KullathonSitthisarnwattanachai-FinalExam.pdf

by Mos Kullathon

Submission date: 21-May-2022 11:28PM (UTC-0700)
Submission ID: 1841539108
File name: KullathonSitthisarnwattanachai-FinalExam.pdf (745.96K)
Word count: 2873
Character count: 13820

1. Section, Date and Time:
   TIC-TAC, Due **05-23-2022 at 11:55 PM**

Full Name in CAPITAL LETTERS | **SFSU ID**

KULLATHON SITTHISARNWATTANACHAI	921425216

2. Fina Exam (1 exam, 0 dropped): 100 points
3. To prepare for this exam, please review all the related materials including the packages, slides, mock-up exam(s), reading assignments, in-class practices, sample programs posted in the File Manager, and assignments.
4. You do not need to print this exam. No papers. No handwriting. No scanned images. No screenshots. Please type up all your answers in the answer space available in the exam. The provided exam will be in Microsoft Word format. Please submit a single PDF via iLearn.
5. All the rules of an actual exam apply to this exam such as: closed books, closed notes, closed IDEs, and no communication with anyone except the course instructor. The course instructor will be available on Zoom (zoom.ducta.net) or via email during the exam time. You cannot use any other materials or tools but only the provided exam which will be in Microsoft Word format.
6. Please ask all your questions, if any, during the review sessions. Thank you.

HONOR CODE:

- Please follow the CS Department's policies: https://cs.sfsu.edu/student-policies

- Please follow the course's policies: http://csc340.ducta.net/00-README-StudentConduct_AcademicHonesty.pdf

PART A – 50 Points

A.1 - 5 pts – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

A Bandai Namco interviewer asks you "Why Smart Pointers?" Please start your explanation with a code segment which properly demonstrates your points.

```
int a = 5;
int *pA = &a;
```

Smart pointers are pointers that automatically manage the resource it is pointing to. Using a raw pointer like above requires the programmer to manually free the memory or risk causing a memory leak. Smart pointers automatically deallocate memory when an object is no longer in accessible.

```
{
    auto smart_string{std::make_unique<std::string>("test")};
    std::cout << *smart_string;
}
```

Once the program leaves the block above, the `smart_string` will no longer be accessible and will therefore be destroyed automatically.

`shared_ptr<CSC340> sPtr { make_unique<CSC340>() };`
Should we expect an error? Why?

Assuming "CSC340" is a valid class, no. The unique_ptr created by make_unique is temporary within the parameter scope of the creation of the shared_ptr. This means that it simply "converts" the given (temporary) unique_ptr into a shared_ptr. Calling std::move is not necessary since the unique_ptr has yet to bind as an lvalue.

A.2 - 5 pts – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

A **`weak pointer`** is monitoring an object. Please explain in detail the steps you will take to finally have the monitored object owned by a **`unique pointer`**.

If a weak pointer is monitoring an object, it must be owned by a shared pointer. One cannot convert a shared pointer into a unique pointer because unique pointers are based on an exclusive ownership. However, you can check the `use_count()` of weak pointer to see when ithas a single ownership. To get access, use `lock()`, and finally, move the ownership to a unique pointer instance by using `std::move()`.

How is **memory leak** different from **dangling pointer?** Please recommend the best practice to avoid both **memory leak** and **dangling pointer**.

A.3 - 5 pts – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

It is legal to have a **pure virtual destructor** in a **C++** class as far as that destructor has a function body. Please explain the logic behind this requirement.

The "pure" part: destructors do not return anything and are therefore considered pure.
The "virtual" part: destructors can be declared as virtual so that its children may implement their own deallocation procedure. If the parent's destructor does not have a body, then upon destruction, there would be nothing to invoke after the child's destructor.

Pure virtual functions in C++ can have function body implementation. Please explain a scenario when and how this implementation is used. Then please provide a sample code to access/invoke that original function code if it was overridden by a child class.

A.4 - 10 pts – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

```
...
int funcB(int);

int funcA(int n) {

    if (n <= 0)
        return 1;
    else
        return n + funcB(n - 2);
}

int funcB(int n) {

    if (n <= 3)
        return 1;
    else
        if (n > 5) {
            return n * funcA(n - 9);
        }
        else {
            return n + funcA(n - 1);
        }
}

int main() {

    for (int i = 1; i < 4; i++) {
        cout << funcA(i) << endl;
    }

    return 0;
}
```

What is the output of this program?

2

3

4

Please explain your work and your answer.

- The loop in the main invokes funcA three times.
    - First iteration (bottom to top)
        - return 1
        - 1 + funcB(1-2) // 1+1
        - funcA(1) // 2
    - Second iteration
        - return 1
        - 2 + funcB(2-2) // 2+1
        - funcA(2) // 3
    - Third iteration
        - return 1
        - 3 + funcB(3-2) // 3+1
        - funcA(3) // 4

A.5 - 5 pts – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

Each lambda can have **6** components. Please name the **4** optional components.

[ ] P1:	parameter list
[ ] P2:	return type annotation
[ ] P3:	“mutable”
[ ] P4:	“throw”

Please write a lambda which has **5** components and returns a **`function pointer`**. Please explain your work and your answer.

[]() mutable -> void(*)() { return []{}; };

Has five components, in order of appearance: capture group, param list, “mutable,” return type, and function body; where optional, the components are redundant but constitute a legal expression. Return type indicates the lambda returns a function pointer that returns void; the body creates a void-returning lambda inline and returns it.

Compare in detail: **lambda closure** vs. **lambda class**

Do you think it is possible to write a recursive lambda expression? Please explain in detail.

Probably. One could provide the capturing group with a reference to the invoking scope, then access it within the body of the lambda.

function<void()> re = [&]() { re() }; // lambda body should have access to `re` because `this` is supplied by &

would probably work(?), but why…

However, it is probably not possible to make a recursive IIFE since the lambda itself has to be defined somehow so that it can be referenced inside of its body to call itself.

A.6 - 10 pts – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

How is **Merge Sort** different from **Quick Sort**?

Both "divide and conquer" the given list recursively in order to sort them. Merge sort divides list into half until each sublist becomes a singleton. Then, merge and sort sublists until complete. Quick sort divides a list around a pivot and sort them by swapping elements that are greater than the pivot to the right, and those lesser to the left. Efficiency of quick sort depends on location of the pivot, whereas merge sort's performance is consistent.

71 97 79 15 63 96 36 51 62 18 17

Show the contents of the array above **each time** a **Merge Sort** changes it while sorting the array into **ascending order**. Please explain your work and your answer.

71	97	79	15	63	96	36	51	62	18	17	Start
71	97	79	15	63	96	36	51	62	18	17	First split
71	97	79	15	63	96	36	51	62	18	17	Second split
71	97	79	15	63	96	36	51	62	18	17	Third split
71	97	79	15	63	96	36	51	62	18	17	Fourth split
71	97	79	15	63	96	36	51	62	18	17	All sublist split into singletons
71	97	79	15	63	96	36	51	62	**17**	**18**	First merge
71	97	**15**	**63**	**79**	**36**	**51**	**96**	**17**	**18**	**62**	Second merge
15	**63**	**71**	**79**	**97**	**17**	**18**	**36**	**51**	**62**	**96**	Third merge
15	**17**	**18**	**36**	**51**	**62**	**63**	**71**	**79**	**96**	**97**	Fourth merge, sorted

A.7 - 10 Points – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

- Memory Area **1**: Environment
- Memory Area **2**: Runtime Stack
- Memory Area **3**: Free-store
- Memory Area **4A**: Uninitialized Data
- Memory Area **4B**: Initialized Data
- Memory Area **5**: Binary Program

```
#include <iostream>
#include <functional>
using namespace std;

static string model;

class Car {
public:
    Car() {};
    static string carModel;
    string name{ "McQueen" };
};

string Car::carModel = "S";

Car* testCar(string str) {
    unique_ptr<Car> uCarPtr { make_unique<Car>() };
    static Car car;
    Car* driverlessCar = new Car();
    return &car;
}

function<int (void)> testLambda() {

    int price = 100000;
    int rank = 1;

    function<int(void)> carLambda = [rank, &price]()->int {
        cout << "carLambda in testLambda" << endl;
        return price + rank;
    };

    return carLambda;
}

int main(int argc, char* argv[], char* envp[]) {
    Car* carPtr = testCar("CS");
    string carName{ carPtr->name };
    model = Car::carModel;

    auto testLambdaPtr = testLambda();
    cout << testLambdaPtr() << endl;

    return 0;
}
```

a. In **which memory area** is this element stored? Please **state** your choice and explain **why**?
b. The **lifetime**, beginning & end, of this element? **Why**?

uCarPtr object **[1] [2]** [3] **[4a] [4b] [5]**

Why [area]?

The object is in the free-store because it is created using `make_unique()`. It is managed by the unique pointer that owns it.

What lifetime and why?

Until `uCarPtr` goes out of scope. Since `uCarPtr` is a unique_ptr, it is the sole owner of the object and manages the lifetime of the object that is created on the free-store. When it is no longer accessible, it will deallocate the memory for the object.

carLambda expression **[1]** [2] **[3] [4a] [4b] [5]**

Why [area]?

The expression is evaluated at runtime and is contained inside a function.

What lifetime and why?

Until the function completes its execution. The expression is contained inside a function.

testLambdaPtr closure **[1] [2]** [3] **[4a] [4b] [5]**

Why [area]?

The function object for the lambda is stored on the heap once evaluated.

1
What lifetime and why?

The function object lasts until the end of the program.

envp [1] [2] **[3] [4a] [4b] [5]**

Why [area]?

1
\`envp\` lies in the parameter scope of the \`main()\` function.

What lifetime and why?

1
Until the function completes its execution. In this case, since it's the \`main()\`, it lasts till the program terminates.

PART B – 50 Points

B.1 - 10 Points – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

```
#include <iostream>
using namespace std;

const int y = 1;

int main() {

   int static y = 2;

   int i = 3, j = 4, m = 5, n = 6;

   int a = [](int x, int i = 1) { return x * i; } (y, 3);

   int b = [=](int x) { return [=](int b) { return b + j; }(x) % 7; }(a);

   int c = [=](int x) mutable ->int {

      m = 6;

      return [&](int j) mutable {
         y = a * b;
         return y / j;

      }(x)-m;

   }(b);

   cout << a << endl;
   cout << b << endl;
   cout << c << endl;
   cout << m << endl;
   cout << y << endl;

   return 0;
}
```

This program **outputs 5 lines**. What are they? Please explain your work and your answer.

1. 6

2. 3

3. 0

4. 5

5. 18

B.2 - 10 Points – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

```
Installer* make(const Installer& i, const double& s) {
   unique_ptr<Installer> u{ make_unique<Installer>(i, s) };
   return u.release();
```

}

1

Please use 5 different approaches to create **function pointer `funcPtr`** which points to **`make`**. Please explain your work and your answer.

`auto`

auto funcPtr = make;
auto should automatically deduce the type of the function.

1

`Actual type`

Installer* (*funcPtr)(const Installer&, const double&) {make};

`Function object, <functional>`

std::function<Installer*(const Installer&, const double&)> funcPtr{make};

`typedef`

typedef Installer*(*ref)(const Installer&, const double&);
ref funcPtr{make};

`using`

B.3 - 10 Points – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

```
int wakeDaemon(const int& pId, Daemon* pAddress) {
    return 60;
}

string wakeDaemon(const int& pId, int tVal) {
    return "pId: " + to_string(60 * tVal);
}
```

Please create 2 function pointers:

- fPtr1 points to the first **wakeDaemon**
- fPtr2 points to the second **wakeDaemon**

Please explain your work and your answer.

Cannot use auto with overloaded function, must spell out full type.

1

`First`

int (*fPtr1)(const int&, Daemon*) {wakeDaemon};

`Second`

std::string(*fPtr1)(const int&, int) {wakeDaemon};

B.4 - 10 Points – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

```
unique_ptr<PC> name_uPtr { make_unique<PC>(" accountId") };
```

Please write 1 function header which can help properly pass the above unique pointer in main to a function and explain how it works in detail.

void DoSomething(std::unique_ptr<PC> pc);
Caller should provide the function with the ownership of the object by using std::move(). For example, to pass name_uPtr to the function, the caller should do: DoSomething(std::move(name_uPtr)). Since name_uPtr is a unique_ptr, one cannot pass-by-value, and thus ownership must be passed on to the parameter scope of the function.

Please describe a scenario when the approach you proposed above may not be the best solution then provide a better approach. And code the function header for the better approach.

void DoSomething(std::shared_ptr<PC> pc);
This approach instead uses shared_ptr. This way, the caller can provide the smart pointer by creating a shared ownership instance. This may be suitable if the function actually requires some other return type and cannot return the control back to the caller, or if ownership is required by other members.

B.5 - 10 Points – *Your answer must be in your own words, be in complete sentences, and provide very specific details to earn credit.*

```
...
Student* func() {

    unique_ptr<Student> arr[]
    {
        make_unique<Student>("CSC340")
    };

    // #1 Insert Code
    return arr->release();
}

int main() {
    // #2 Insert Code
    std::weak_ptr<Student> weak;
    weak = std::make_shard<Student>(std::move(f()[0])); // move surviving object and make weak_ptr
    std::cout << weak.use_count() << std::endl;         // zero; died in make_shared param scope
    weak.reset();                                       // "properly destroy it"
    std::cout << weak.use_count() << std::endl;         // "check again," still zero
    // control block should die at the end of main()
}
```

[#1 Insert Code]: **Write code** to keep all the object(s) which element(s) of array **arr** owns alive outside of the scope of **func**.

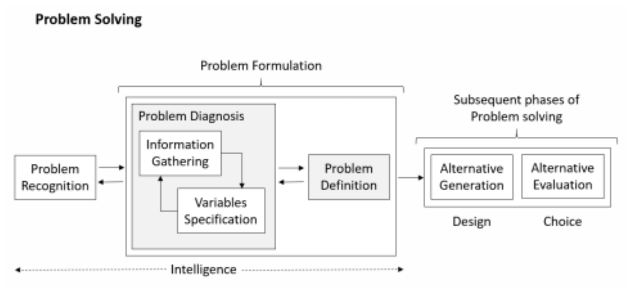
See above.

[#2 Insert Code]: **Write code** to have a weak_ptr monitor the object which survived; Then test if it has any owner; Then properly destroy it; Then test again if has any owner; Then destroy the Control Block.

See above.

PART C – 10 Extra Credit Points

C.1 - 5 pts



What did the IDEO team do during the Problem Formulation phase? What was the most important question? Can we use the same approach in Software Development?

They gathered information about the traditional shopping cart in order to identify the issues about them and what they could do to make them better. The group does not have formal leadership and pools ideas from different group. The same approach can be applied in software development; we first need to identify the problem that we are solving before implementing the solution. There are many ways to approach a problem, so the one you come up with may not necessarily be the one. Finally, we can combine everyone's ideas to come up with the best possible solution. With the IDEO team, they brought in different aspects of the shopping cart and combined them into one design.

1

C.2 - 5 pts

Your programming team lead/leader asks you to use raw pointer to implement a memory bound function. You know that using Smart Pointers is the way to go. And you heard that the leader does not know Smart Pointers. What would you do and say? Please provide your answer in detail.

Approach the lead with respect. First, you present your case about using smart pointers. Your assumption may not be correct that (a) the lead does not know about smart pointers or (b) that smart pointers are necessarily appropriate for your situation. If they are not aware of smart pointers, bring them up to speed and explain why you think it may be suitable for your case. Alternatively, pitch it to the team to see what the rest think.

KullathonSitthisarnwattanachai-FinalExam.pdf

ORIGINALITY REPORT

61%	28%	0%	61%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to CSU, San Francisco State University Student Paper	61%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		

KullathonSitthisarnwattanachai-FinalExam.pdf

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

Your final letter grade for this course is: **A**

Congratulations! It was my pleasure working with you this semester. Good luck on your other final exams and in all your future endeavors. Thank you. Duc

Important Dates:
- Friday, June 3, 2022: Spring '22 Grades Available on SF State Gateway

- Friday, June 17, 2022: Spring '22 Official Transcripts Available with Semester Grades

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10