application
- It is the responsibility of the application to tell the MLD library during initialization about all structures it is using. This is called "*structure registration*"
- MLD library will maintain the structure database (preferably a linked-list) to store application structure information
- Key to search in structure database is "name of structure"
- Let us see how it can be done step by step !

Project Development Phase : 1
Designing MLD Structure Database

Structure Registration



Project Development Phase : 1
Designing MLD Structure Database

Modelling of Structure Database

```
#define MAX_STRUCTURE_NAME_SIZE 128
#define MAX_FIELD_NAME_SIZE 128

typedef struct _struct_db_rec_ struct_db_rec_t;

struct _struct_db_rec_{
    struct_db_rec_t *next;
    char struct_name [MAX_STRUCTURE_NAME_SIZE];  // key
    unsigned int ds_size;
    unsigned int n_fields;
    field_info_t *fields;
};
```

1

```
typedef struct _field_info_{
    char fname [MAX_FIELD_NAME_SIZE];
    unsigned int size;
    unsigned int offset;
    data_type_t dtype;
    char nested_str_name[MAX_STRUCTURE_NAME_SIZE];
} field_info_t;
```

2

```
typedef enum {
    UINT8,
    UINT32,
    INT32,
    CHAR,
    OBJ_PTR,
    FLOAT,
    DOUBLE,
    OBJ_STRUCT
} data_type_t;
```

3

Project Development Phase : 1
Designing MLD Structure Database

Example :

```
typedef struct emp_ {

        char emp_name[30];
        unsigned int emp_id;
        unsigned int age;
        struct emp_ *mgr;
        float salary;
} emp_t;
```

Fields[0]	"emp_name"	sizeof(char) * 30	0	CHAR	NULL
Fields[1]	"emp_id"	sizeof(unsigned int)	30	UINT	NULL
Fields[2]	"age"	sizeof(unsigned int)	34	UINT	NULL
Fields[3]	"mgr"	sizeof(void *)	38	OBJ_PTR	emp_t
Fields[4]	"salary"	sizeof(float)	42	FLOAT	NULL

field_info_t

NULL	"emp_t"	sizeof(emp_t)	5	fields

struct_db_rec_t

Project Development Phase : 1
Designing MLD Structure Database

Helping Macros :

Q. Given a Structure and a field name within the structure, write a C macro which compute :

1. *Size of the field*
2. *Offset of the field*

```
typedef struct emp_ {

        char emp_name[30];
        unsigned int emp_id;
        unsigned int age;
        struct emp_ *mgr;
        float salary;
} emp_t;
```

```
#define OFFSETOF(struct_name, fld_name)     \
    (unsigned int)&(((struct_name  *)0)->fld_name)
```

```
#define FIELD_SIZE(struct_name, fld_name)   \
    sizeof(((struct_name  *)0)->fld_name)
```

Project Development Phase : 1
Designing MLD Structure Database

Helping Macros :

Q. Given the below structure, create an array of objects of this structure type

```
typedef struct _student {
        char stud_name[32];
        int rollno;
        int age;
} student_t;
```

Soln :

```
student_t stud_array[] = {
        {"Abhishek", 123, 168},
        {"Shivani", 111, 151}
};
```

Project Development Phase : 1
Designing MLD Structure Database

Structure Database Code Walk

Finally,

1. Let us see how the application is suppose to populate the structure database of MLD library

2. Let us verify using printing functions that structure registration done by the app with MLD library is correct

Lets Walk through the code which illustrate the example

Project Development Phase : 1
Designing MLD Structure Database

Example :

```
typedef struct emp_ {

        char emp_name[30];
        unsigned int emp_id;
        unsigned int age;
        struct emp_ *mgr;
        float salary;
} emp_t;
```

Fields[0]	“emp_name”	sizeof(char) * 30	0	CHAR	NULL
Fields[1]	“emp_id”	sizeof(unsigned int)	30	UINT	NULL
Fields[2]	“age”	sizeof(unsigned int)	34	UINT	NULL
Fields[3]	“mgr”	sizeof(void *)	38	OBJ_PTR	emp_t
Fields[4]	“salary”	sizeof(float)	42	FLOAT	NULL

field_info_t

NULL	“emp_t”	sizeof(emp_t)	5	fields

struct_db_rec_t

Project Development Phase : 1
Designing MLD Structure Database

```
typedef struct emp_ {

    char emp_name[30];
    unsigned int emp_id;
    unsigned int age;
    struct emp_ *mgr;
    float salary;
} emp_t;
```

```
/*Step 1 : Crate an array which have field's information*/
static field_info_t emp_fields[] = {
        FIELD_INFO(emp_t, emp_name, CHAR,    0),
        FIELD_INFO(emp_t, emp_id,   UINT32,  0),
        FIELD_INFO(emp_t, age,      UINT32,  0),
        FIELD_INFO(emp_t, mgr,      OBJ_PTR, emp_t),
        FIELD_INFO(emp_t, salary,   FLOAT, 0)
    };

    /*Step 2 : Register the structure in structure database*/
    REG_STRUCT(struct_db, emp_t, emp_fields);
```

```
#define FIELD_INFO(struct_name, fld_name, dtype, nested_struct_name)    \
    {#fld_name, dtype, FIELD_SIZE(struct_name, fld_name),               \
        OFFSETOF(struct_name, fld_name), #nested_struct_name}
```

Project Development Phase : 1
Designing MLD Structure Database

Printing functions

- Write a *print_structure_rec()* function to dump the information of one structure record

```
void
print_structure_rec (struct_db_rec_t *struct_rec);
```

- Write a *print_structure_db()* function to dump the structure database info

```
void
print_structure_db(struct_db_t *struct_db);
```

- Fn to add the structure record in a structure database

```
int /*return 0 on success, -1 on failure for some reason */
add_structure_to_struct_db(struct_db_t *struct_db, struct_db_rec_t *struct_rec);
```

Project Development Phase : 1
Designing MLD Structure Database

Summary

- We complete the Phase 1 of the project
- So, our MLD library now have all the information about structures being used by the application
- MLD library can now use the structure database to manipulate application objects in whatever way it wishes to
- It is the application responsibility to populate MLD's structure database info at the time of start
- Now, we enter phase 2 - > Object database Creation

Project Development Phase : 1
Designing MLD Structure Database

- Exercise :

Implement the following function in mld.c/.h. The function must return pointer to the structure record corresponding to the structure name passes as second arg. If such a record is not found, return NULL

```
struct_db_rec_t *
struct_db_look_up(struct_db_t *struct_db,
                  char *struct_name);
```

Project Development Phase : 2
Design and Implement Object database

Goals :

- Now, Its time that MLD libraries also knows about all objects the application has malloc'd
- Whenever the application malloc a new object, MLD library will store the relevant information about this object such as
  - Corresponding structure details of the object
  - Address of the object
- The object record holds the above information of the object
- Idea is, MLD library must have all information about all dynamic objects the application is using at any point of time
- MLD library maintains a database called *Object database* to keep track of all dynamic objects being used by the application
- Let us discuss the implementation step by step

Project Development Phase : 2
Design and Implement Object database

```
typedef struct _object_db_rec_{

    object_db_rec_t *next;
    void *ptr;  /*Key*/
    unsigned int units;
    struct_db_rec_t *struct_rec;
} object_db_rec_t;
```

```
typedef struct _object_db_{
    struct_db_t *struct_db;
    object_db_rec_t *head;
    unsigned int count;
} object_db_t;
```

We will write our own calloc, lets call it xcalloc

```
void *
xcalloc (object_db_t *object_db, char *struct_name, int units );
```

Eg : emp_t * emp = xcalloc(object_db, “emp_t”, 1);

xcalloc does the following :

1. Allocate “units” units of contiguous memory for object of type “struct_name”
2. Create the object record for new allocated object, and add the object record in object database
3. Link the object record with structure record for structure “struct_name”
4. Return the pointer to the allocated object

Thus, xcalloc allocates memory for the object, but also create internal data structure in MLD Library so that MLD can keep track of the newly allocated object

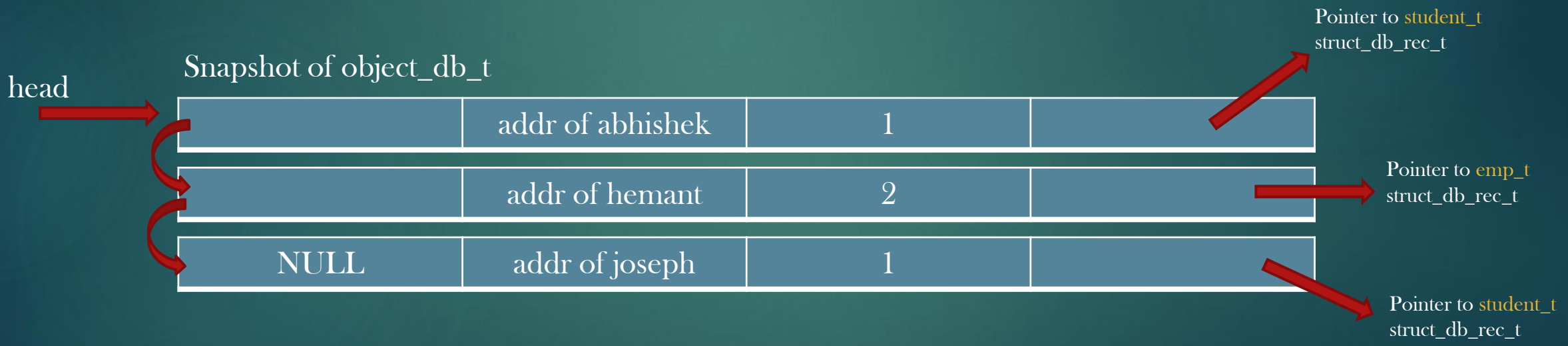
Project Development Phase : 2
Design and Implement Object database

```
typedef struct _object_db_rec_{

    object_db_rec_t *next;
    void *ptr;  /*Key*/
    unsigned int units;
    struct_db_rec_t *struct_rec;
} object_db_rec_t;
```

Let us suppose, the application create three objects :

```
student_t *abhishek = xcalloc(object_db, "student_t", 1);
emp_t *hemant = xcalloc(object_db, “emp_t”, 2);
student_t *joseph = xcalloc(object_db, "student_t", 1);
```

```
typedef struct _object_db_{
    struct_db_t *struct_db;
    object_db_rec_t *head;
    unsigned int count;
} object_db_t;
```



Let us Discuss the implementation step by step !

Project Development Phase : 2
Design and Implement Object database

Exercise :

- Implement corresponding xfree() function :

```
void
xfree(object_db_t *obj_db, void *ptr);
```

- Implement dumping function to dump object databases

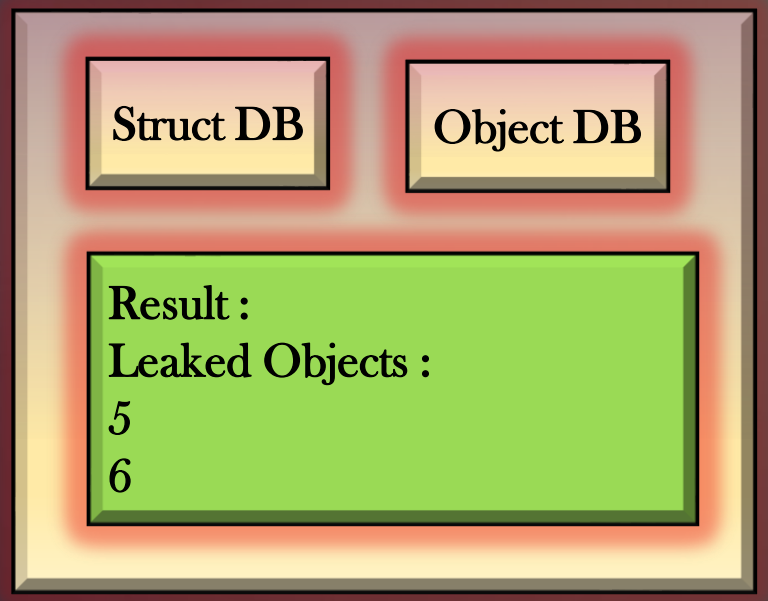
```
void
print_object_rec(object_db_rec_t *obj_rec);

void
print_object_db(object_db_t *object_db);
```

Summary

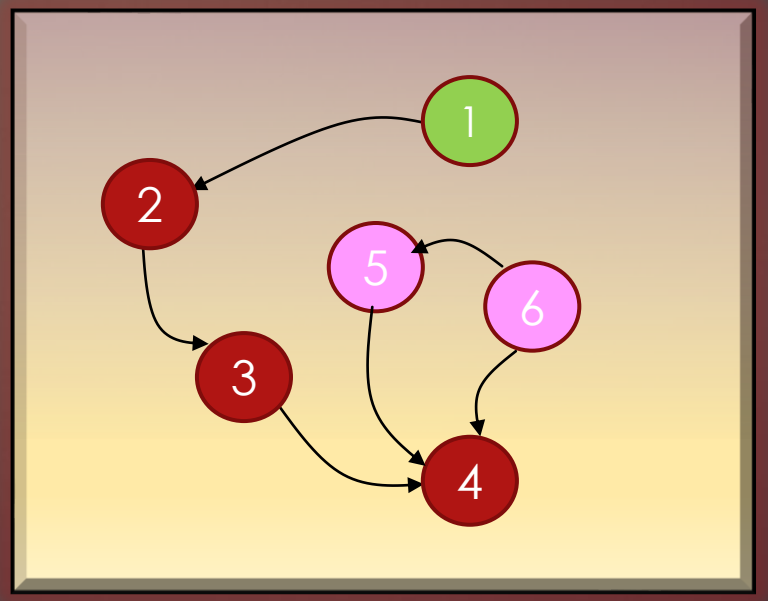
- MLD Library maintains two databases : Structure Database, and Object database
- Application register all its structures with MLD library at the time of initialization. This is one time process.
- Structure database should be read-only database, it need not be modified ever once populated
- Whenever the application *xcalloc* an object, object record entry is inserted in object database
- Object database keeps track of what all objects are being used by the application
- Now, that MLD has all the information it required to manipulate application objects, MLD library can do whatever it wants to do with application objects !

Project Development Phase : 3
Memory Leak Detection Algorithm
Goals

- The purpose of MLD library is to process object database, with the help of structure database, and find Leaked application objects and report them
- We need to implement memory leak detection algorithm in MLD library to accomplish this goal



MLD Library



Application Data structures

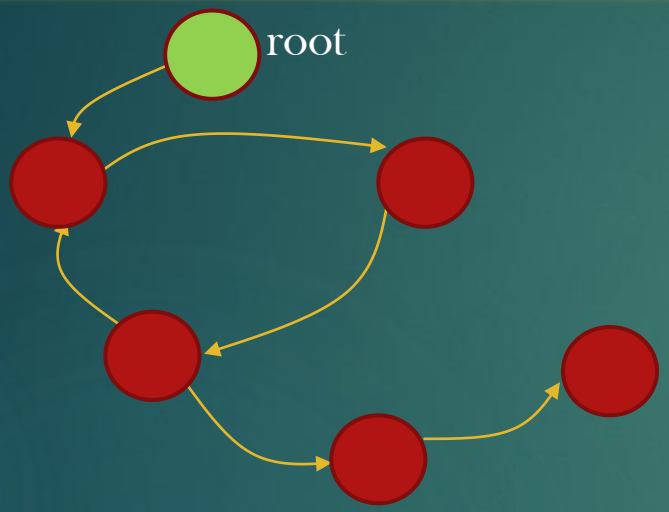
Project Development Phase : 3
Memory Leak Detection Algorithm
Application Data structures as Disjoint set of Graphs

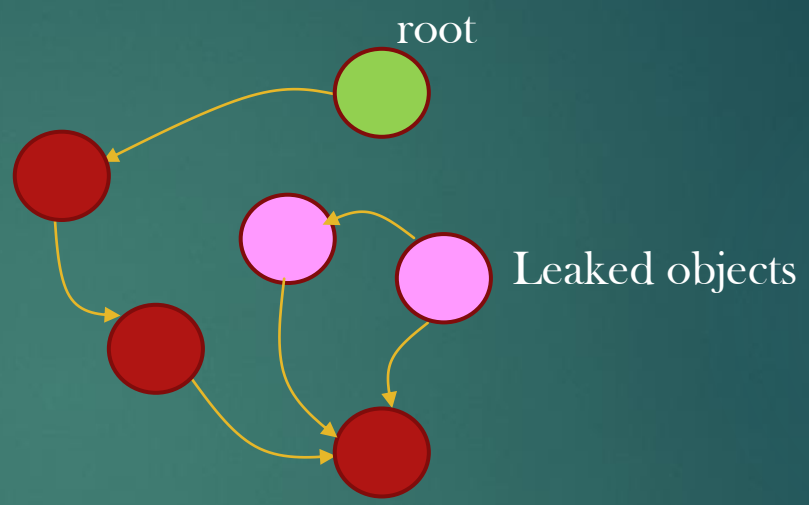
- Leaked objects are those objects which are not reachable from any other objects
- Finding the set of leaked objects is a graph problem

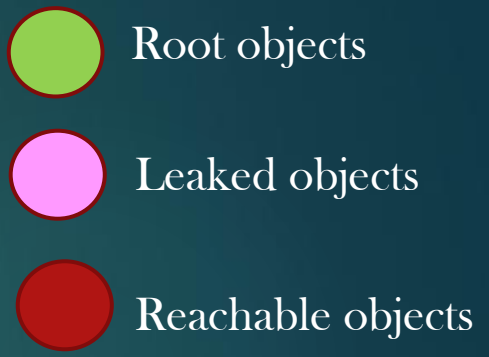
  Given a graph of nodes and edges, find all the nodes which are not reachable from any other nodes

- Application objects have references to one another, overall, all application objects combined take the shape of a graph
- The graph can be disjoint – it can be set of isolated graphs
- Each Isolated Individual graph has a special node called root of the graph
- You application Data structures always takes the *shape of disjoint set of graphs*

Project Development Phase : 3
Memory Leak Detection Algorithm
Leaked and Reachable Objects



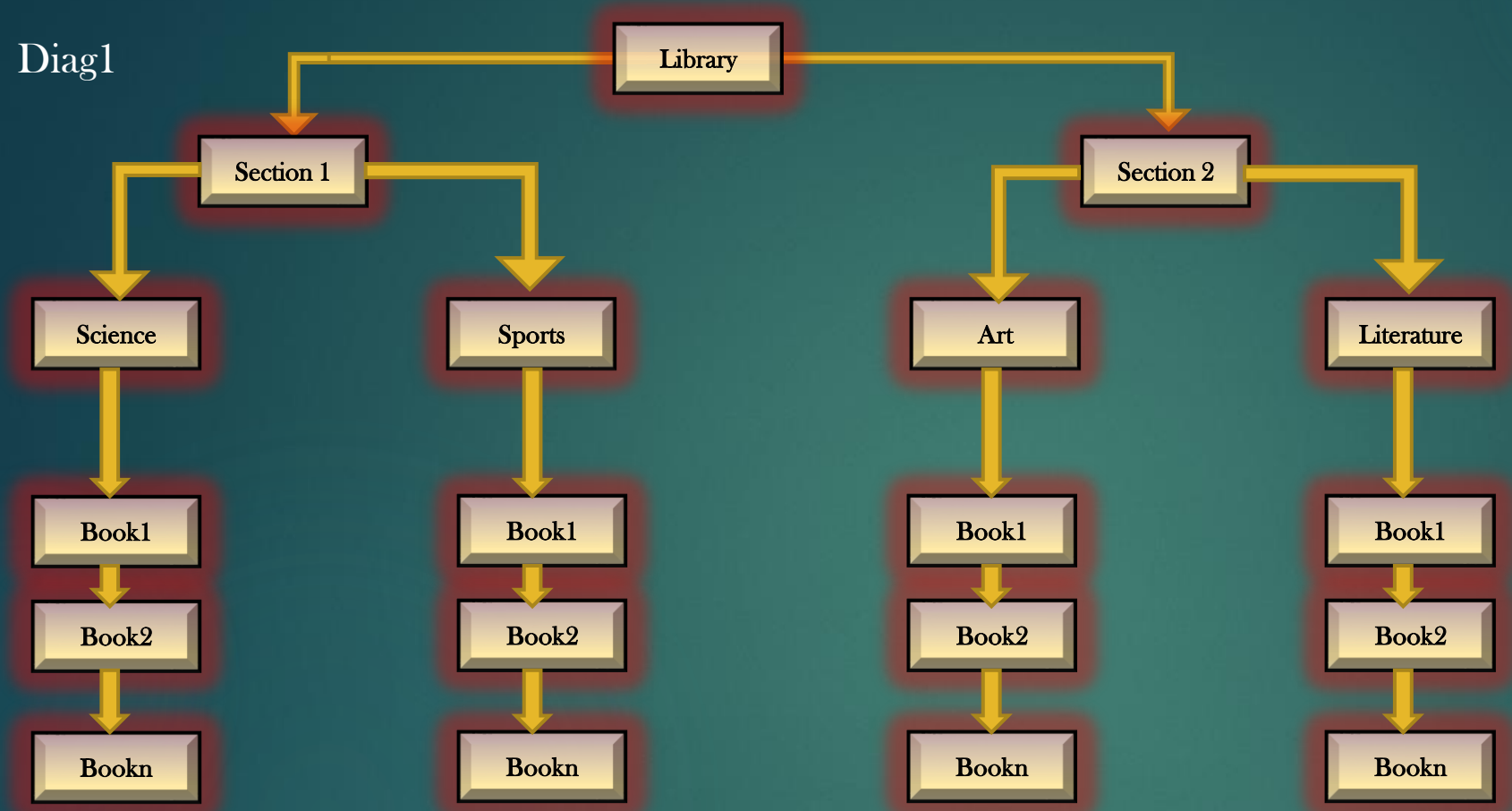




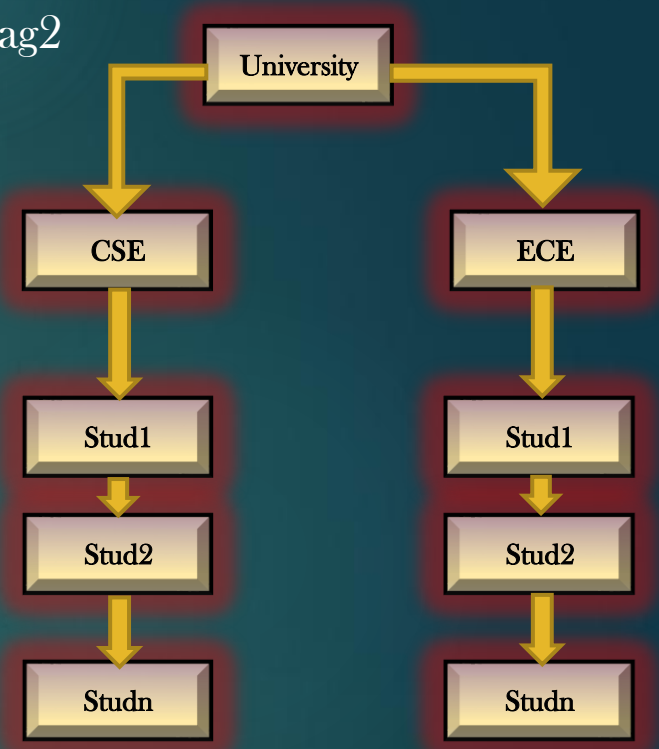
- Root objects are usually global Or static objects maintain by your applications
- Every other malloc'd objects MUST be reachable from at-least one root object
- Malloc'd object which are not reachable are leaked objects
- Assignment on designing data structures and verify that application data structures always take the form of disjoint set of graphs with root nodes

Project : Design and Implement Memory Leak Detector

Diag1



Diag2



- The new references are formed or destroyed as students borrows or returns the books to library
- At any point of time, the overall data structure is Directed Cyclic Graphs

Project Development Phase : 3
Memory Leak Detection Algorithm
Root Objects

- The Application has to tell MLD library the set of all root objects
- MLD library must provide an API using which an application can register its root objects
- Application can create root objects in two ways :

```
emp_t  emp ;    /* Simply creating a Global root object */

void mld_register_global_object_as_root (object_db_t *object_db,
                                         void *objptr,
                                         char *struct_name,
                                         unsigned int units);
```

Create a new object db record entry
In object db of MLD library, mark it as root

```
emp_t *emp = xcalloc(object_db, "emp_t", 1);  /* Dynamic Root Object */

mld_set_dynamic_object_as_root(object_db_t *object_db, void *obj_ptr) ;
```

Search an existing object db record entry
In object db of MLD library,
mark it as root

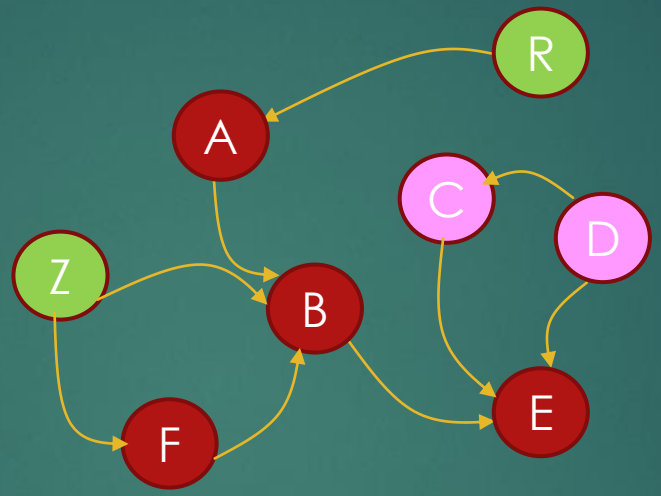
Project Development Phase : 3
Memory Leak Detection Algorithm
Root Objects

- The Memory leak detection algorithm begins from root objects
- Global objects are referenced by global variables in application, so, Global Objects cannot be leaked
- In most cases, Global objects are also the root objects
- Our MLD Library assumes dynamic root objects of the application are also never leaked by the application. If DRO are leaked by the application, our MLD algorithm will not report it since it starts Memory leak detection algorithm from root object assuming root objects are always reachable
- And it make sense, you want to start your journey and you need to cover 100 stations starting from station 1 where you are already present. Its not possible that you wont reach station 1 starting from station 1 (paradox ! :p)
  That's why MLD library assumes root objects (Dynamic or Global are always reachable)
- Let us now see how MLD algorithm in action . . .

Project Development Phase : 3
Memory Leak Detection Algorithm – Dry Run

Object Database

Object	Is Visited	Is Root
A	0	0
B	0	0
C	0	0
D	0	0
E	0	0
R	0	1
F	0	0
Z	0	1



Root objects

Leaked objects

Reachable objects

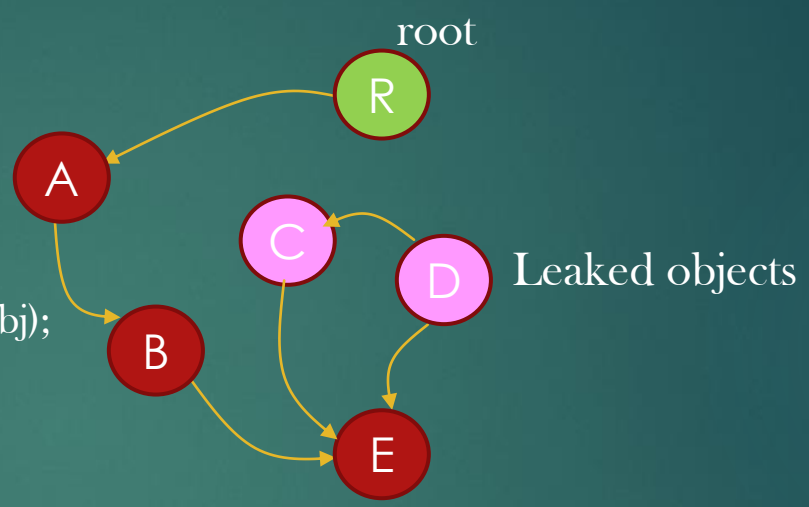
Analysis :

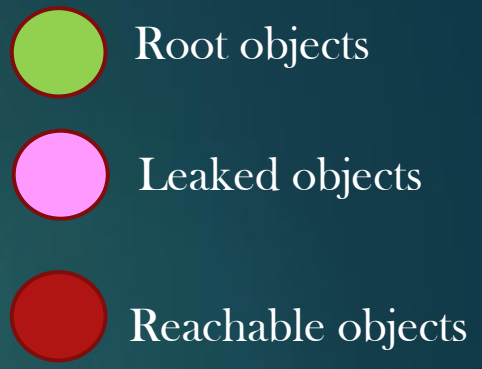
1. MLD algorithm is recursive
2. MLD algorithm is basically a DFS algorithm
3. *is_visited* flag is used to avoid loops

Project Development Phase : 3
Memory Leak Detection Algorithm : Level 1 Pseudocode

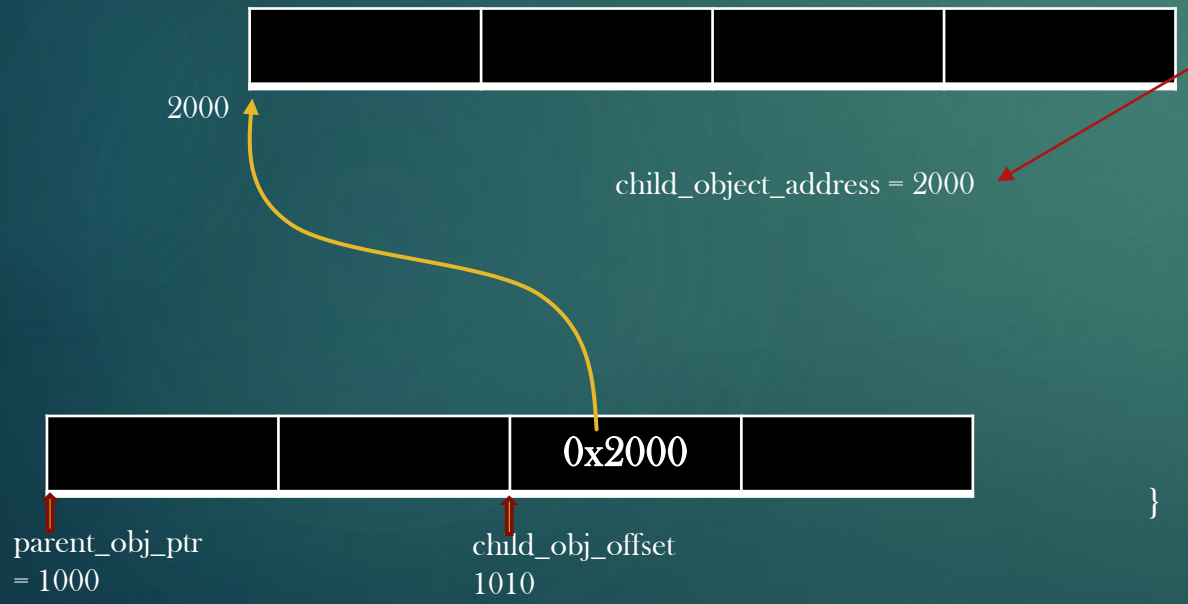
```
init_mld_algorithm(object_db);
root_obj = get_first_root_object(object_db)
while(root_obj){

        if(root_obj already visited){
                root_obj = get_next_root_object(object_db, root_obj);
                continue;
        }

        mark root_obj as visited

        /*mark all  objects reachable from root_obj as visited */
        mld_explore_objects_recursively(object_db, root_obj);

        root_obj = get_next_root_object(object_db, root_obj)
}
```

Print “All objects in object db which are not visited. Those are Leaked objects”





Project Development Phase : 3
Memory Leak Detection Algorithm :
Level 2 Pseudocode



Level 2 : Pseudocode

```
mld_explore_objects_recursively (object_rec) {

        char *parent_obj_ptr = object_rec->ptr;
        void *child_object_address = NULL;

        for all Fields F in object_rec->struct_rec {

                if F->dtype != OBJ_PTR  continue;

                child_obj_offset = parent_obj_ptr + F->offset;
                 memcpy(&child_object_address, child_obj_offset, sizeof(void *));

                 if(!child_object_address) continue;
                 child_object_rec = object_db_look_up(object_db, child_object_address);
                 if(!child_object_rec->is_visited){
                        mark child_object_rec as visited
                        mld_explore_objects_recursively(child_object_rec);
                }
                else{
                        continue; /*Do nothing, explore next child object*/
                }
        }
}
```

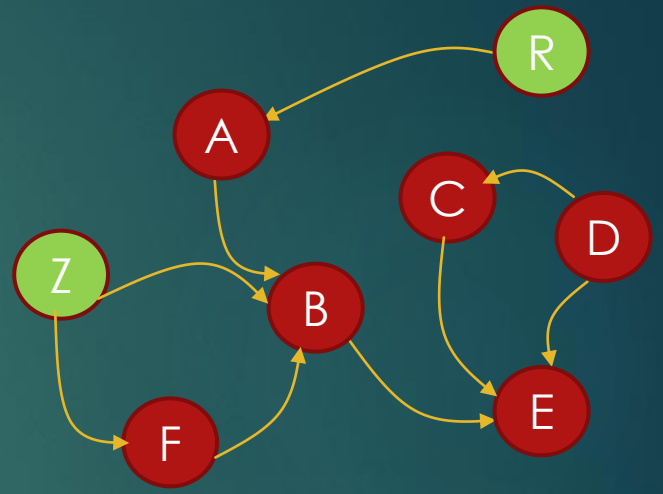
Project : Design and Implement Memory Leak Detector

```
R {
  A *a;
}

A {
   B *b;
}

Z {
   B *b;
   F *f;
}

B {
  void *e;
}

E {
  No OBJ_PTR fields
}

C {
   E *e;
}

D {
  C *c;
  E *e;
}
```

Assumptions :

Root Objects to be selected in the following order : R , Z
Child objects to be selected with lower alphabet first

Run your MLD algorithm on this input graph and determine the Leaked objects



Decoupling MLD Library from Application

- One of the Properties of Libraries is that they should have loose coupling with the application
- Application must just *#include* the public hdr files provided by the library and must use the public functions provided by the library through the imported header file
- Our MLD library, at this point of time, is tightly coupled with application which is not a good thing
  - Our application need to create object database and structure database explicitly
  - Our application need to maintain pointers to object database and structure database so that it can pass these pointers to xcalloc and xfree functions
  - Our application has direct access to structure database and object databases which must be sole proprietary assets of MLD library and application has no business to have direct access to these databases other than through MLD public functions/APIs
- Decoupling of MLD library with application will help application to stay naive and unaware of internals of MLD library
- Applications must know only *WHAT* aspects of the library and not *HOW* the library achieve its goals
- This section, precisely, is not related to MLD library, but in general applies to any library
- We will do this section step-wise explained through a simple doc. Just follow the steps explained in the doc and modify your MLD code accordingly. No Video Lecture Videos for this section of the course

Enhancement 2 : Handling VOID pointers

- Surprise Surprise Surprise !!
- Our MLD Library automatically support void *objects !
- I leave it to your analysis as to why it supports void * objects (you have to make minimal code changes however !)

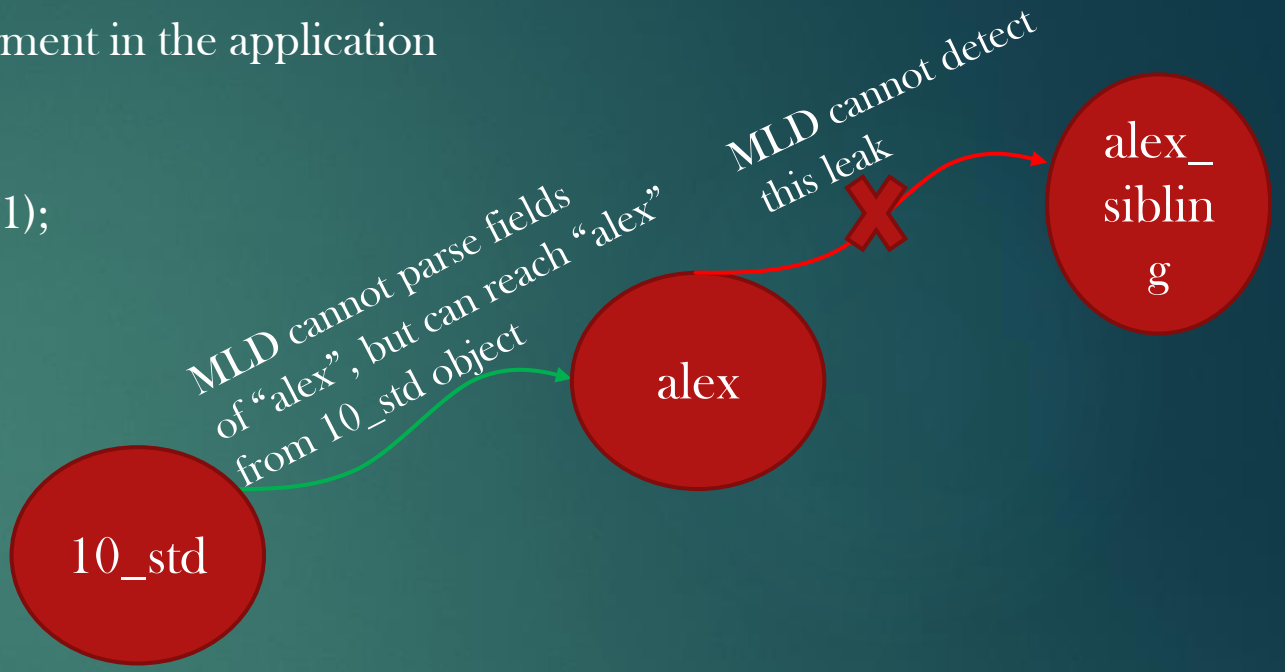
Enhancement 3 : Handling Pointers to Pointers

- Does our MLD library handles * * or * * * type pointer members
- No !
- So, lets see what does it take to support multi-level indirections !

Handling VOID pointers

- Consider the Developer has written the following code segment in the application

```
student_t *alex =  xcalloc(object_db, "student_t", 1);
student_t *alex_sibling =  xcalloc(object_db, "student_t", 1);
alex->sibling = alex_sibling;

class *10_std = xcalloc(object_db, “class_t", 1);
10_std->class_monitor = alex;


typedef struct class_ {
        . . .
        . . .
        void *class_monitor;
} class_t
```



Given a pointer to 10_std object, We cant access the Fields of object 10_std->class_monitor

MLD library only knows that 10_std->class_monitor is void * Object and it has no structure information about this

Handling VOID pointers

1. Registration of void* fields

```
FIELD_INFO (class_t, class_monitor, VOID_PTR,   0)
```

2. Add VOID_PTR enumeration

3. Make changes in mld_explore_objects_recursively() fn to handle the field VOID_PTR

```
if (current field F of current object O is of data_type VOID_PTR)
        Mark the object pointed to by the field F as Visited
        DO NOT CALL mld_explore_objects_recursively
```

Handling Multi-Level Indirections

Project Analysis

- So, let us Analyze this project now

- We will discuss the interview question as I stated in the beginning of the course –

  Why C/C++ programming languages cannot have inbuilt Garbage collector like Java have ?

- Now that you have implemented the MLD library which is based on the algorithm to find *reachability of objects,* you must analyse the MLD algorithm short-comings and limitations

- There are many tricks in which one can always write a C/C++ code to fool MLD library to report false leaks

- We will see these tricks are easy to implement in C/C++ program but not in JAVA by virtue of the fact that JAVA is a pure object oriented language

- This section of the course is most important, preparing you to answer the interview question

- Let us discuss some scenarios where MLD library fails miserable by C programs and not by Java programs

Project Analysis

- Case 1 : Storing the pointer to non-pointer data types !
  - At the end of day, pointer are numerical numbers which represent the address in process virtual address space, and therefore can always be stored like you store the *age* of employee

```
struct emp_t {
        char name[32];
        . . .
        . . .
        unsigned int designation;
};

struct des_t {
        char name[32];
        int job_code;
        int salary_range;
};
```

```
struct emp_t   *emp = xcalloc(. . . );

struct des_t *des = xcalloc(. . .);

emp->designation = (unsigned int)des;
```

MLD library will report is as a leak (false alarm)

JAVA do not allow it !

Project Analysis

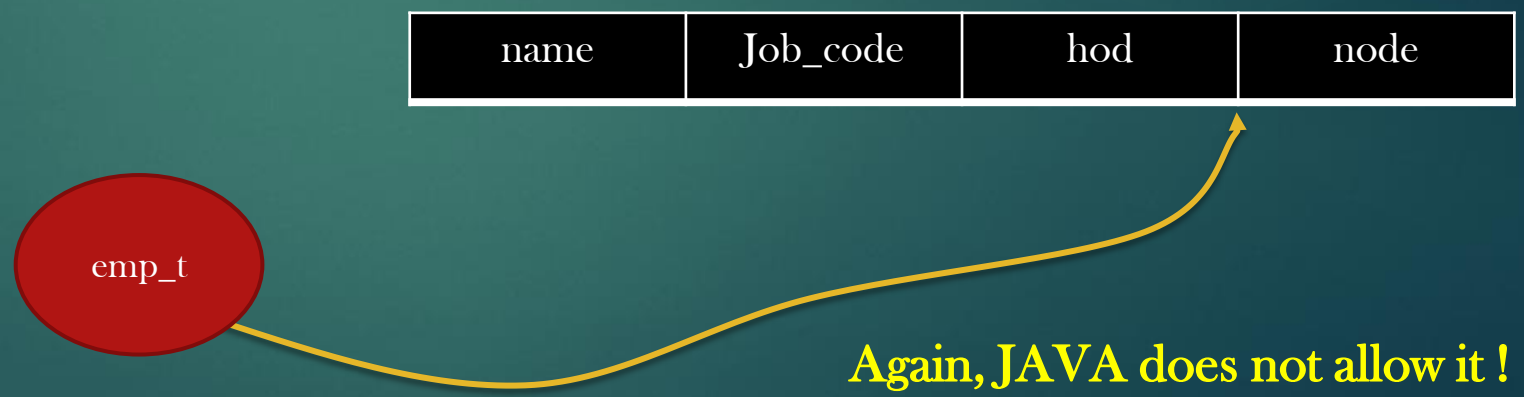
- Case 2 : Indirect reference to objects
  - This happens when one pointer points to memory location not returned by calloc

```
struct emp_t {
        char name[32];
        . . .
        . . .
        stuct list_node_t *node;
}
```

```
struct des_t {
        char name[32];
        int job_code;
        emp_t *hod;
        struct list_node node;
};
```

```
struct emp_t  *emp = xcalloc(. . . );

struct des_t *des = xcalloc(. . .);

emp->node = &des->node;
```

MLD Library cannot parse fields of des_t objects from emp_t object



Again, JAVA does not allow it !

Project Analysis

- Case 3 : Embedded Objects
  - Allowed by C/C++ but not by Java

```
struct des_t {
        char name[32];
        int job_code;
};


struct emp_t {
        char name[32];
        struct des_t des;
};
```

C structure

```
struct emp_t {
        char name[32];
        struct des_t des;
};


struct emp_t {
        char name[32];
        struct des_t *des;
};
```

Java class

```
class Emp{
        char name[32];
        Des des;
};
```

Because Java do not allow embedded objects, you cannot have a reference pointing to Embedded objects in Java

Project Analysis

- Case 4 : Unions
  - Our MLD library cannot handles unions ! ☹
    - Unions don't have fixed size. Size of the unions = size of the largest structure under union

```
union profession {
        struct doctor{
                . . .
                . . .
        } dctr;
        struct engg{
                . . .
                . . .
        }engg;
};
```

- There are no unions in JAVA

Conclusion

- If you write a C/C++ program following pure object oriented notion as in case of JAVA, then

MLD Library = Java Garbage collector

- But, doing so places so many constraint on programmer to use C/C++. These languages provide developer to harness the direct access to Memory address (whereas Java do not), so why not make use of it ?

- C has not been designed to be used in a object oriented way.

- Note that – Developing a non object oriented software in object oriented way is disastrous. OOPs is not always great !

Thus, C/C++ developers has not chosen and never will to implement
MLD like library in standard C/C++ libs

Thank you for enrolling into this short sweet course ! ☺