Initial class.

Tuesday, May 8, 2018 11:33 AM

Assignment 1. } → Due soon.

learn material on

Assignment 0 due by Friday May 18 at 4pm
" 1 " " " May 25 at 4pm

* lecture slide supplement

Wednesday, May 9, 2018 11:28 AM

office hours

Tuesday 1:30 - 2:30

Software Performance

Wednesday, May 9, 2018 11:28 AM

slide 1432

PAM model - all operations take place

loop and subroutines → not elementary operations.

asymptotic complexity: what happens when the problem size goes to ∞

big oh → upper bound

big omega → lower bound.

big theta → range

Abstract Data types.

Wednesday, May 9, 2018 12:11 PM

pg

arrays are contiguous.

Wednesday, May 9, 2018 11:29 AM

/home/frodo/public/ugls_lab/bin/sde_shell

For submission, do IDENTIFICATION.txt

Assignment_precheck

Ln -s tmp_cmake/compile_commands.json
YouCompleteme

Set(CMAKe_cxx_flags) -fsanitize=address -fsanitize=undefined

Wednesday, May 16, 2018 1:34 PM

Cmake -H. Btmp_cmake -DCMAKE_BUILD_TYPE=Release

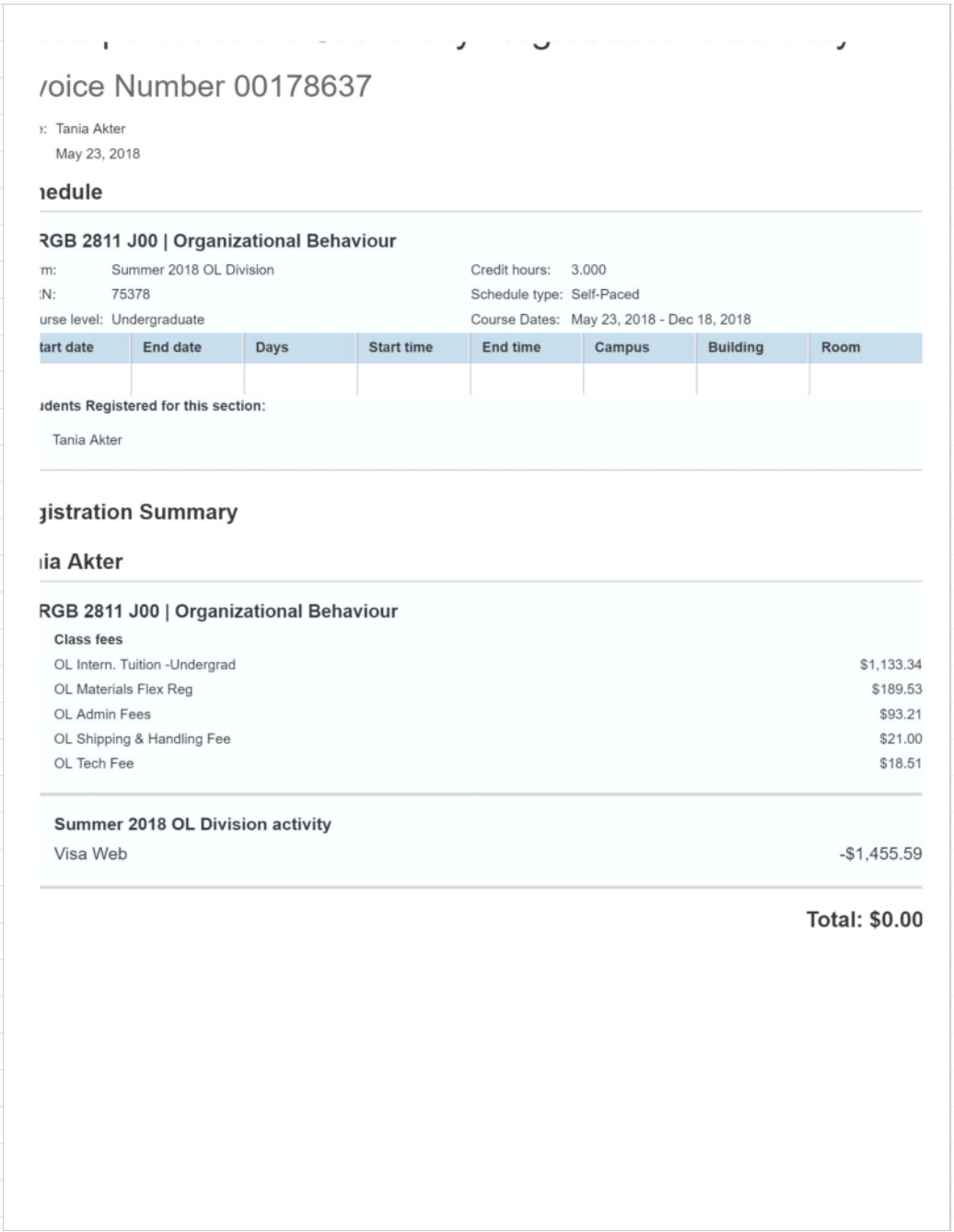
Cmake --build WHEREVERYOUHAVEPUTTHEBUILD

In source build -> don't do

Boost? Link libraries
Boost chrono

Printout

Friday, May 11, 2018 11:51 AM



/oice Number 00178637

: Tania Akter

May 23, 2018

ıedule

RGB 2811 J00 | Organizational Behaviour

m: Summer 2018 OL Division
N: 75378
urse level: Undergraduate

Credit hours: 3.000
Schedule type: Self-Paced
Course Dates: May 23, 2018 - Dec 18, 2018

tart date	End date	Days	Start time	End time	Campus	Building	Room

ıdents Registered for this section:

Tania Akter

gistration Summary

ıia Akter

RGB 2811 J00 | Organizational Behaviour

Class fees

OL Intern. Tuition -Undergrad	$1,133.34
OL Materials Flex Reg	$189.53
OL Admin Fees	$93.21
OL Shipping & Handling Fee	$21.00
OL Tech Fee	$18.51

Summer 2018 OL Division activity

Visa Web	-$1,455.59

Total: $0.00

Page 164. → 164

Function type and const qualifier

const expr → for classes?

Pointer type, iterator type, & reference type

Foo.cpp example

```
6  ns for  & x
10 cont ::vector
11
13      ::const_ iterator p
20  const std:: vector<int>
22  const std:: complex(  )
```

→ why defining in header file is fundamentally wrong?

→ Inline Functions. } → why is it ok?

Ifndef)

A when using inline function → always give the definition.

✳

Read

Remarks on Header files & Function declarations

Remarks on Header files and function Declarations

Tuesday, May 22, 2018 11:43 AM

- Lecture slide supplement 4

Go through the Counter example

Inline - > definitions in header file
Template -> definition in header file (or there will be linker errors)

For default parameters, put it in the declarations

Non-static member function -> has an implicit "this"

Adding constexpr to a function

Wednesday, May 23, 2018 11:41 AM

Constexpr constructors

Contexpr must always have literal type

Moving VS copying example

Wednesday, May 30, 2018 11:32 AM

-Rvalue expression returns by value, never by reference

??About const with move

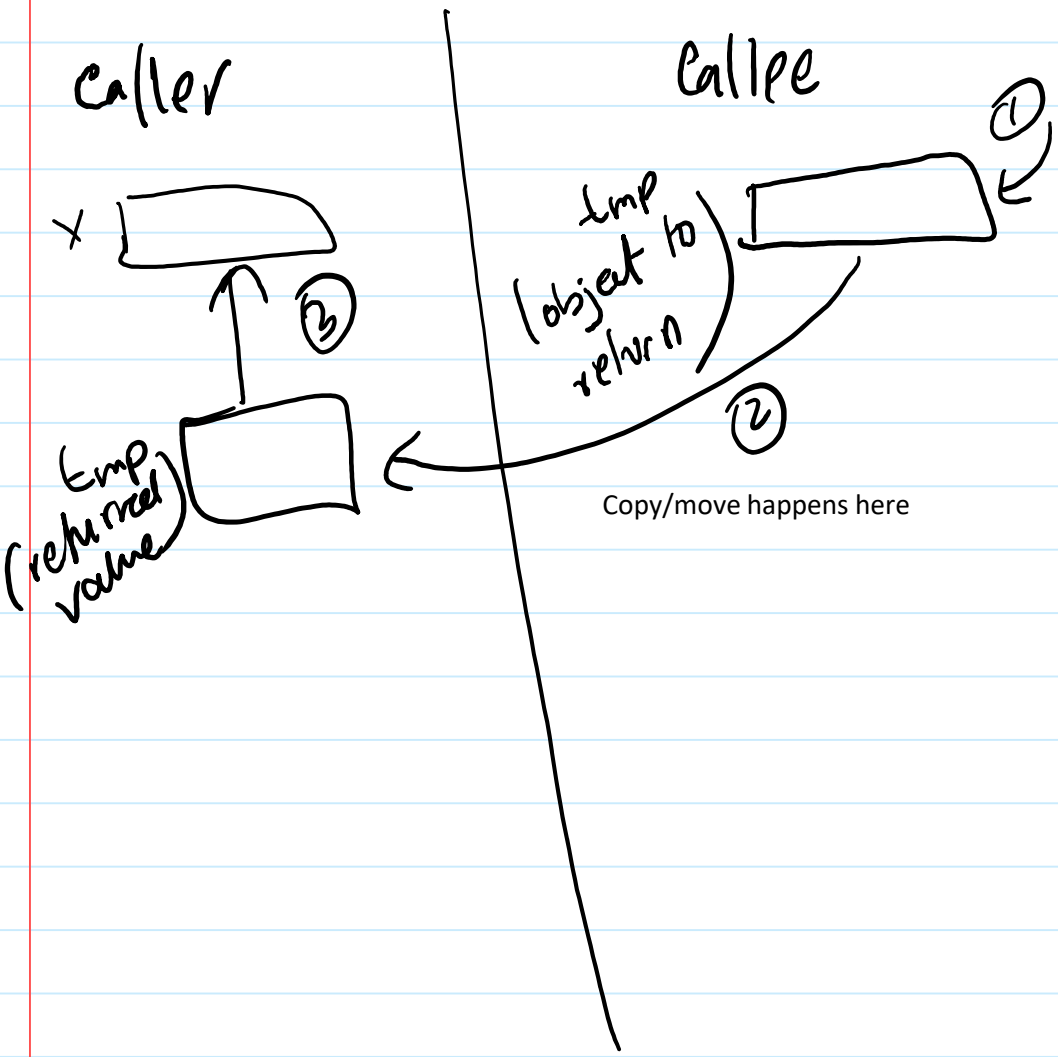
Std:move returns an rvalue reference to an object

If the return type of an expression is const, when we do s = get_const_value(), copying is done.

Copy Elision

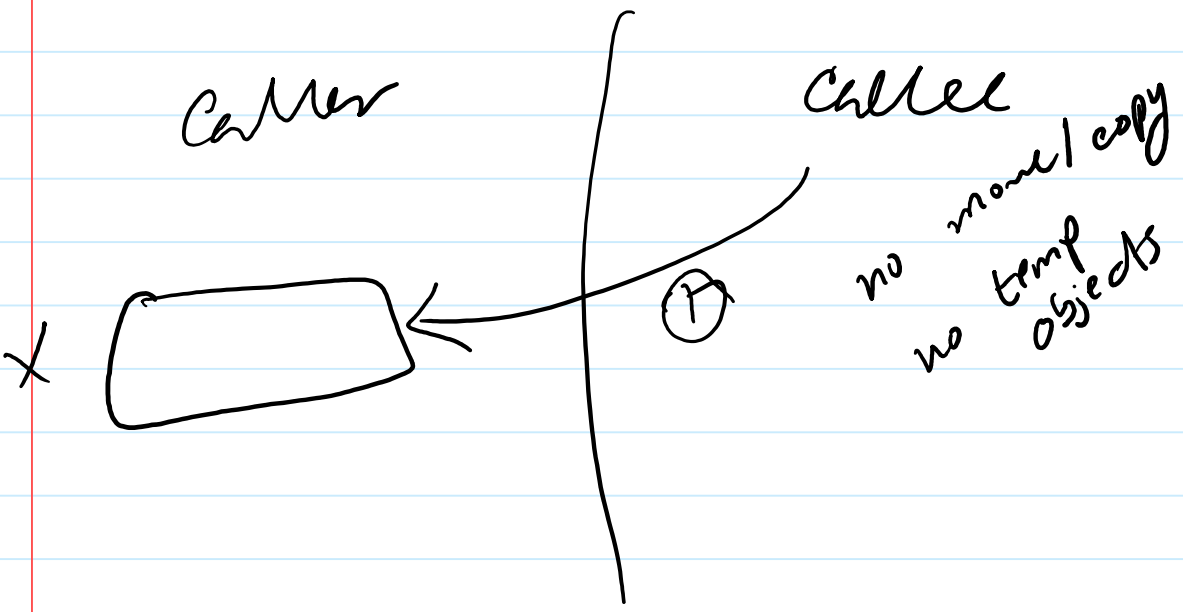
When we are avoiding move/copy operation

Return by-balue example 1: summary

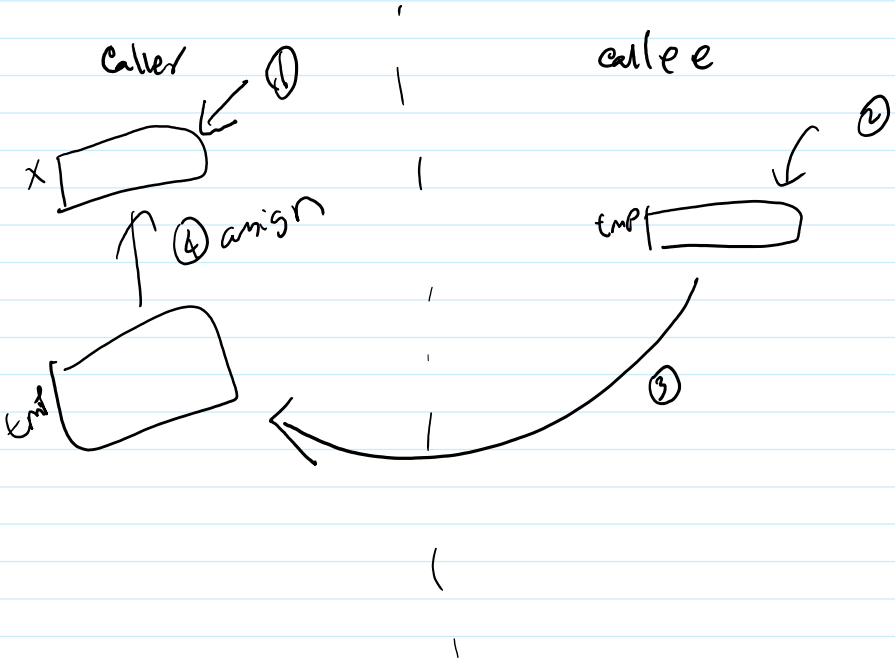


Copy/move happens here

Copy elision:



Return-by-value example 2:



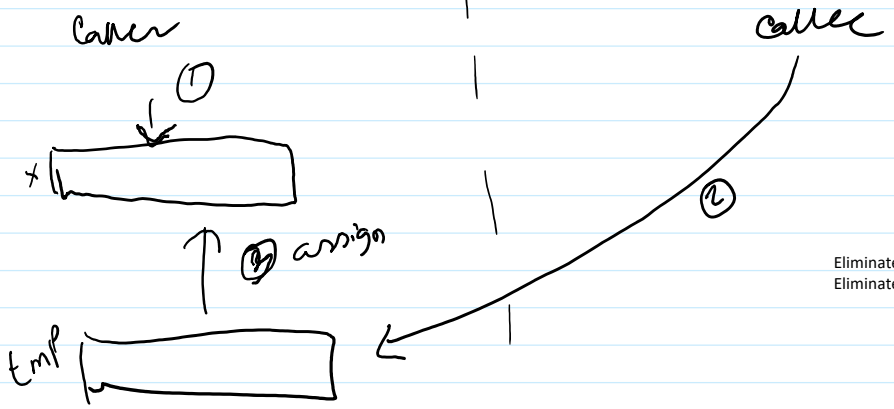

```
T calle()
{
        Return t()
}

Void caller() {

T x;
X  = callee();

}
```

Copy elision';



Eliminates 1 copy/move
Eliminates 1 temporary

- Since X is already constructed, the compiler will not construct on top of it, hence we need a temporary object

Copy Elision and Returning by Value

Wednesday, May 30, 2018 11:59 AM

Pg 296?

When does it have to do it, and when is it a nice to have?

Factory functions: Functions that makes objects

```
Widgest func1() {return widget()}
Widget func2() {widget 2; return w;}

Main{
        Widget x(func1())
        Widget x(func2())
}
```

In case of when a function call is returning a named object, copy elision is not guaranteed.

Unnamed object returns guarantees copy elision

Case where copy eliison is not possible:

```
Widget w;
Widget x;
If()
Return w
Else
Return x
```

Because the compiler does not know which one will be returned, cannot to lazy construction till when it is needed. Hence it constructs it first and returns later. Copy elision is not possible

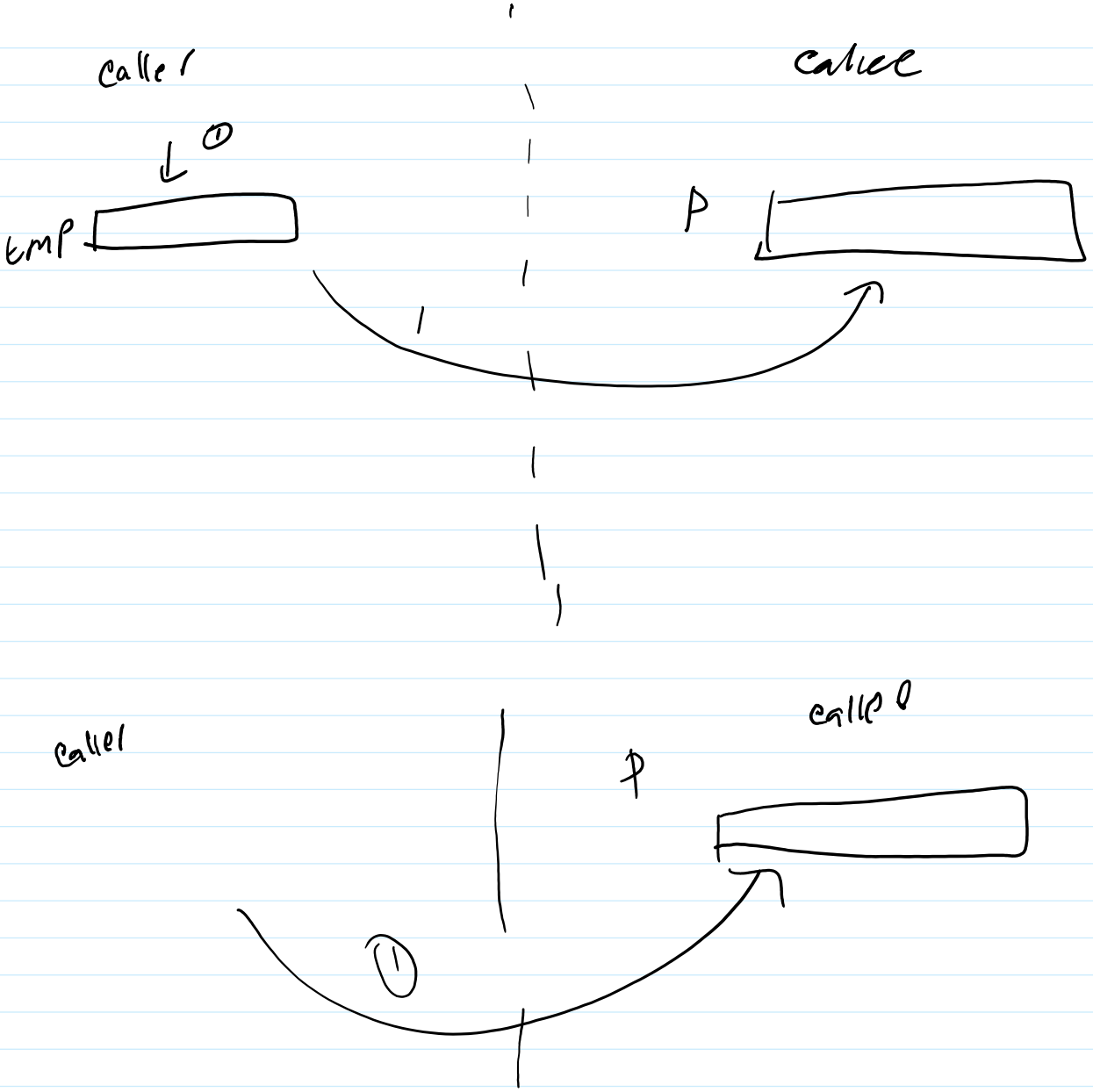
Case where copy eliiosn is possible:

```
....
If()
Return Widget(42);
Else
Return Widget(0);
```

The compiler does not need to construct till when it is needed.

Pass by-value exampe: summary

Wednesday, May 30, 2018 12:12 PM




```
Void callee(T p)
{
}

Void caller(){
      Callee(T)
}
```


Removes 1 tmp object
Removes 1 copy/move

Implicatio nof Rvalue-Reference Type function parameters

Friday, June 1, 2018 11:35 AM

T::T(const T&) <- lvalue

- You can olny bind rvalue referenceds to rvalue expressions
- T&& x= [] <- rvalue
- So you can only bind rvalue reference to temporary objects
- So if parameter passes by rvalue reference, safe to change value

T::T(T&&) <- rvalue

Exception

Friday, June 1, 2018 11:43 AM

Catching exception - 803

For catching, we do catch by reference, because we wanna do move, not copy since copy operation itself may throw an exception

NEVER throw an exception in destructors

Stack unwinding:

- Things get destroyed in the opposite order of creation
- See Stack Unwinding Example = 811

Function Try Blocks

Tuesday, June 5, 2018 11:32 AM

//remember to include <stdexcept>

Pg 817 --> start

Lecture slide supp - Example return statement and moving/copying

When you return by value, copy elision is allowed .

If you are returning something that's a local variable to a function, you treat it as an rvalue. If there is a move constructor you can call on the object, you can move. (Ask this question when copy elision is allowed / (not strictly requried) since you don't know if it is going to copy or move)

//

- When you make a temporary object and it's not the same type as the return type, it will either through compiler error, or will generate the return type for us, and copy elision will be required

Function try Blocks

What happens when you construct an object?

- Construction of base class from left to right
- Data members in the order of top to bottom
- Constructor body

Destructor:

- Destructor body
- Data member from bottom to top
- Base class objects from right to left

Constructor might through in:

- Constructor base class object
- ?
- ?

Construction/ Destruction example

Exception specification

Noexcept specifier

Example:

Void func1() // may throw anything
Void func2() noexcept{false} // promise that function will throw
Void func3() noexcept{true} // promise that the function will never throw
Void func4() noexceot // does not throw

//unless overriden, never make your destructor throw.

Noexcept is a function's type, not a function signature. So you cannot overload on noexcept specifier.

Example in pg 824 (swap function)

// if the move construction on T does not throw ever, AND move assignment never throws, it does not throw.

Void exchange(T&, a T& b) noexcept(std:: is_nothrow_move_constructible_v <T> && ….) ….

What happens when we throw exception on noexcept function?

Noexcept means the function does not emmit function. If the exceptions are being thrown inside and being taken care of, it is fine.

Eg:

```
Void die() noexcept {
        Throw 0;
}
```

When we use noexcept(true), we are promising that in current, or ANY FUTURE version of the code, the function will not throw.

Exception and Function Calls (pg 826)

T Func(T) noexcept(expr);
T x;
T y = func(x); // function call

When can exception happen?

- Parameter passing since parameter passing by value will cause either move or copy operations (which are functions)
- Returning by value, you have to propagate the value out of the function, which requires move or copy constructors.

If exception due to paarameter must be avoided:

- Pass by reference, since it's just passing the nickname of a variable/ object.
- Ensure noexcept move and/or copy constructors for the value being returned by value
- Ensure function invoked is implemented in a way that guarantees copy elision.

Addition and Subtraction

Wednesday, June 6, 2018 11:55 AM

Pg - 829?

Start pg 1651

Setting and querying rounding mode

He will check the rounding mode if it is set back to normal

#include <cfenv>

Assignmnet 2

Friday, June 8, 2018 6:13 AM

Tania Akter
V00810640

8.1 :

The answers are given in highlighted texts

```
1 #include <iostream>
2 #include <vector>
3 #include <utility>
4
5 std::vector<int>&& func1(std::vector<int>& x) {
6 return static_cast<std::vector<int>&&>(x);

7 // x? lvalue

8 // static_cast<std::vector<int>&&>(x)? rvalue

9 }
10
11 int main() {
12 std::vector<int> x = {1, 2, 3};
13 std::vector<int> y;
14 int a;
15
16 for (auto i = x.begin(); i != x.end(); ++i) {

17 // x.begin()? rvalue

18 // ++i? lvalue, since prefix returns lvalue reference

19 *i += 5;

20 // i? lvalue

21 // *i? lvalue, because I is a pointer type

22 // *i += 5? lvalue

23 }
24
25 a = x[0];

26 // x[0]? lvalue


27 ++a; a++;

28 // ++a? lvalue

29 // a++? Rvalue
30
```

```
31 y = func1(x);
32 // func1(x)? rvalue, since the return type is a rvalue reference type
33 // y = func1(x)? lvalue
34 }
```

Friday, June 8, 2018 2:37 PM

8.24

The answers are written in highlighted texts

47 int main() {
48 Widget a;

49 Widget b(a); // ??? Copy constructor required

50 Widget c = a; // ??? Copy assignment required

51 Widget d(std::move(c)); // ??? Move constructor required

52 Widget e = std::move(d); // ??? Move assignment required

53 Widget f(make_widget_1()); // ??? Move/copy constructor may be elided, since after the conditional branch, only one object is required.

54 Widget g(make_widget_2(true)); // ??? Move/copy constructor required since copy elision is not possible

55 c = a; // ??? Copy assignment required

56 b = std::move(c); // ??? Move assignment required

57 a = make_widget_1(); // ??? // Move/copy constructor may be elided, but copy assignment for a is required

58 a = make_widget_2(true); // ??? Move/copy constructor required, and copy assignment required

59 func_1(a); // ??? Move/copy elision is guaranteed since it is passed by value

60 func_1(std::move(a)); // ??? Move/copy elision is not guaranteed.

61 func_1(make_widget_1()); // ??? Move/copy constructor may be elided because one widget is constructed after the conditional branch in the code.

62 func_2(std::move(b)); // ??? Move/copy elision cannot happen

63
}

Friday, June 8, 2018 2:37 PM

8.25

The answers are written in highlighted field:

66 // ... (code omitted here changes x, y, and z)

67 z = x + y; /* ??? */ Temporary objects required.

One temporary object is required is formed in line 53, to copy the value of x such that the result of addition of x stored in a temporary object and returned, making sure the value stored in x itself is not changed. The z then gets the value of the addition.

68 z = z + z; /* ??? */ Temporary objects are required, in line 53, with counter(x). A temp object is made with the first z operand, It was required to store the value of first z operand, and the value from the second operand was added to the temporary object, and returned.

Two temporary objects are created during argument passing in line 32.

69 y = ++z; /* ??? */ No temporary

70 z = y++; /* ??? */ Temporary object formed in line 26, to hold the old value of y. It was required to make sure z gets the old value of y, and then y is incremented.

71 x = z; /* ??? */ One Temporary object created when calling operator= in line 13

72 std::cout << x << ' ' << y << ' ' << z << '\n';
73 }

Friday, June 8, 2018 2:37 PM

8.26

Array based implementation requires amortized O(1) for pushing and popping to the stack. However, it has an worse case time complexity of O(n) for push when the array is full, to move the existing elements into a new array, which invalidate any references to the old array. Arrays have contiguous memory blocks, which is better for caching.

Node based implementation has O(1) for both push and pop. However, node based implementation does not have contiguous memory storage, which is bad for caching. Moreover, the nodes take more memory then array based implementation because nodes have an extra pointer pointing to the next node, which is an overhead. Node based implementation makes sure it does not go out of capacity, which gets rid of the necessity to copy the entire stack over to a new one.

Friday, June 15, 2018 11:30 AM

The alignof operator

Operator Array Delete(i.e operator delete[])

New -> allocate memory, and then construct object in the allocated memory

Tuesday, June 19, 2018 11:41 AM

Optional value example: optval.hpp

Assignment 3

Friday, June 22, 2018 1:12 PM

Tania Akter
V00810640

6.1

```
void func() {
    initialize(); // perform initialization
    do_work(); // do some work
    cleanup(); // perform any necessary cleanup
}
```

It is not exception safe. Because if *do_work()* throws an exception, *cleanup()* is never called and cleanup operation is not performed, which may cause resource leak.

6.2

a) die3, die, countdown, hello, i, bjarne, herb, dv, u, z
b) s

6.3

a. Func() can throw while allocating memory for buf1 and buf2, and it is not handled. The allocate memory parts should be in try-catch block.

a. It can throw during saving formatting flag or restoring formatting flags. It can also throw during the print statement in line 11. The entire function body should be kept in a try-catch block, or use RAII.

b. Put() can throw an exception since std::deque::push_back may throw still throw an exception. The function body should be kept in a try-catch block.

6.5

analyze() does not throw exception because it has nothrow guarantee with *noexcept*.

doWork() may throw exception because it does not have the *noexcept* filter. Moreover, while passing the argument *globalThing,* since it is not a built in type, copy/move constructor is called, which may throw, causing *doWork()* to throw.

Wednesday, June 27, 2018 11:36 AM

Virtual memory Exercise

- Virtual and physical memory parameters
- 16 bit va
- 12 bit pa
- Page size is 256 bytes

Determiee: numberof virtual pages: 2^16/ 2^8 = 256
Numberof physical pages:
2^12 / 2^ 8 = 16
Number of bits in page offset:
Log_2 256 = 8
Number of bits in virtual page number 16-8 = 8

Ppn - > physical page number
Access violation happens because you are not supposed to access that part of the physical address, may not be readable, writable, or executable.

You can't shuffle around what page offset is. Page offset is always constatn, from virtual to physical.

Wednesday, June 27, 2018 11:47 AM

Translation lookaside buffer (TLB)

Virtual and physical caches

- Translation from virtual to physical address happens only when there is a cache miss
- How addresses translate from virtual to physical depends on what process is using it. So we may have the same physical address for two diffreent virtual address

Cache aware algorithm

Code Transformations to improve cache efficiency

Loop interchange

Merging arrays
Loop interchange
Loop fusion
Blocking

Blocking example

Before merging the array, high possibility of cache misses.
Onsexpr int num_pt = 89384
x[]
y[]
z[]

After:
Num_pot = 898p
Struct pt
{
X
Y
Z
}
Pt p[num_pt]

A4

Wednesday, June 27, 2018 1:32 PM

Pg 1212
Forward iterator example

Use of std::move in reutrn expressions -> BAD

Read up on substitution failure

Std::enable_if_t<std::is_convertible_v<OtherT*, T>>

Assignment 4

Friday, July 6, 2018 3:11 PM

Tania Akter
V00810640

8.17

a.
The best option is Array-based sorted list
The element insertion can be done in sorted order in the beginning since the contents are known. Since the container is first created and not dot change frequently, it means we do not have to worry too often about growing the container to insert more values.
We can also search for particular value from already sorted list by using a binary search.

Since we also have to iterate over the container in sorted order, we iterate over easily since it is already sorted.

b.
Node-Based sorted list
Node based sorted list can be used to insert and remove at a constant time, hence modifying the container frequently is not costly. We can search for a particular value using binary search in log(n) time, since the list is already sorted. Moreover, iterating over the elements in sorted order is going to be a linear time since the list is already sorted.

c.
Node-based heap.

We need heap because we are always removing the largest element from the container. Since the content of the container is frequently updated, we can take advantage of having node-based heap and do the insert and remove operations in logarithmic time.

8.18
If we want to initialize a big container size, such as an array, constructing it will construct each of the elements in it, which is costly. We want to only allocate memory for the object in this case without constructing every element in it, hence we want the memory allocation and construction to be separate. Similarly, we may not want to both destroy objects and deallocate memory at the same time. One might want to free the memories to be used later, but if we destroy the object that lives inside, we might be destroying objects that the user has passed.

8.19

Advantages:

1. Elements are stored contiguously in memory
   It provides advantages on caching the elements.

2. No per-element storage overhead
   When storing elements with node, we need an iterator which is an overhead for each element

insertion. With array-based containers, we can get rid of the extra overhead

8.23

a.
Process has a substantial amount of state, and are neither movable, hence they cannot be stored in nonintrusive containers. We need intrusive container to store the processes.

b
We need intrusive container, because the nonintrusive container provide stable references, and for the mutexes to be used, we need to have stable references.

C
We need intrusive container because the mutexes only need to exist inside the container.