12/12/24

1. Because non-deterministic random number generators are generally slow and exhaustible, what is recommended If you want large amounts of random numbers that are different for each execution?

//\* NDRNG(from random lecture) is generally slow. How do you optimize? Ans: pass it as seed to mt19931(something like this, check your notes)

* 1. **Using a NDRNG to seed a pseudo random number generator**
  2. Measure the amount of entropy left in std::random\_device and add accordingly.
  3. Restrict the use of random numbers by other processes
  4. Use a uniform distribution instead of a normal distribution
  5. Save the random numbers generated for each execution to reuse
  6. Ensure that the NDRNG is being refreshed with new resources of randomness
  7. Use multiple NDRNGs simultaneously

1. When you iterate over a map, with a range-based for loop, what type do you get for each entry?
   1. A value
   2. A key
   3. A submap
   4. **A pair of key and value**
   5. An input stream
   6. A size 2 vector
   7. An iterator to the entry
2. Calling the front member function on an empty vector does what?
   1. Returns an empty string
   2. Returns a null reference
   3. Returns a size 0 dynamically allocated array
   4. Returns a default object of the same type as the element of the vector
   5. Returns a 0
   6. Returns a null pointer
   7. **Undefined behavior**
3. Recursive functions all share what characteristics?
   1. They take two arguments
   2. They are generic
   3. They work on vectors
   4. They replace the need for input or output
   5. **They can invoke themselves**
   6. They are required for dynamic memory management
   7. They return Boolean values
4. What is the primary advantage that std::array has over C-style or traditional arrays?
   1. They can be dynamically allocated
   2. They can be used with algorithms (<algorithm>)
   3. **They store their size**
   4. They are optimized
   5. They are faster
   6. They take less memory
   7. They can be dereferenced
5. The class std::mt19937\_64 that was used in-lab and in the homework is an example of which of the following?
   1. An iterator
   2. An algorithm that takes a half-open range
   3. A complex number
   4. A fixed-width integer
   5. A type of distribution
   6. A container
   7. **A pseudo-random number engine**
6. In order to provide a default ability for a user-defined class to be sorted, what operator must be overloaded? (E.g., if vector<Student> was to be used with the sort algorithm, what operator should be implemented for Student?)
   1. **operator<**
   2. operator<<
   3. destructor (~)
   4. Rule of three
   5. operator=
   6. operator>>
   7. copy constructor
7. If a vector named vec has a size of 1, which of the following expressions must be true?
   1. vec.add(0) == 1
   2. vec.clear() == 1
   3. vec.reserve() == 1
   4. vec.capacity() == 1
   5. **++vec.begin() == vec.end()**
   6. vec.empty()
   7. vec.rbegin() == vec.end() + 1
8. Iterators into a vector become invalid once what of the following occurs?
   1. **The capacity of the vector increases**
   2. The iterator is incremented
   3. The iterator is equal to the begin iterator
   4. The iterator is copied
   5. The vector is iterated over
   6. The vector is dereferenced
   7. The iterator is assigned to
9. What trait do all C-style strings share?
   1. **Terminated with a null character**
   2. They are created at compile time (also create runtime, like dynamic)
   3. They are string literals

//Not all C-style strings are string literals. They can also be arrays of characters or dynamically allocated strings.

* 1. They are non-empty

//A C-style string can be empty, represented as a single null character ('\0')

* 1. They are const

//C-style strings are not inherently const. String literals are immutable (effectively const), but other C-style strings (e.g., arrays of characters) can be mutable.

* 1. They can’t hold whitespace characters
  2. They are non-empty

1. size\_t is an example of which of the following categories:
   1. A vector
   2. **An unsigned integer type**
   3. A C-style string
   4. The value returned by a string’s size member function
   5. A floating point variable
   6. The largest possible value
   7. An address
2. A pointer to an element in an array is an example of what type of iterator? (specification needed)
   1. Bidirectional // list, set, map
   2. Lambda
   3. Reversible
   4. Dereferenced
   5. Forward // forward\_list
   6. **Random access // vector, deque, array**
   7. Secondary
3. How many elements will be copied in the following statement assuming array\_ptr points at the first element in an array of size 20?

copy(array\_ptr, array\_ptr + 4, array\_ptr + 5);

* 1. 0
  2. 1
  3. 2
  4. **4**
  5. 5
  6. 8
  7. 9

1. Which of the following standard library features allows you to use cin as a source for elements in an algorithm like transform?
   1. lambda\_expression
   2. sort
   3. overloading
   4. ranges::copy
   5. back\_inserter
   6. **istream\_iterator**
   7. ifstream

// code explanation

// std::copy(std::istream\_iterator<int>(std::cin), std::istream\_iterator<int>(), std::back\_inserter(vec));

1. back\_inserter is a function with what return type?
   1. A pointer
   2. size\_t
   3. predicate
   4. inserter
   5. int
   6. vector
   7. **iterator**
2. In function “test” in section 13.2 of the required textbook, there is the following expression \*p != ‘a’. Which of the following more parenthesized expressions is equivalent?

void test () {

string m {“Mary had a little lamb”};

for (auto p : find\_all(m, a))

if (\*p != ‘a’)

cerr << “a bug!\n”;

}

* 1. \*(p !)= ’a’
  2. \*p (!=’)a’
  3. \*p!(=’a’)
  4. **(\*p)!=’a’**
  5. \*(p!=’a’)
  6. \*(p!=)’a’
  7. \*p!(=)’a’

1. What advantages do smart pointers have over naked/raw pointers?

**// book answer: smart pointers can release the resource when they fall out of scope**

Advantages of Smart Pointers:

**Automatic Memory Management**: Smart pointers automatically allocate and deallocate memory, reducing the risk of memory leaks and dangling pointers.

**Exception Safety**: Smart pointers clean up resources automatically when *an exception is thrown*, ensuring safety.

**Shared and Unique Ownership**: Smart pointers like std::shared\_ptr and std::unique\_ptr manage ownership and prevent accidental misuse or *double-deletion*.

**Custom Deleters**: They allow custom resource cleanup, making them versatile for managing non-memory resources (e.g., file handles).

**Thread Safety**: std::shared\_ptr provides thread-safe reference counting.

1. How are static class members different from non-static class members?

// General: static class members belong to the class and are shared across all instances, while non-static members belong to individual objects and are unique to each instance

**Shared Across All Instances**:

Static Members: Shared by all objects of the class. They belong to the class itself, not any specific instance.

Non-Static Members: Each object gets its own copy, and they are tied to individual instances of the class.

**Access Method:**

Static Members: Can be accessed without creating an instance of the class, using the class name (e.g., ClassName::staticMember).

Non-Static Members: Must be accessed through an instance of the class (e.g., object.nonStaticMember).

**Memory Allocation:**

Static Members: Allocated only once, at class-level scope, and exist for the lifetime of the program.

Non-Static Members: Allocated separately for each object and exist only for the lifetime of the object.

**Initialization:**

Static Members: Must be explicitly initialized outside the class definition.

Non-Static Members: Initialized through constructors or default values inside the class.

**Context of Use:**

Static Members: Cannot access non-static members directly because they do not belong to an instance.

Non-Static Members: Can access both static and other non-static members of the class.

1. What is a characteristic of all keys for unordered maps?
   1. implementation operator<<
   2. halting function
   3. default constructor
   4. implementation operator <
   5. custom implementation (rule of 3)
   6. **a hash function**
2. Most algorithms take parameters that have iterators specifying the beginning and end of an iterator (e.g, std::sort(vec.begin(), vec.end())). What is the namespace of the version of the algorithm can be passed as a reference (for example: sort(v))?

// \*What namespace allows you to use sort(v) instead of v.begin(), v.end() in an algorithm?

* 1. (answer: **std::ranges**)

1. What command do you have to do prior to git commit?

**git add**

1. Instead of catching/throwing exceptions, how do you ensure no runtime cost using assertions?

// Use Compile-Time Assertions (**static\_assert**)

1. When will this run: if(cin>>i) ?

**Ans:** The condition `if(cin >> i)` succeeds (can "work") when valid input is provided that matches the type of `i`, the stream is in a good state, and there is data to read. It fails (cannot "work") when the input is invalid for the type of `i` (e.g., entering a string for an integer), the stream encounters an error (e.g., EOF or hardware failure), or the stream is already in a fail state.

1. v is a vector initialized with a size of 4: what values does v hold after iota(v.begin(), v.end() - 1, 0)

// **{0,1,2,0}, to be honest, the last value is vector default value, might be 0, or else**

1. Given code snippet:

vector<string> w = {“string”, “badsa”, “sja”};

String x = accumulate(w.begin(), w.end(), w.at(0), xxx)

What does the definition of xxx look like?

**// this is the xxx part code be like (concatenation)**

**auto xxx = [](const std::string& a, const std::string& b) {**

**return a + b;**

**};**

**// this is the xxx part code be like separator included**

**auto xxx = [](const std::string& a, const std::string& b) {**

**return a + " " + b;**

**};**

1. How many data members does the vector class in Section 6.2.1 of the required textbook hold?

class Vector {

public:

Vector(int s);

~Vector() { delete[] elem; }

Vector(const Vector& a);

Vector& operator=(const Vector& a);

double& operator[](int i);

const double& operator[](int i) const;

int size() const;

private:

double \*elem;

int sz;

};

// These **two** members. Double \* and int.

1. Why shouldn’t you always use -O3?

// increase the **compilation time**

// larger binary size

// Potential Runtime Performance Degradation

1. What is the instance of a class called?
   1. entry
   2. **object**
   3. return value
   4. address
   5. element
   6. pointer
   7. function
2. Why is it bad to reference local variables?

Ans: Referencing local variables is bad **because they are destroyed when the function ends, leading to dangling references, undefined behavior, potential memory corruption**, and hard-to-debug errors; instead, return by value or use dynamic allocation if persistence is needed.

1. Which of the following choices best describes friend?

**Ans:** **as a non-member function with access to the class’s private members**

1. Why should you put template in header?

**Ans: To allow access to the template definition during compilation**

1. When do we use int32\_t instead of int?

**Ans: We use int32\_t instead of int when we need an exact 32-bit integer type with predictable size across platforms, as opposed to int, whose size can vary depending on the system (e.g., 16-bit, 32-bit, or 64-bit)**

1. What characteristic does stable\_sort have but not sort?

Ans: The key characteristic of **stable\_sort** is that it preserves the relative order of equivalent elements, meaning if two elements are considered equal by the comparison function, their original order in the input sequence remains unchanged in the sorted output. In contrast, **sort** does not guarantee this stability, which can result in a different relative order for equivalent elements. This makes stable\_sort essential when maintaining the relative order of elements is important, such as when sorting records by one key while preserving the order of another key.

1. What is the scope of x declared in front of a function?

Ans: **global**

1. Given a vector of pointers to ints, which one of the following returns an int?

**a. vec[0]  
b. \*vec[0]  
c. vec.at(0)  
d. &vec[0]  
e. \*vec.begin()**

**Ans: b**

1. Given a vector of vector of ints, if the following code is passed in with indices that run over the size of the vector, what error will occur? (e.g., say our 2D vector is sized 4 by 4 -> if vec[5][5] is passed, what error)?

Ans: if it is vec[5][5], then it is **Undefined behavior;**

If it is vec.at(5).at(5), **Throws a std::out\_of\_range exception for indices out of bounds**

1. class Lunch{

~Lunch(); <- what is this

};

Ans: **Destructor**

1. Which of the following allows a class to act as a specialized version of a different class?
   1. Undefined behavior
   2. Streaming
   3. **Inheritance**
   4. Access
   5. Parameters
   6. Move operations
   7. Concentration
2. What is an equivalent declaration for the following lambda function:

auto func [] (string x, int i) { return x + to\_string(i); };

* 1. int func (string, int)
  2. void func (string, int, string)
  3. int func (string)
  4. int func (auto, string, int)
  5. char func (int, string)
  6. char func (int) -> string
  7. **std::string func(string, int)**

1. What does the expression new Dog[10] mean?

// The expression new Dog[10] means that:

**An array of 10 objects of type Dog is dynamically allocated on the heap.**

The Dog class's default constructor (if defined) is called for each of the 10 objects.

1. Why use O3 instead of the default g++?

// Using -O3 instead of the default g++ optimization level (-O0 or -Og) **provides aggressive compiler optimizations**, which can significantly improve the performance of your code in terms of execution speed and efficiency.

1. Consider the following implementation of the operator<< overload in a class MyClass. Why is this implementation of operator<< incorrect? How can it be fixed?

#include <iostream>

class MyClass {

public:

int data;

// Incorrect implementation

std::ostream operator<<(std::ostream os) {

os << data;

return os;

}

};

* 1. This implementation is correct as written.
  2. **The operator<< function should pass the std::ostream object by reference, not by value, to avoid unnecessary copies and preserve the state of the stream.**
  3. The operator<< function should return void instead of std::ostream.
  4. The operator<< function must be declared as const to work properly.

1. Referencing the read\_and\_sum function in section 2.3 of the required textbook, which utilizes a class named Vector created before it. What would be the difference if you used std::vector instead?

// Vector class in 2.3

class Vector {

public:

Vector(int s) :elem{new double[s]}, sz{s} { }

double& operator[](int i) { return elem[i]; }

int size() { return sz; }

private:

double \*elem;

int sz;

};

// Function referenced in problem

double read\_and\_sum (int s) {

Vector v(s);

for (int i = 0; i != v.size(); ++i)

cin >> v[i];

double sum = 0;

for (int i = 0; i != v.size(); ++i)

sum += v[i];

return sum;

}

**Ans**: Using std::vector instead of the custom Vector class would simplify the implementation by removing the need to **manually manage memory** and implement operators like []. Additionally, std::vector provides better safety, such as bounds-checking with .at() and automatic resizing.

**Memory Management**: std::vector handles memory allocation and deallocation automatically, reducing the risk of memory leaks and errors associated with manual memory management in the custom Vector class.

**Safety and Functionality**: std::vector includes built-in features like bounds-checking with .at(), dynamic resizing, and convenient member functions like .push\_back() and .size(), making it more robust and versatile than the manual implementation.

**Efficiency and Optimization**: std::vector is a part of the Standard Template Library (STL) and benefits from extensive optimization and testing, which ensures better performance and reliability compared to a custom implementation.

**Code Simplicity**: Using std::vector eliminates the need for boilerplate code to implement constructors, destructors, and other utilities, leading to cleaner and more maintainable code.

1. Consider the following C++ code snippet:

#include <iostream>

#include <map>

int main() {

std::map<std::string, int> myMap;

myMap["newKey"]; // No value explicitly assigned

std::cout << myMap["newKey"] << std::endl;

return 0;

}

What happens when the program tries to access the key "newKey" whose value does not already exist in the map? (as, when we access an entire key that doesn’t exist, a runtime exception (std::out\_of\_range) is thrown)

* 1. A runtime error occurs because the key does not exist.
  2. A compile-time error occurs because the key is not initialized.
  3. **A new key-value pair is created in the map, with the default value for the value type (int, which is 0 in this case).**
  4. Undefined behavior occurs because the key does not exist in the map.