data_tructures(&algorithms, lecture10)

Doan Trung Tung, PhD – University of Greenwich (Vietnam)

Plan

Memory Pool 01

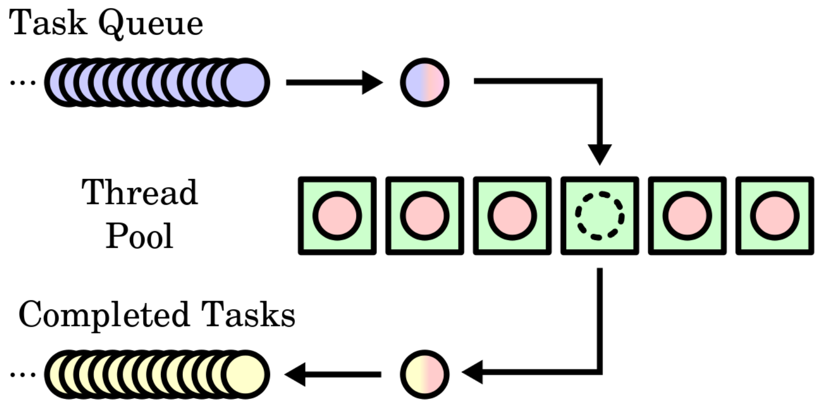
Synchronize processes 02

Design Patterns by C 03

Memory Pool

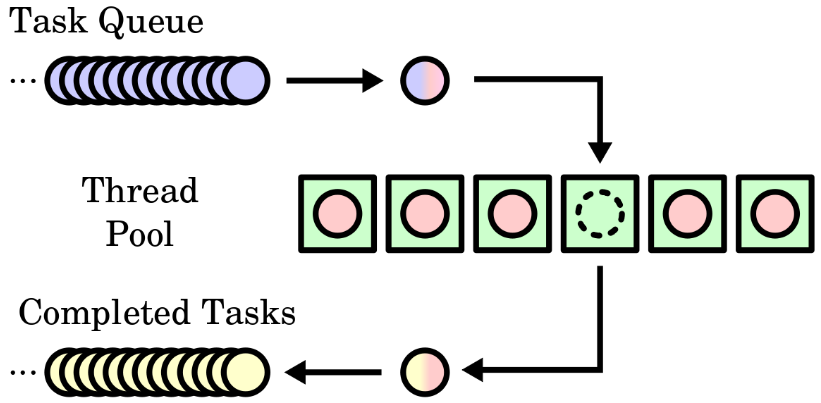
What is a pool?

- In computer science, a pool is a collection of resources that are kept ready to use, rather than acquired on use and released afterwards.
- Resources can refer to any kind of system resources such as file handles, memories or sockets, etc.



What is a pool?

❖ A pool client requests a resource from the pool and performs desired operations on the returned resource. When the client finishes its use of the resource, it is returned to the pool rather than released and lost.



Why memory pool?

- Dynamic memory allocation is often inefficient in terms of time and/or space.
  - Most memory allocation algorithms store some form of information with each memory block that will take up more space in a block that is being used by the program.
  - When small objects are dynamically allocated (i.e int) the system algorithm will reserve space for the header fields as well, and we end up with a 50% waste of memory.

Why memory pool?

- Dynamic memory allocation is often inefficient in terms of time and/or space.
  - In larger systems, the memory overhead is not as big of a problem (compared to the amount of time it would take to work around it), and thus is often ignored.
  - However, there are situations where many allocations and/or deallocations of smaller objects are taking place as part of a time-critical algorithm, and in these situations, the system-supplied memory allocator is often too slow.

Why memory pool?

- Dynamic memory allocation is often inefficient in terms of time and/or space.
  - Using Pools gives you more control over how memory is used in your program. For example, you could have a situation where you want to allocate a bunch of small objects at one point, and then reach a point in your program where none of them are needed any more.
  - Using pool interfaces, you can choose to run their destructors or just drop them off into oblivion; the pool interface will guarantee that there are no system memory leaks.

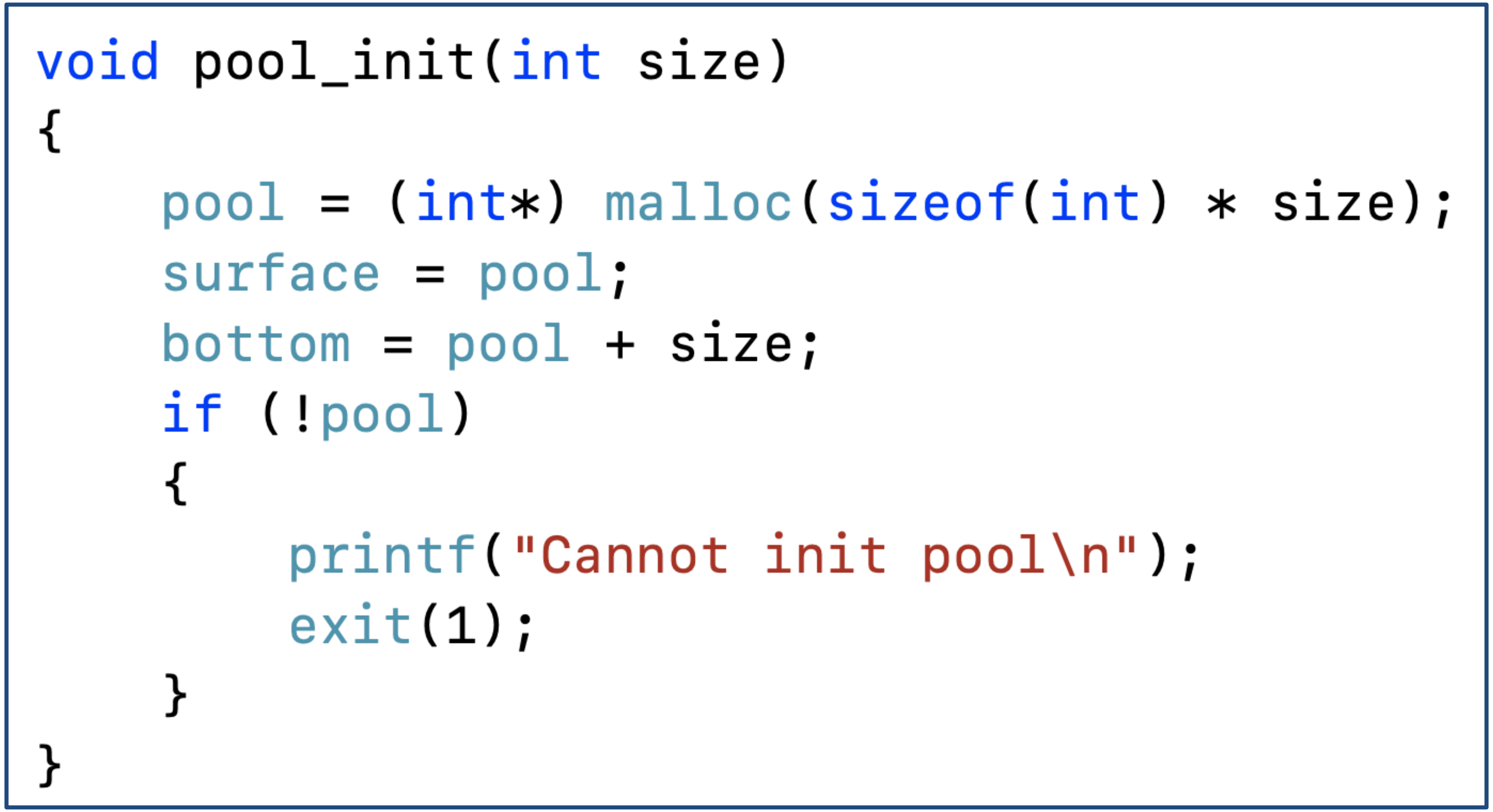
General memory pool?

- No, that will slow-down system performance
- Memory pool is specific by scenario and its problem
- Sill, there are libraries for memory pool (boost) but you should write your own memory pool for your own problem

Memory Pool scenario #1

- A large number of objects are heavily allocated
- No deallocated one by one
- Pool will free all object at once
- Advantages:
  - Fast
  - Simple
  - Allocation for series of objects
- Disadvantages:
  - No reused allocated block

Memory Pool scenario #1



```c
void pool_init(int size)
{
    pool = (int*) malloc(sizeof(int) * size);
    surface = pool;
    bottom = pool + size;
    if (!pool)
    {
        printf("Cannot init pool\n");
        exit(1);
    }
}
```

[11]

Memory Pool scenario #1

```c
int* p_malloc(int size)
{
    if (!is_shallow(size))
    {
        int* curr = surface;
        surface += size;
        return curr;
    }
    else return NULL;
}
```

```c
void pool_free(void)
{
    free(pool);
}

int is_shallow(int size)
{
    return bottom - surface < size;
}
```

[12]

Memory Pool scenario #1

- Re-write code so that Pool can allocate memory for any type of objects (currently integers)
- Test new Pool

Memory Pool scenario #2

- A large number of objects are heavily allocated
- Pool can free one object to reuse allocated block
  - But should be fast
- Pool can free all object at once
- Advantages:
  - Fast & Simple
- Disadvantages:
  - No allocation for series of objects
  - Need time to handle fragmentation or full of allocation

Memory Pool scenario #2

```
typedef struct
{
    u_int element_size; // size of each element
    u_int pool_size;    // size of the whole pool (max number of elements)
    u_int n_avails;     // current number of allocated elements
    u_int curr;         // current available position to allocate
    ul_int* freed;      // address of elements to be freed
    u_int n_freed;      // number of elements to be freed
    ul_int* avails;     // mark available or not when allocating
    void* pointer;      // pointer to the blocks of memory reserved by the pool
} memory_pool;
```

Memory Pool scenario #2

```
 // create a pool of memory
memory_pool* create_pool(u_int element_size, u_int pool_size);
// allocate memory from the pool
void* pool_alloc(memory_pool *pool);
// free memory pointed by p from the pool
void pool_free(memory_pool *pool, void* p);
// find next available position to allocate
int find_next(memory_pool *pool);
// clean the pool to reuse freed memory
void clean_pool(memory_pool* pool);
// delete no longer use pool
void delete_pool(memory_pool* pool);
// chekc if pool is full
int is_full(memory_pool* pool);
// clean pool at one specific address
void clean_at(memory_pool* pool, ul_int addr);
```

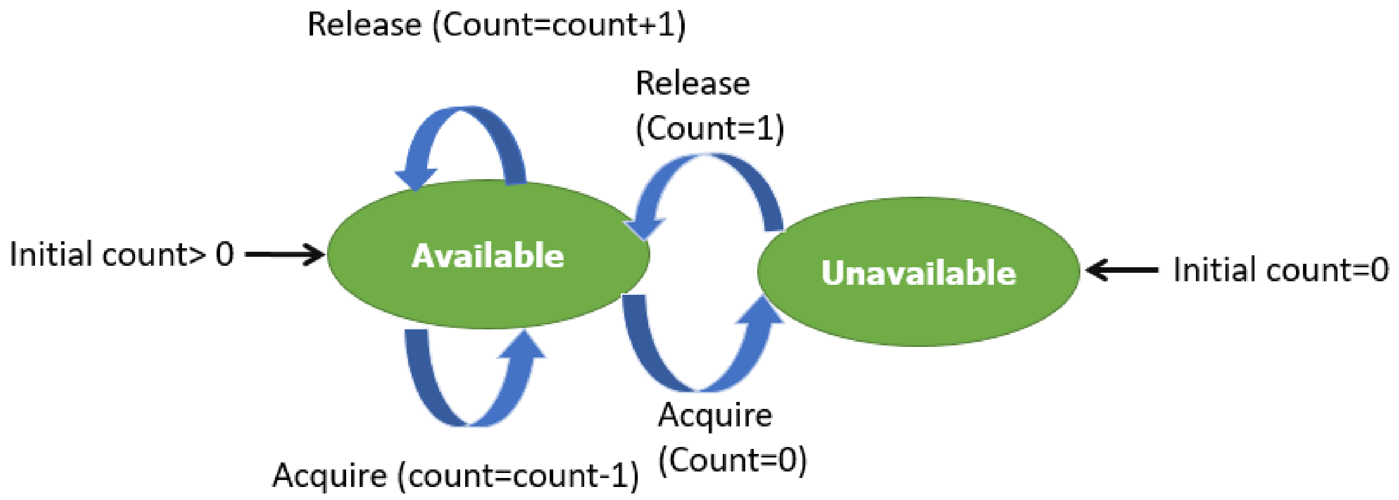
Synchronization

Semaphore

- Semaphore is a data handling technique which is very useful in process synchronization and multithreading
- Semaphore is simply a variable that is non-negative and shared between threads.
- A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread.
- It uses two atomic operations:
  - wait
  - signal

Semaphore

- How semaphore works



Semaphore

❖ Semaphore vs Mutex

Semaphore	**Mutex**
Signaling	Locking
Value can be modified by different threads	Only thread that lock mutex can unlock it
Synchronize multiple resources	Synchronize one resource
Multiple threads run in parallel	Multiple threads but not in parallel
Binary semaphore is quite similar to mutex but not identical	

POSIX Semaphore in C

- Include `semaphore.h` header file
- Compile the code by linking with `-lpthread -lrt`
- Functions:
  - sem_init: initialize a semaphore with positive value
  - sem_wait: block thread if value is 0, decrease if value is positive
  - sem_post: increase semaphore value (may exceed initial value)
  - sem_getvalue: get value of semaphore
  - sem_destroy: destroy semaphore, threads that are waiting will continue

Example: Threads pool

- Semaphore is normally used to control resources pool in multithreading
- Scenario: A server uses threads pool to handle request from multiple users.
  - When a request arrives, a thread is get out of pool to handle it
  - After finishing a request, a thread goes back to pool
  - Semaphore is used to control threads pool

Example: Thread pool

- Thread pool

```
typedef struct
{
    int *threads;
    int *available;
    int thread_ptr;
    int size;
} thread_pool;

thread_pool* init_pool(const int size);
int get_thread(thread_pool** pool);
void free_thread(thread_pool** pool, const int pos);
void clean_pool(thread_pool** pool);
void dump(thread_pool *pool, int req, int tid);
```

Example: Thread pool

❖ Getting a thread out of pool: handle_request

```
sem_wait(&semaphore); // wait for signal
int tid = get_thread(&pool);

if (tid == NO_THREAD)
{
    perror("Something wrong with thread pool!\n");
    return NULL;
}
printf("Thread %d awakes to handle request %d\n", tid, req);

sleep(rand() % HANDLE_TIME); // simulate time to handle request

printf("Thread %d finishes. Go back to pool...\n", tid);
free_thread(&pool, tid);

sem_post(&semaphore); // signal that one resource is released
```

Example: Thread pool

- Getting a thread out of pool: main

```
sem_init(&semaphore, THREAD_ONLY, POOL_SIZE);
pool = init_pool(POOL_SIZE);

for (int rq_id = 0; rq_id < NREQUESTS; rq_id++)
{
    pthread_create(&req_threads[rq_id], NULL,
                   handle_request, (void*)rq_id);
    // simulate waiting time for new request
    sleep(rand() % REQUEST_RATE);
}

// waiting other threads to finish
for (int rq_id = 0; rq_id < NREQUESTS; rq_id++)
{
    pthread_join(req_threads[rq_id], NULL);
}

clean_pool(&pool);
```

Design Patterns

Design Pattern: Definition

- Wikipedia:
  - "A design pattern is a general repeatable solution to a commonly occurring problem in software design"
- Christopher Alexander:
  - "Each pattern describes a problem which *occurs over and over again* in our environment, and then describes *the core of the solution* to that problem, in such a way that you can *reuse* this solution a million times over, without ever doing it the same way twice" (GoF,1995)

Elements of a pattern

- Each pattern has 4 elements:
  - Name: describe in general what the pattern is or does
  - Problem: describe a general / specific scenario where the pattern can be applied
  - Solution: describe a general / specific solution for the problem
  - Consequences: describe pros / cons of applying pattern

Elements of a pattern

- Example:
  - Name: Singleton => something unique / single
  - Problem: Allow creating only one instance in the system (for consistent)
  - Solution: Using static (+ other OOP techniques)
  - Consequences: no global access, can be extended, etc.

Design Pattern: Classification

- Classical patterns: 23 patterns in 3 categories
  - Creational: deal with creating objects problems
  - Structural: deal with organizing objects in a system problems
  - Behavioral: deal with object behaviors problems
- Modern patterns:
  - MVC
  - N-layers
  - Single Page Application
  - etc.

Design Pattern & OOP

- Design Pattern uses many OOP characteristics
  - Constructor
  - Abstract
  - Inheritance
  - Overriding
  - Polymorphism

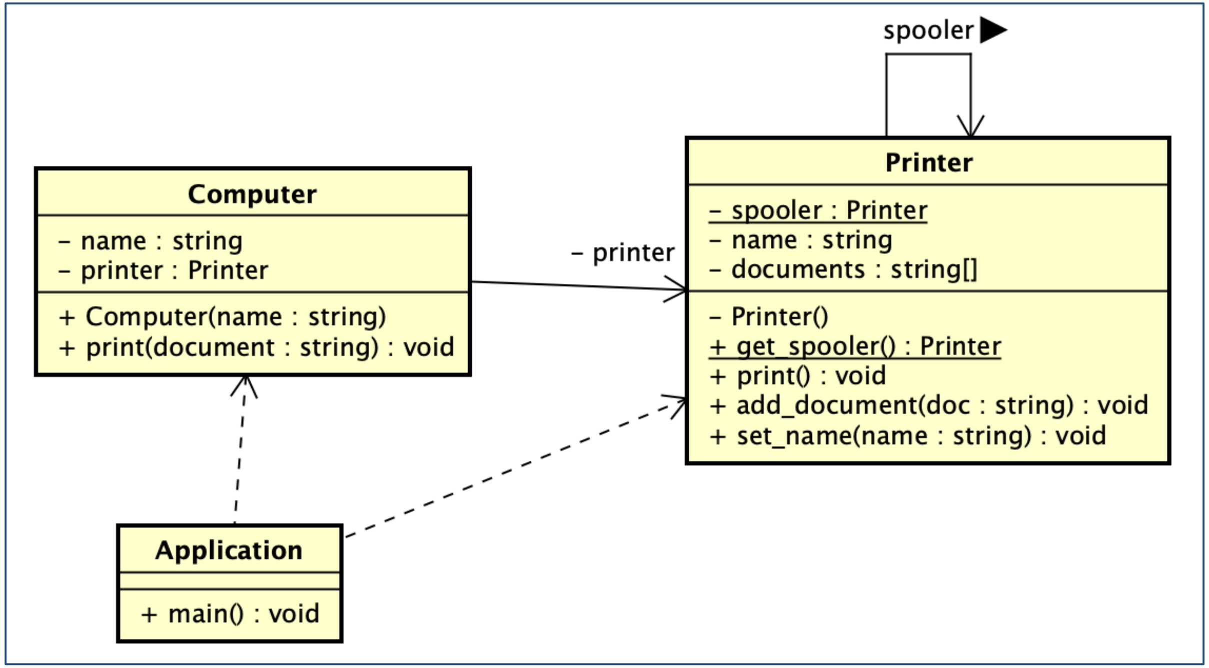
- Is there Design Pattern in C?

Pattern example: Singleton

- Intent: Ensure a class only has one instance, and provide a global point of access to it
- Motivation:
  - It's important for some classes to have exactly one instance. For example: there are many printers in a system, there should be only one printer spooler.
  - How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects

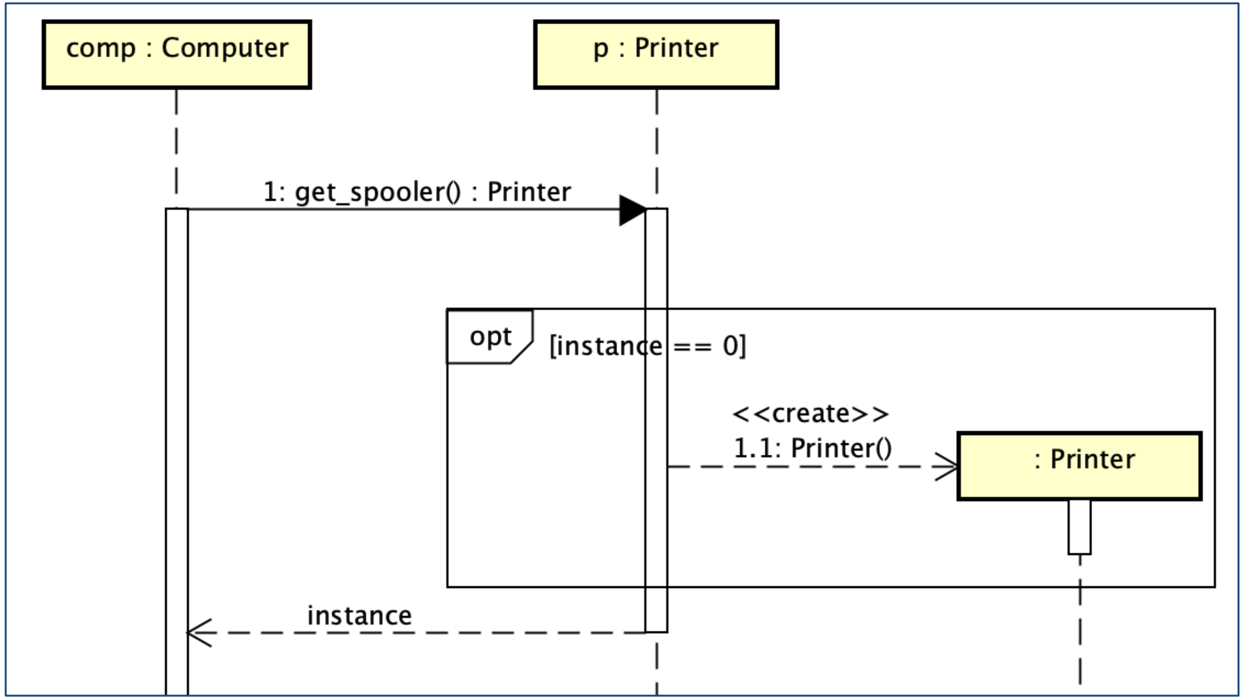
Pattern example: Singleton

- Solution



Pattern example: Singleton

- Solution



Singleton in C

❖ Using struct + static

```
typedef struct
{
    int n_pages;
    document queue[NDOCS];
    int front;
    int rear;
} printer;

printer* get_printer(void);
```

Abstraction
of printers

Function that return
a unique printer

Singleton in C

- Using struct + static

```
printer* get_printer(void)
{
    static printer* spooler = NULL;
    if (spooler == NULL)
    {
        spooler = (printer*) malloc(sizeof(printer));
        spooler->front = -1;
        spooler->rear = -1;
        spooler->n_pages = INK_PAGES;
    }
    return spooler;
}
```

Singleton in C

❖ Using “Singleton”

```
int main(int argc, const char * argv[])
{
    computer_print(COMP1, 500);
    computer_print(COMP2, 1200);
    return 0;
}

void computer_print(int computer, int pages)
{
    printer* spooler = get_printer();
    document doc = {computer, pages};
    add_document(spooler, doc);
    print(spooler);
}
```

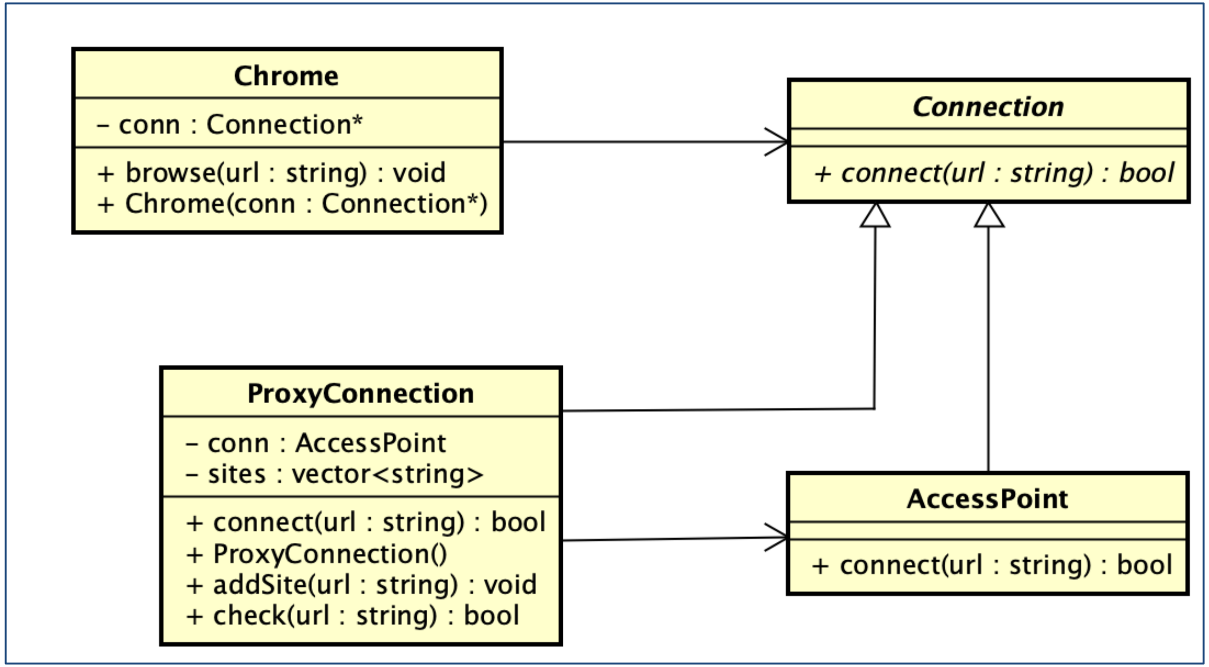
It will be called twice but each time refer to the unique spooler instead creating 2 instances

Pattern example: Proxy

- Intent: Provide a surrogate or placeholder for another object to control access to it.
- Motivation:
  - To defer the full cost of object creation and initialization until we actually need to use it => virtual proxy
  - To provides a local representative for an object in a different address space => remote proxy
  - To controls access to the original object => protection proxy

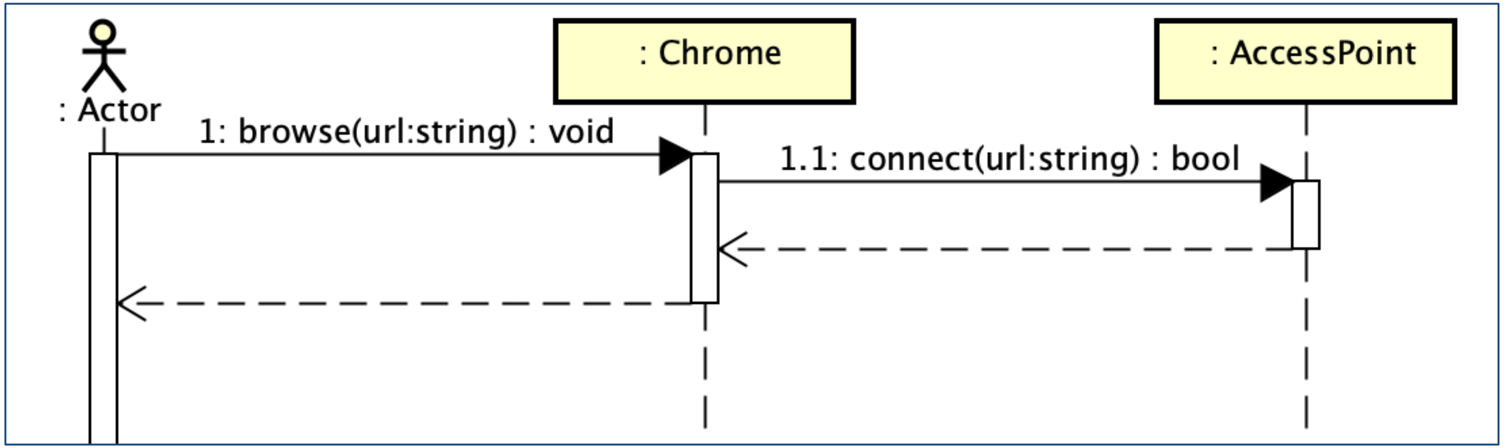
Pattern example: Proxy

❖ Solution


Chrome
– conn : Connection*
+ browse(url : string) : void
+ Chrome(conn : Connection*)
Connection
+ connect(url : string) : bool
ProxyConnection
– conn : AccessPoint
– sites : vector<string>
+ connect(url : string) : bool
+ ProxyConnection()
+ addSite(url : string) : void
+ check(url : string) : bool
AccessPoint
+ connect(url : string) : bool

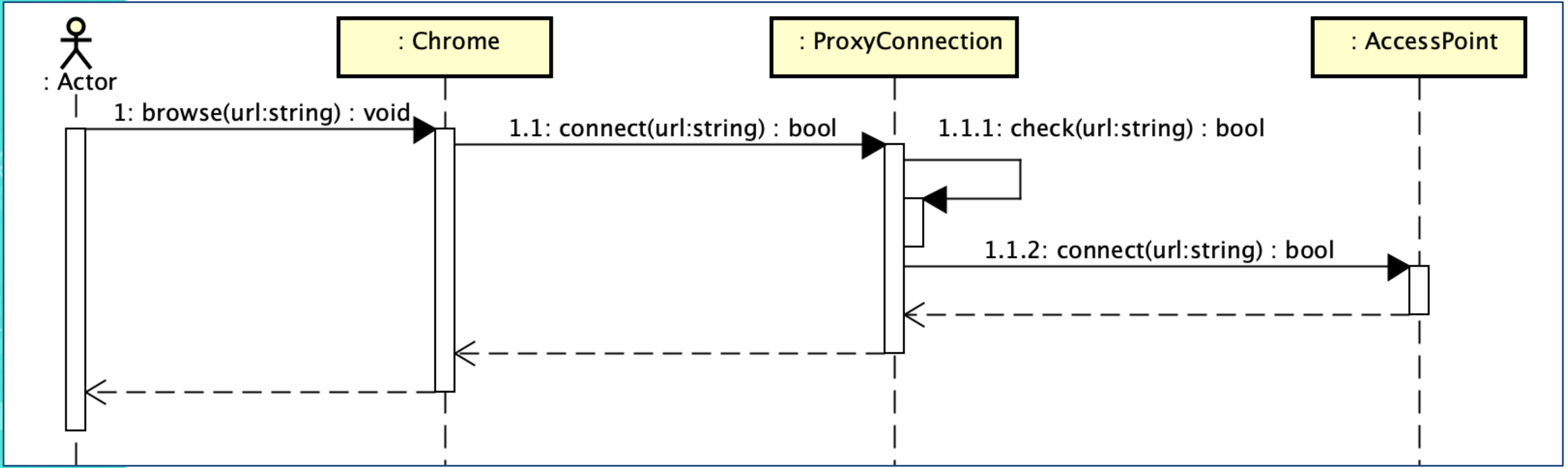
Pattern example: Proxy

❖ Solution



Pattern example: Proxy

❖ Solution


: Actor
: Chrome
: ProxyConnection
: AccessPoint
1: browse(url:string) : void
1.1: connect(url:string) : bool
1.1.1: check(url:string) : bool
1.1.2: connect(url:string) : bool

Proxy in C

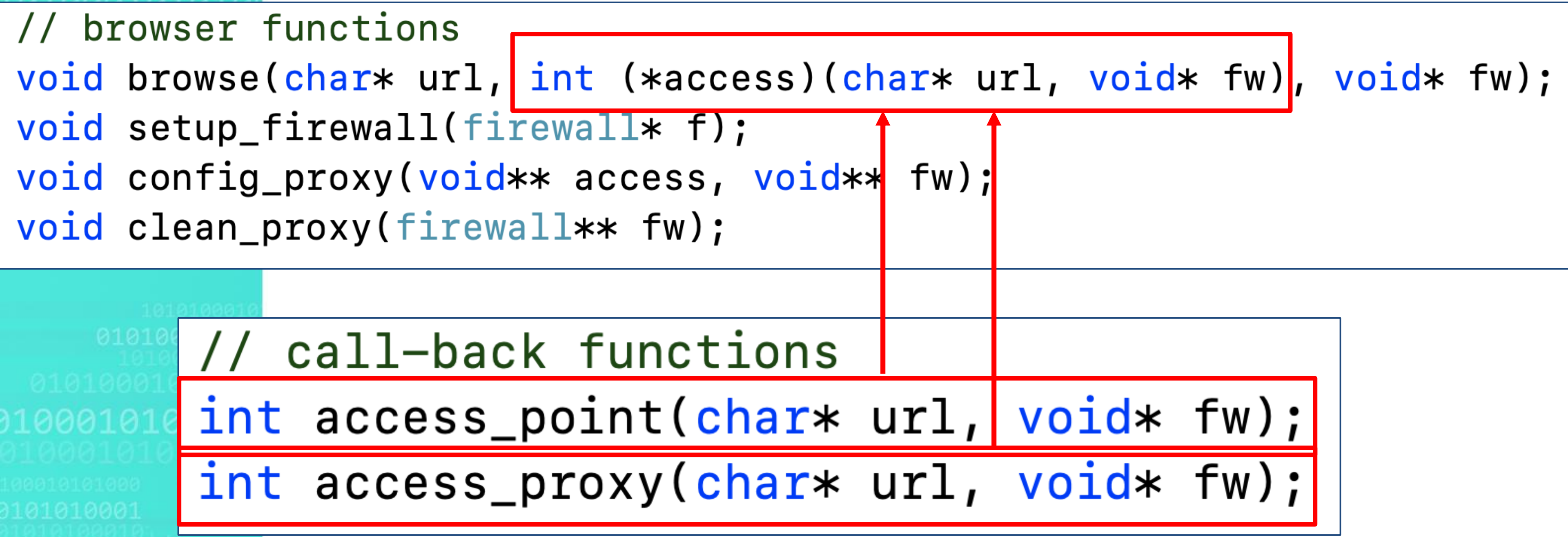
- Using: struct, callback function, void*

```
typedef struct
{
    int n_urls;
    char urls[MAX_URLS][URL_LEN];
} firewall;
```

```
// browser functions
void browse(char* url, int (*access)(char* url, void* fw), void* fw);
void setup_firewall(firewall* f);
void config_proxy(void** access, void** fw);
void clean_proxy(firewall** fw);
```

Proxy in C

- Using: struct, callback function, void*



```
// browser functions
void browse(char* url, int (*access)(char* url, void* fw), void* fw);
void setup_firewall(firewall* f);
void config_proxy(void** access, void** fw);
void clean_proxy(firewall** fw);
```

```
// call-back functions
int access_point(char* url, void* fw);
int access_proxy(char* url, void* fw);
```

Proxy in C

❖ Using: struct, callback function, void*

```
void browse(char* url, int (*access)(char* url, void* fw), void* fw)
{
    if (access(url, fw))
    {
        printf("[Chrome]: Connected to %s\n", url);
        printf("[Chrome]: Start browsing %s\n", url);
    }
    else
    {
        printf("[Chrome]: Cannot access to %s\n", url);
    }
}
```

Not a real function, will be defined later by call-back function

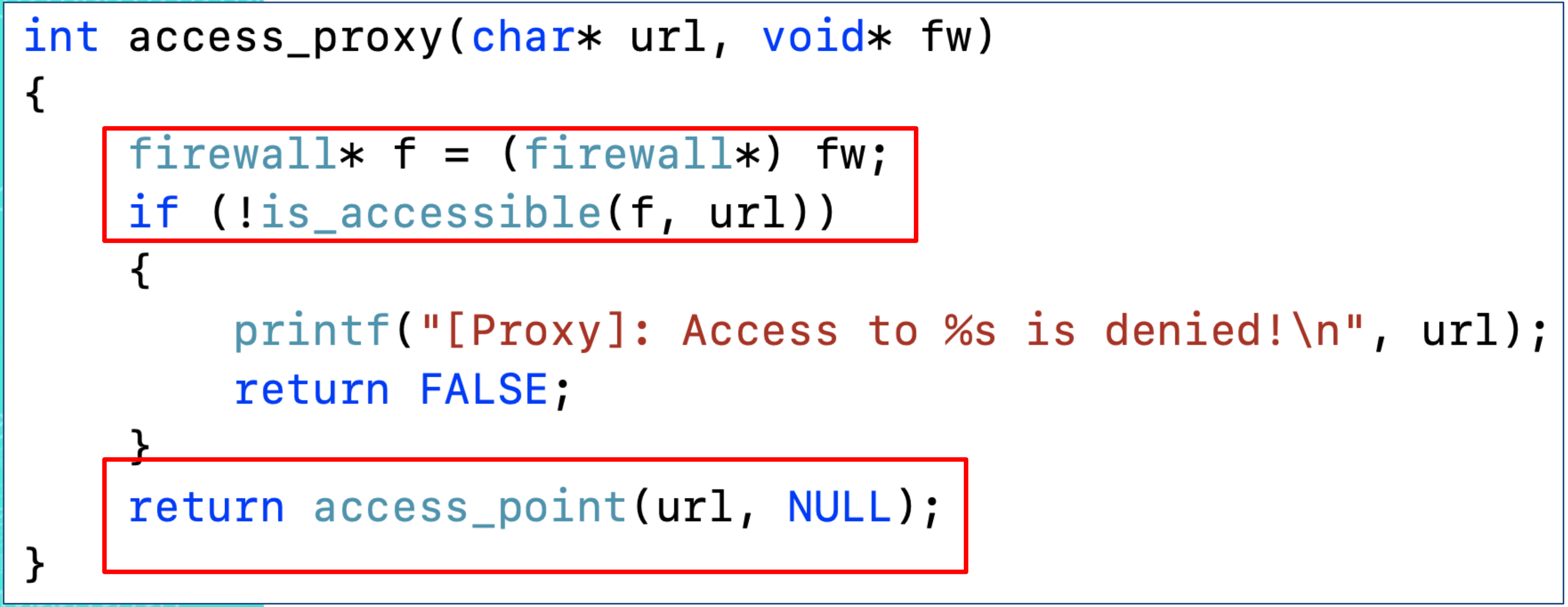
Proxy in C

- Using: struct, callback function, void*

```
int access_point(char* url, void* fw)
{
    if (!is_url(url))
    {
        printf("[AP]: Invalid ulr!\n");
        return FALSE;
    }
    printf("[AP]: Getting ip address for %s ... done!\n", url);
    return TRUE;
}
```

Proxy in C

- Using: struct, callback function, void*



```c
int access_proxy(char* url, void* fw)
{
    firewall* f = (firewall*) fw;
    if (!is_accessible(f, url))
    {
        printf("[Proxy]: Access to %s is denied!\n", url);
        return FALSE;
    }
    return access_point(url, NULL);
}
```

Proxy in C

- Using: struct, callback function, void*

```c
void* access = access_point;
void* fw = NULL;

browse("http://vnexpress.net", access, fw);
browse("bbc.com.vn", access, fw);
browse("google", access, fw);

config_proxy(&access, &fw);

browse("vnexpress.net", access, fw);
browse("bbc.com.vn", access, fw);
browse("cornhub.com", access, fw);
```

Same interface, different results

Design Patterns in C

- Possible with limitations
  - Class => Struct
  - Overriding => Call-back function
  - Dynamic typing => Void pointer
- Still the main concept is kept
- Limitations:
  - Lack of encapsulation, inheritance
  - Pattern with complicated structure will be difficult

SURVEY KHOÁ HỌC

TEST (45’)

https://tinyurl.com/viettel-dsa