* Introduction to the Standard Library :

→ Libraries implement programming solutions for much of the repetitive detail encountered in the development of modern applications.

String Library :

→ Provides support for 3 general types of strings

① string classes
② string_view classes
③ null terminated c-style string functions.

① String [#include <string>]

std::string : Stores sequences of characters using 'char'

→ Public member functions :

→ operator= : Assigns a string to the current string.
→ operator[] : Accesses a specified character in the string.
→ size() : Returns the number of characters in the string.
→ substr() : Returns a substring of the string.
→ find(char c) : Finds the first occurence of 'c' in the string.
→ rfind(char c) : Finds the last occurrence of 'c' in the string
→ find_first_of(char c) : Finds the first occurence of 'c'
→ find_last_of(char c) : Finds the last occurence of 'c'
→ operator+= : Appends a character or a string to the current string.

→ Helper functions -
→ operator== : Equality comparision
→ operator!= : Inequality comparision
→ operator>> : Extracts the string from the input stream
→ operator<< : Inserts the string into the output stream

NOTE: Unlike 'const char*', a string type keeps track of the no. of characters without using the null byte as a terminator. It can include one or more null bytes.

② string_view [#include <string_view>]

std::string_view : Non owning, read-only view into a contiguous sequence of characters with the first element at position zero.

→ Public member functions:
Similar to string classes, including assignment, access, size, substring and find operations.

→ Helper Functions:
Equality comparison, inequality comparision and insertion into the output stream.

NOTE: 'std::string_view' doesnot own the characters it points to.

DIFFERENCE BETWEEN STRING AND STRING VIEW IS IN NOTION IN DETAIL!

* Standard Template Library (STL) :

→ Most prominent part of the C++ Standard Library.

→ It consists of:
- → Container template classes
  - → sequential containers
  - → container adapters
  - → associative containers
- → iterators
- → algorithms
- → function objects

→ STL is used because its a good idea not to reinvent the wheel.
→ Reuse the components that are made to save time.

→ C++ 11 and C++ 17 library facilities are listed in OOP Notes.

* Containers and Iterations:

→ Containers:

→ Sequential Containers: STL provides sequential containers in the form of class templates. These templates include:

① Array:
→ contiguous storage of fixed size.
→ Memory for a 'std::array' is allocated on the stack, not on the free store (heap).
→ size cannot be changed dynamically.

E.g.
```
#include <array>
std::array<int,5> myArray = {1,2,3,4,5};
```

② Vector:

→ contiguous storage

→ 'std::vector' is a dynamic array that can change in size.

→ store their elements on free store (heap) and can adjust their size as required.

E.g.

```
#include <vector>
std::vector<int> myVector = {1,2,3,4,5};
myVector.push_back(6); //Dynamically resize to accomodate the
                       new element
```

→ Member functions include:

Refer Notion for full list of member functions

③ deque:

→ defines classes that manage doubly-ended queues

→ insertion and deletion of elements can happen at either side

→ Memory for 'std::deque' is allocated on the free store (heap).

E.g.

```
#include <deque>
std::deque<int> myDeque = {1,2,3,4,5};
myDeque.push_front(0); //Efficiently add an element at the front
```

→ Member functions include:

Refer Notion for full list of member functions

④ forward_list:
→ non contiguous storage of variable size, singly linked list

⑤ list:
→ non contiguous storage of variable size, doubly linked list
→ optimised for insertion and removal of elements anywhere throughout the list.
E.g.
```
#include <list>
std::list<int> myList = {1,2,3,4,5}
myList.push_front(0); // Efficiently add an element at the front.
```

NOTE: This template is sub-optimal for fast random access, for fast random access, the vector and deque templates are more efficient.

Example and member functions - in NOTION

→ Container Adapters:

STL includes adapters for converting a container class to operate in a specific context. Adapters include:
① stack - last in, first out (LIFO) context
② queue - first in, first out (FIFO) context
③ priority_queue - first element is always the "greatest"

① Stacks

→ 'std::stack' is a container adapter that operates in a Last In, First Out (LIFO) context.

→ It is implemented by default using 'std::deque' (double-ended queue) but can be configured to use other containers as well.

Member functions and e.g. in notion.

② Queues

→ 'std::queue' is a container adapter that operates in a First In, First out (FIFO) context.

→ It is implemented by default using a 'std::deque' but, like 'std::stack', can use other containers.

Member functions and e.g. in notion

→ Iterators

- An iterator is an object that points to an element in a sequence.
- STL iterators simulate sequential access to elements of STL containers, similar to raw pointers' access to simple arrays.
- Container classes that don't implement contiguous storage of elements require iterators to access their elements.
- We use iterators to insert elements into a sequence or to remove them from a sequence.

Takes the form

Container<type>::iterator identifier;

E.g.; to define an iterator named iter for a vector of double,

std::vector<double>::iterator iter;

++ → updates the iterator to pain to the next element
-- → updates it to point to the previous element
* → returns the value of the element pointed by the iterator.

** MEMBER FUNCTIONS AND EXAMPLE IN OOP NOTES

→ Member functions used for removing OR inserting elements in OOP notes.
2 More iterators examples
→ List Example / Deque Example → OOP NOTES

* Algorithms

→ The algorithms category of the C++ STL provides a variety of programming solutions that operate on ranges of elements within containers.
→ These are expressed in the form of functions within the std namespace.
Three libraries in this category are:
- functional standard function objects
- algorithm - standard algorithms
- numeric - standard numeric operations

Each library is independent of any other library.

→ Functional Library:

→ defines templated function objects that can be passed as arguments to other functions.
→ defined in the header file <functional> and consists of

• wrapper class templates

① std::function is a template that acts as a wrapper for function objects, allowing them to be assigned, copied and called like functions. It can wrap regular functions, functors or lambda expressions.

② std::reference_wrapper is a template class that facilates the copying and assigning of references. Allows references to be stored in containers and can be used anywhere a regular reference can be used.

Reference wrapper example in NOTES.

• Function Templates:

① std::bind binds one or more arguments to a function object, creating a new function object.

E.g.

```
double multiply (double x, double y ){ return x*y;}
int main()
{
    auto p = std::bind (multiply, 10, 3);
    std::cout << "Product= " <<(p)<<(std::endl;
}
```

Output:

Product=30

② std::ref returns an 'std::reference_wrapper' instance for the supplied argument.

Example in NOTES

⑧ • Operator Class Templates

These templates provide function objects for various operators allowing them to be used in algorithms that accept function objects or lambdas.
E.g. std::plus, std::minus, etc.

⑨ → Algorithm Library:

→ algorithm function templates perform common operations on ranges of elements in a sequence.
→ applies to not only containers, but also to strings and built-in arrays.

→ function templates are defined in header <algorithm>

⇒ Consists of
① Queries

→ count
• counts the number of occurences of a certain value in an array.
• Example in notes

→ count_if

- counts the total number of numbers provided certain conditions.

② Modifiers

→ copy

- std::copy is used to copy a range of elements from one container to another
- std::copy(source_begin, source_end, destination_begin);
- Example in Notes.

→ copy_if

- Similar to std::copy, but, ~~only~~ allows you to copy only those elements from a source range that satisfy a specified condition.
- Syntax:

```
std::copy_if(source_begin, source_end, destination_begin,
             condition);
```

- Example in notes

→ transform

- std::transform is used to apply a specified operation to each element in a range and store the result in another range.
- std::transform(source_begin, source_end, destination_begin, unary_operation);
- Example in notes.

③ Manipulators

→ sort

- std::sort is used to sort the elements of a container in ascending order.
- std::sort (begin, end);
- Example in notes.

→ Numeric library

→ provides standard templated functions for performing numeric operations on ranges of elements in a sequence.
→ #include <numeric> header

① accumulate

→ std::accumulate is used to compute the sum (or accumulate the values) of a range of elements.
→ Syntax
std::accumulate (begin, end, initial_value);
→ Example in notes

② inner_product

→ std::inner_product is used to compute the inner product of two ranges.
→ Syntax
std::inner_product (first1, last1, first2, init, binaryop1, binaryop2)

- first1, last1 : The range of the first sequence
- first2 : The beginning of the second sequence
- init : The initial value for the accumulator
- binaryop1, binaryop2 : Binary operations to apply.

→ example in NOTION

② partial_sum:

→ std::partial_sum is used to compute the partial sum of the elements in a range and store the results in another range.

→ Syntax:

std::partial_sum (first, last, result, binary_op);

- first, last: range of elements to be included in partial sum
- result: The beginning of the destination range where the partial sums will be stored
- binary_op: Binary operation function to be apply.

* ~~Creation Objects~~

* Raw Pointers:

→ In C++, a raw pointer is a built-in type that holds an address of a memory location in the abstract machine.

→ The pointer provides direct access to the object that occupies the region of memory that starts at that address.

→ Dereferencing the pointer accesses the value stored in the region of memory starting at the pointer's address.

→ C-Style Character Strings:

→ These strings are arrays of byte sized elements

→ Addresses of consecutive elements differ by one, that is the memory is byte-addressable memory.

→ Example in notes
→ std::hex manipulator specifies the current output format as hexadecimal.

String Literals
→ A sequence of characters surrounded by double quotes is called a string literal.
→ A string literal is an unmodifiable lvalue.
char *p = "Avoid overwriting"; //poor coding style
const char * p = "Avoid overwriting"; //good coding style

→ Expressions

→ Arithmetic Operations
① Addition
If one operand is of pointer type, the other must be of integral or unscoped enumeration type. The expression evaluates to the address that is the no. of types beyond the address stored in the pointed operand. That is, the integral operand gives the offset in types (not bytes).
Example in notes

② Subtraction
If the left operand is of pointer type, the right operand must be of integral type or of the same pointer type.
→ The result of subtracting an integral type from pointer type is the address that is the number of pointed-to types before the address stored in the left operand.
→ The result of subtracting a pointer type from another pointer type is an integer of synonym type 'ptrdiff_t'. This integer holds the no. of types (not bytes)

between the two addresses.

→ Postfix operations

Postfix expressions are useful for moving between adjacent type.

① Increment and Decrement

Dereferencing operator * has lower precedence than the post-fix operator ++. This means that the expression *s++ is evaluated as *(s++).

② Reference to a Pointer

Lets us change the address that has been stored in a pointer outside a function from within the function in the same way that a reference to a variable that holds a value.

Example in notes.

* Smart Pointers

→ Smart pointer is a container or a wrapper for a raw pointer.

→ Advantage: They deallocate memory automatically which avoids potential memory leaks.

→ Three types

① Unique Pointer → std::unique_ptr

② Shared pointer → std::shared_ptr

③ Weak pointer

Both are part of the <memory> header and offer different ownership semantics to cater to specific needs in C++ programming.

* Multithreading:

Multithreading is the ability of multiple parts of one program to be executed at the same time.

Example:

```
#include <iostream>
#include <thread>
void func1() {
    for(int i=0; i<7; i++)
        std::cout << "-";
}
void func2() {
    for(int i=0; i<7; i++)
        std::cout << "-";
}
int main() {
    std::thread worker1(func1);
    std::thread worker2(func2);
}
```

Output:

[The function which is fastest will complete first and both will run at the same time]

→ A multi threaded programming solution improves the elapsed time to complete execution by distributing independent tasks across separate hardware threads.

→ Performance:

Described on the basis of two aspects:

① Tasks:
Task represents group of instructions, it can refer to an entire program or a relatively small part of much larger program.

② Communications:
Occur across communication channels between a processor and main memory or a processor and a resource, such as a file stream, input stream or output stream.
→ Time taken to complete a communication is referred to as its latency.
→ The latency of a communication between a processor and a resource is atleast one order of magnitude greater than that between the processor and main memory.
→ Thus, accessing a resource can leave the processor idle, and the processor waiting for a communication to complete is available to execute another task or another program.
→ OS are designed to switch from one task to another which is termed as multi tasking.

- Problems of large size can consume enough time or space to make the program's performance a critical programming issue.

- The time and space that a program requires for its completion is described in terms of complexity.

① Time complexity refers to the rate at which elapsed time grows with problem size.

② Space complexity refers to the rate at which memory requirements grow with problem size.

→ Processes and Threads

① Process

- Process is an instance of a program executing on the host platform. It starts when the user loads an executable file into memory and execution begins. If a process requests a resource, the system's scheduler may temporarily switch to another process and return to the waiting one. This transition is called context switch. When a process finishes, control goes back to the OS, making memory available for a new process.

② Thread

A thread is a sequence of program instructions representing an independent flow of control within a process. Lightweight versions of process parts and can be of two types, hardware and software threads. A hardware thread is a

mini process that executes a software thread. Depending on the hardware available, the OS can schedule several threads for concurrent execution. This is called multithreading.

Terminology in multithreading

① Concurrent Task and Thread Spawning.

- A process starts with a single thread.
- When an instruction divides the execution path into concurrent tasks, it spawns a child thread.
- Reuniting of child and parent threads is called synchronization, with the point of reunion being the synchronization point.

② Shared Memory

- Multi-threaded programs share memory.
- Challenges arise when threads access shared variables, requiring careful attention for reproducibility and determine solutions.

③ Race Conditions

- Occurs when two threads can update the same memory location simultaneously.
- Results in different outcomes between runs due to unpredictable execution order.
- Techniques to avoid race conditions include shared states, mutexes, locks and atomics.

PGC 384

④ Shared States

- State shared by two threads to communicate a value.
- Used in C++ 11 thread libraries.

⑤ Mutexes, Locks and Atomics

- Mutex (mutual exclusion) excludes access by other threads to a memory location while one thread updates it.
- Implemented using locks, with the owning thread releasing it before others can acquire it.
- C++ 11 provides libraries for mutexes and locks.
- Atomics library for lock-free execution treats instructions as indivisible.

⑥ Deadlocks

- Occurs when two or more threads are waiting for each other to complete (execution) resulting in a perpetual (forever) blockage.

* Thread Classes

→ Thread and future libraries contain all the templates for simple multi threaded solutions. The thread library provides support for creating and managing threads that execute concurrently while future library provides support for retrieving the result from a function that has executed in the same or a concurrently executing thread. These libraries implement the Resource Allocation Is Initialization (RAII) idiom.

→ Thread Class:

- Defined in the header file <thread>.
- Thread object is either

① Joinable - represents thread which is actively running and ready to be executed. You can call join() on a joinable thread to wait for it to finish its execution before continuing with the rest of the program.

② Non-joinable - represents a thread that has already been joined and which has completed its execution.

Example in Notes

→ One cannot copy create a copy of a 'std::thread' object. This restriction is in place to avoid issues related to managing the ownership of the underlying thread.

→ std::thread::id type represents a thread identifier. Each thread has a unique identifier.

→ The class definition for std::thread includes an overload of the insertion operator (<<) for a right operand of type 'std::thread::id'. This allows us to display thread identifiers using 'standard::output.

→ std::thread class includes a template constructor for constructing objects that execute functions, function objects, or lambda expressions. Has the following form

```
template <typename Fn, typename... Args>
explicit thread (Fn&& fn, Args&&... args);
```

→ Here Fn is the type of function, function object or lambda expression and Args are the arguments passed to the function itself.

→ Thread identifier:
The thread identifier of a joinable thread object is accessible from within the function executing the thread's task. std::this_thread::get_id() returns this identifier.

→ future library:

→ <future> library provides functionality for asynchronous programming, allowing tasks to communicate through shared states. Key components of this library include 'std::future', providers like 'std::promise', 'std::packaged_task', and the 'std::async' function.

① 'std::future':
- Represents a value that will be available in the future.
- Handles results of async operations.

Providers (complement 'std::future'):

② std::promise:
- Creates or acquires a shared state.
- Defines 'set_value' to store a value in the shared state.

③ std::packaged_task:
- Wraps a task and a shared state.

④ std::async

- Launches a task asynchronously and returns a future for result retrieval.
- This function provides an extremely simple pair that spawns a thread to execute a task and creates a future for ~~ret~~ retrieving the return value from that task.