**ABOUT C++**

**1. WHAT IS C++?**

**C++ is an object-oriented programming language, which was developed by Bjarne Stroustrup.**

**2. WHAT ARE THE DIFFERENCES BETWEEN C AND C++?**

**C is a procedure-oriented programming language and does not have some features that C++ supports such as function and operator overloading, namespaces, virtual and friend functions, data hiding etc. Unlike C, C++ provides a high level of resources and memory management.**

**3. DATA TYPES IN C++?**

**We divide C++ data types into 3 groups:**

**3.1 PRIMITIVE TYPES:**

**Integer [(short int) 2 - (int) 4 - (long int) 8 bytes]: Whole numbers.**

**Character [(char) 1 byte]: Single-character.**

**Boolean [(bool) 1 byte]: True (1) or false (0).**

**Floating-point [(float) 4 bytes]: Numbers with decimals.**

**Double floating-point [(double) 8 bytes]: Numbers with decimals.**

**Void: Empty value, only pointers can be declared as void.**

**Wide Character [(wchar\_t) 2 bytes]: Stores more than one character.**

**3.2 DERIVED TYPES:**

**Function: It is derived from basic data types and performs some operations.**

**Pointer: It holds the memory address of another variable.**

**Array: It stores multiple values which are the same type.**

**Reference: It is the alias of an already existing variable.**

**3.3 USER-DEFINED TYPES:**

**Class: It is the blueprint of an object.**

**Structure: It is used to group several related variables.**

**Enumeration: It contains a fixed set of constants.**

**Union: Similar to “Structure” but its variables share the same memory location.**

**Typedef: It is simply giving another name to an existing data type.**

**4. WHAT IS NAMESPACE DIRECTIVE?**

**Namespace prevents name collision, especially when your code includes multiple libraries. The entire namespace can be declared as well as all the names in a namespace can be used individually with the “using” directive. For example;**

**using namespace std; or using std::endl;**

**5. WHAT ARE THE DIFFERENCES BETWEEN POINTER AND REFERENCE?**

**Unlike pointers(\*p), a reference(&r)**

* **must be declared and initialised at the same time,**
* **is not allowed to re-assign,**
* **does not need extra space on the stack, it shares its original variable memory location,**
* **cannot be NULL, it has to be a valid value,**
* **cannot be used with the arithmetic operators,**
* **is just an alias of an existing variable, not its memory address,**

**6. WHAT IS PASSING BY (VALUE, REFERENCE, POINTER, CONST REFERENCE, CONST POINTER)?**

**Passing by value means that a copy of the actual variable is passed into the function as a parameter, the actual variable does not get affected by the function.**

**Passing by reference means that the actual variable is passed into the function as a parameter and this variable can be modified when the function is called because we deliberately refer to the actual one by using reference. If we use “const” reference, the function cannot change the variable's value.**

**Passing by pointer is similar to passing by reference, so the pointer’s value can be modified permanently in the function. Unlike reference, we also can re-assign our pointer in the function but it is passed by value, which means that our pointer is copied before passing into the function, so the actual one, which is outside of the function, does not get affected. If we use “const” pointer, the function cannot change the memory address that the pointer points to.**

**6. WHAT IS AN INLINE FUNCTION?**

**If a function is inline, the compiler changes the normal calling process and that reduces the execution time and the program size because a separate location is not allocated on the memory for controlling the function code. Inline keyword is a suggestion, which means that the compiler can ignore it.**

**7. WHAT ARE STATIC\_CAST AND REINTERPRET\_CAST?**

**Static\_cast is used to convert one type into another. It is a compile-time conversion.**

**Reinterpret\_cast is used to the conversion of unrelated types (pointer to pointer, pointer to function, pointer to integral types etc.). It should not be used unless you do not have another option as it is inherently an unsafe cast type.**

**8. WHAT IS THE DIFFERENCE BETWEEN PREFIX AND POSTFIX?**

**If we use prefix, the value is incremented/decremented before being evaluated; this process works in reverse for postfix.**

**There is no difference between them, if they are used alone. (i++;) and (++i;) have the same effect.**

**9. WHAT ARE THE STACK AND HEAP?**

**Local variables and function parameters are stored contiguously on the stack and the allocation is done automatically by the compiler. The stack memory can also be described as temporary storage.**

**The heap is a dynamic memory that allows programmers to allocate/de-allocate memory manually and it also has more storage space than the stack.**

**10. WHAT IS THE DIFFERENCE BETWEEN NEW AND MALLOC()?**

**Both new and malloc() are used to allocate space on the heap. New is an operator which calls the constructor; however, malloc() is a C standard library function. In addition, new is faster than malloc().**

**11. WHAT ARE ACCESS MODIFIERS AND WHAT ARE THEY USED FOR?**

**Access modifiers (public, private, protected) indicate the accessibility of class members.**

**If a member is public, it is accessible to other classes or functions.**

**Private means that members are accessible only from inside the same class or friend functions.**

**Protected is similar to private; in addition, protected members can be accessed by the derived class members.**

**If a member’s accessibility is not specified, it is private by default.**

**12. WHAT IS THE DIFFERENCE BETWEEN STRUCT AND CLASS?**

**We can simply say that struct is a value type whereas class is a reference type. Hence, struct members are public and are stored on the stack while class members are stored on the heap. Unlike struct, class have 3 access modifiers which are public, protected and private.**

**13. WHAT ARE THE ADVANTAGES AND DISADVANTAGES OF USING POINTERS AS A CLASS MEMBER?**

**The advantages of using a pointer as a class member are lazy initialisation (to delay the creation of an object until it is used), reduction of header dependencies and giving more control to programmers.**

**The disadvantage of using a pointer as a class member is having to write our copy constructor and operators and that means more work, using a pointer can easily cause memory leak and accessing a pointer data member is slower than a non-pointer.**

**14. WHEN SHOULD WE DEFINE A DESTRUCTOR?**

**When we need to perform an action other than destructing class members, we define our destructor. For example, freeing memory.**

**When we destruct an object through a base class pointer, the destructor must be defined as virtual.**

**15. WHAT IS THE DIFFERENCE BETWEEN SHALLOW AND DEEP COPY?**

**Shallow copy can simply be defined as copying all data members of an object to create a new object but copied members still refer to the same location as its original; however, when we use deep copy, the object is copied and then located in a new allocated space in the memory. Unlike shallow copy, a deep copied object is independent of its origin.**

**16. WHAT ARE BASE CLASS AND ABSTRACT BASE CLASS?**

**A base class is the parent class, from which the other classes are derived and inherit members. An abstract base class is the base class which contains at least one pure virtual function.**

**17. WHAT IS RUN-TIME TYPE INFORMATION?**

**It is a mechanism that allows us to determine the type of an object at run-time. We can use dynamic\_cast (converts base class pointer to derived class pointer-downcasting) operator, type\_info (holds the type of an object) class and typeid (returns the type of an object) operator in this concept.**

**The base class must have one virtual function (run-time polymorphism) for the dynamic\_cast conversion to be successful.**

**18. WHEN SHOULD NOT WE CALL VIRTUAL FUNCTION?**

**Virtual functions should not be called in a constructor or a destructor because they are called in a certain order because of the nature of the inheritance, which means that virtual functions are not virtual during the construction and destruction processes.**

**19. WHAT IS THE DIFFERENCE BETWEEN CLASS AND TYPENAME IN TEMPLATE?**

**There is no difference between “class” and “typename” unless we use nested templates. We must use “class” in nested templates.**

**20. WHAT IS TEMPLATE-METAPROGRAMMING?**

**Template-metaprogramming is a technique that the program performs computations at compile-time in other words it runs at compile-time, to write a metaprogram, we use compile-time types and variables (const, constexpr) instead of run-time features (RTTI, virtual).**

**21. WHAT IS A METAFUNCTION?**

**A metafunction is actually a class/struct that can also return a value just as regular functions do and it is commonly used in the context of template metaprogramming.**

**For instance:**

template <int num1, int num2>

struct Add

{

    static constexpr int value = num1 + num2;

};

**“type\_traits” is a standard c++ library that provides many metafunctions.**

**22. WHAT IS EXCEPTION HANDLING?**

**When the program runs, unexpected problems may occur, in which case we use the exception handling technique to prevent the program from terminating with an error.**

**Exception handling consists of some keywords such as try, throw and catch:**

**“try” is a block of code which includes “throw” keywords, which means that the program may throw an exception inside the try block.**

**“catch” is used to catch the exception which is thrown and includes the next step which will be done if an error occurs.**

**23. WHAT IS ITERATOR?**

**An iterator is an object which points to elements in containers, thus they are used to move through the contents of containers.**

**24. WHAT IS LAMBDA EXPRESSION?**

**A lambda is a kind of function object which can capture variables in scope (a closure). It has value[=], reference [&] and both [v1, &v2] capture modes.**

**25. WHAT ARE SMART POINTERS?**

**Unlike raw pointers, smart pointers destroy themselves where they are out of scope so that memory leaks do not occur, in other words, programmers do not have to bother freeing memory.**

**There are 3 types of smart pointers:**

**unique\_ptr: It allows only one owner of the object.**

**shares\_ptr: It allows more than one owner that point to the object. Reference counter, which is accessed by using use\_count() method, includes the number of owners.**

**weak\_ptr: It is used with shares\_ptr but not added to reference counter.**

**26. WHAT IS THREAD?**

**Threads are parts of the process. We can create thread objects by using std::thread and pass callable objects(function pointers, function objects, lambdas, member functions) into them.**

**27. WHAT ARE SINGLE-THREADED AND MULTI-THREADED PROGRAMMING?**

**If threads execute in order, it is called single-threaded programming, which means that one thread waits to finish another thread before it runs; in contrast, detached threads run independently from other threads, if we divide one program into two or more separate parts by using detached threads, it is called multi-threaded programming.**

**We use join() and detach() methods to determine which way a thread works.**

**28. WHAT ARE MUTEX AND DEADLOCK?**

**Mutex prevents separate threads which share the same data resources from accessing the data at the same time.**

**In concurrent computing, if each thread waits for another waiting process to proceed, it is called deadlock because the program cannot make any progress.**

**29. WHAT IS RAII (RESOURCE ACQUISITION IS INITIALIZATION)?**

**It is a programming idiom which means that a resource is tied to the lifetime of the object, in other words, when the object goes out of the scope, its resources are released.**

**30. WHAT IS GENERIC PROGRAMMING?**

**Generic programming is to write algorithms which can work with all data types, because of which a generic program provides flexibility and reusability.**

**31. WHAT ARE TIME AND SPACE COMPLEXITIES?**

**Time complexity helps us to understand the relationship between the runtime (number of operations) and the input size of an algorithm.**

**Space complexity helps us to understand the relationship between the memory usage and the input size of an algorithm.**

**Both are expressed with Big O(worst case) notation. Best to worst time complexity (n: input number): Constant Time O(1), Logarithmic Time O(log n), Linear Time O(n), Quadratic Time O(n^2), Exponential Time O(2^n), etc…**

**We choose the best solution for our problem by considering the time and space complexity of algorithms so that we can write more efficient programs.**

DATA STRUCTURES

ARRAY

Each element can be accessed directly

Contiguous in memory

The name of the array is the location of the first element (index 0)

No bounds checking

Fixed Size (No insertion or deletion after declaration)

Lookup by index, Lookup by value – O(**1)**

Space Complexity O(n)

**Corresponds to (STL) std::array:**

Contiguous in memory

Direct elements access

All iterators are available and do not invalidate

arr.front(), arr.back(), arr.size(), arr.at(index), arr{index], arr.data(),arr.empty(), …

[https://en.cppreference.com/w/cpp/container/array](https://en.cppreference.com/w/cpp/container/array%20)

**Dynamic Array (STL) std::vector:**

Dynamic Size

Contiguous in memory

Direct elements access

All iterators are available and may invalidate

Append, Remove Last, Lookup by Index – O(1)

Prepend, Remove First, Insert, Remove, Lookup by value – O(n)

Space Complexity – O(n)

v.front(), v.back(), v.push\_back(obj), v.pop\_back(), v.emplace\_back(obj), v.size(), v.capacity(),

v.empty(), …

<https://en.cppreference.com/w/cpp/container/vector>

LINKED-LIST - **class**

Consists of nodes having value and pointing next node

Not contiguous in memory

Direct access to elements is not provided

Append, Prepend, Remove First – O(1)

Remove Last, Insert, Remove, Lookup by Index, Lookup by Value – O(n)

Space Complexity – O(n)

**Corresponds to (STL) std::forward\_list:**

Direct access to elements is not provided.

Reverse iterator is not available, iterators invalidate when the corresponding element is deleted.

Insertion and deletion – O(1)

fl.front(), fl.push\_front(obj), fl.pop\_front(), fl.emplace\_front(obj), fl.insert\_after(it, obj), fl.emplace\_after(it, obj), fl.erase\_after(it), fl.resize(int), …

<https://en.cppreference.com/w/cpp/container/forward_list>

DOUBLY LINKED-LIST – **class**

Consists of nodes having value and pointing next and prev nodes

Not contiguous in memory

Direct access to elements is not provided

Append, Prepend, Remove First, Remove Last – O(1)

Insert, Remove, Lookup by Index, Lookup by Value – O(n)

Space Complexity – O(n)

**Corresponds to (STL) std::list:**

Direct access to elements is not provided.

Iterators invalidate when the corresponding element is deleted.

Insertion and deletion – O(1)

l.front(), l.back(), l.push\_front(obj), l.push\_back(obj), l.pop\_front(), l.pop\_back(), l.emplace\_front(obj), l.emplace\_back(obj), l.insert(it, obj), l.erase(it), l.size(), l.resize(int), …

<https://en.cppreference.com/w/cpp/container/list>

STACK

LIFO (Last In First Out)

Can be implemented with **a vector** **or linked-list**.

Operations occur on the head for linked-lists and the back for vectors.

Insertion and deletion – O(1)

Space complexity – O(n)

**Corresponds to (STL) std::stack:**

Can be implemented as a vector, list or deque.

Iterators are not provided.

s.push(obj), s.pop(), s.emplace(obj), s.top(), s.empty(), s.size(), …

<https://en.cppreference.com/w/cpp/container/stack>

QUEUE

FIFO (First In First Out)

Can be implemented with **a linked-list**.

Elements are pushed at the tail, popped from the head.

Insertion and deletion – O(1)

Space complexity – O(n)

**Corresponds to (STL) std::queue:**

Can be implemented as a list or deque.

Iterators are not provided.

q.push(obj), q.pop(), q.emplace(obj), q.front(), q.back(), q.empty(), q.size(), …

<https://en.cppreference.com/w/cpp/container/queue>

TREE - **class**

Consists of nodes having value and pointing (left, right) nodes(children) - (binary tree)

Can be Full (each node points to either 0 or 2 nodes),

Perfect (all levels completely filled),

Complete (filled from left to right)

The top node is called root, a node that does not have any child is called leaf

Binary Search Tree: Right child is grater, left is lower

Can’t have duplicate elements

Lookup, Insert, Remove – O(log n)

(In the worst case, binary search tree treats as linked-list - O(n))

Space Complexity – O(n)

HASH TABLE - **class**

We can use a fixed-size (node\*) array - each index has a linked-list of key-value pair(s) (collision – Separate Chaining Method)

We use hash function to determine the address of a pair in hash table

[If the address is not empty, we move through the indexes until we find an empty address(Linear Probing - Open Addressing Method)]

Set, Get – O(1)

Space Complexity – O(n)

We can also use **std::underored\_map** to implement hash table

GRAPH

Consists of vertices(key) and edges(std::unordered\_set of values)

Can be implemented adjacency matrix (2-dimensional array) or

adjacency list (**std::unordered\_map**) – more efficient

Add Vertex, Add Edge, Remove Edge – O(1)

Remove Vertex – O(n)

Space Complexity – O(n)

HEAP

Is a complete binary tree

Can have duplicates

Parents must be greater/lower than their children (max heap/min heap)

Can be implemented with **a vector**: leftChild = 2 \* parentIndex + 1

rightChild = 2 \* parentIndex + 2

parentIndex = (childIndex-1) / 2 (for both)

Can be used to implement priority\_queue (more efficient)

Insert, Remove - O(log n)

Space Complexity – O(n)

**Corresponds to (STL) std::priority\_queue:**

Is implemented as a vector

Elements are stored in order (front is the largest)

Iterators are not provided

Insertion and deletion – O(n)

p\_q.top(), p\_q.pop(), p\_q.push(), p\_q.size(), p\_q.empty(), …

<https://en.cppreference.com/w/cpp/container/priority_queue>

OTHER STL CONTAINERS

**std::deque:**

Not contiguous in memory

Direct elements access

All iterators are available and may invalidate

Append, Prepend, Remove Last, Remove First – O(1)

Insert, Remove – O(n)

Space Complexity – O(n)

d.front(), d.back(), d.push\_back(obj), d.pop\_back(), d.push\_front(obj), d.pop\_front(), d.emplace\_front, d.emplace\_back(obj), d.size(), …

[**https://en.cppreference.com/w/cpp/container/deque**](https://en.cppreference.com/w/cpp/container/deque)

**std::set:**

Ordered by key (operator <)

No concept of front and back.

No duplicates.

Iterators invalidate when the corresponding element is deleted.

s.insert(obj), s.erase(obj or it), s.clear(), s.find(obj), s.empty(), s.count(obj), …

<https://en.cppreference.com/w/cpp/container/set>

(std::unordered\_set, std::multiset, std::unordered\_multiset)

**std::map:**

Elements stored as (key, value) pairs

Ordered by key

No duplicates

No concept of front and back.

Direct element access by using key

Iterators invalidate when the corresponding element is deleted.

Most operations are very efficient.

m.insert(key), m.erase(key or it), m.clear(), m.find(key), m.empty(), m.count(key),

m.at(key), m[key], m.size() …

<https://en.cppreference.com/w/cpp/container/map>

(std::unordered\_map, std::multimap, std::unordered\_multimap)

ALGORITHMS

SEARCHING:

LINEAR SEARCH

We iterate through all elements to check if the current one is the target.

Time complexity - O(n)

Space complexity - O(1)

BINARY SEARCH

We choose a side in an array to look for the target by checking whether the middle element is smaller or bigger than the target, then do it again until the target is found.

Mid = low + (upper - lower) / 2

Time complexity - O(log n)

Space complexity - O(1) (iterative) / O(log n) (recursive)

**Interpolation search:**

Binary search with different formula

Mid = Low + (High - Low) / (Arr[High] – Arr[Low]) \* (X – Arr[Low])

Time complexity - O(log(log n))

BREADTH-FIRST SEARCH

Row by row searching

We use queue<Node\*>: when we pop a (parent) node, we push (its children) into the queue. (For graph: visited and its neighbours)

Time complexity - O(V+E)

Space complexity - O(V)

DEPTH-FIRST SEARCH

Going to the deepest node (leaf) before backtracking (tree)

We can use recursion to implement preorder, Inorder and postorder DFS (tree).

We use stack<Node\*>: we visit every node and push it into stack, when we reach a node that does not have any unvisited neighbours, we return to the previous one by popping to find an unvisited node (graph).

Time complexity - O(V+E)

Space complexity - O(V)

SORTING:

BUBBLE SORT

Always start with the first item and compare it to the next one: if it is greater, swap them and continue until array is sorted.

Example: Array {2,6,9,1,7,3}

When the process(inner loop) is complete for the first time: {2,6,1,7,3,9}

When the process(inner loop) is complete again: {2,6,1,3,7,9}

This means that the greatest item among unsorted items goes to the end every time the process(inner loop) is repeated.

Time complexity – O(n^2)

For almost sorted arrays – Ω(n) (Best Case)

Space complexity – O(1)

SELECTION SORT

Look for the index that has min value: Set minIndex to the first item(0) and compare it to other items, if an item is smaller than it, set the index of the smaller one to minIndex, after checking all items, swap the minimum value and the first item.

Repeat the process for all items.

Time complexity – O(n^2)

Space complexity – O(1)

INSERTION SORT

Always start with the second item and compare it to the previous one: if it is smaller, swap them and continue until array is sorted.

Example: Array {2,6,9,1,7,3}

When the process(inner loop) is complete for the first time: {1,2,6,9,7,3}

When the process(inner loop) is complete again: {1,2,6,9,7,3}

This means that the smallest item among unsorted items goes to the beginning every time the process is repeated.

Time complexity – O(n^2)

For almost sorted arrays – Ω(n) (Best Case)

Space complexity – O(1)

MERGE SORT

Combine two sorted arrays into one sorted array.

Helper function ‘’Merge’’: Cuts array in half and create two new arrays

Merge all sub-arrays until we have an original sorted array

Function ‘’Mergesort’’: Calculates midIndex to determine the bound of sub-arrays(recursive function) and calls merge function.

Time complexity – O(n.log n)

Space complexity – O(n)

QUICK SORT

Helper function ‘’Pivot’’: All items smaller than pivot are on left and all items greater than pivot are on right. (Return pivotIndex)

Function ‘’Quicksort’’: Call pivot function and sort the pivot's left and right sides (recursive function)

Time complexity – Ω(n.log n)

For almost sorted arrays – O(n^2) (Worst Case) – Use bubble or insertion instead.

Space complexity – O(n)

**MULTITHREADING**

**(Notes from Modern C++ (cpp) Concurrency - Mike Shah -** [**https://www.youtube.com/playlist?list=PLvv0ScY6vfd\_ocTP2ZLicgqKnvq50OCXM**](https://www.youtube.com/playlist?list=PLvv0ScY6vfd_ocTP2ZLicgqKnvq50OCXM)**)**

**Parallelism:** Everything happens at once, instantaneously, completely separate.

**Concurrency:** Multiple things can happen at once, sometimes tasks have to wait on shared resources.

Create multiple threads (on the same core) and use std::mutex to prevent data race.

For instance;

std::mutex can be manually locked and unlocked

std::mutex lock; //creation of a mutex

void shared\_value() {

lock.lock();

some code…

lock.unlock();

}

or we can use lock\_guard at the beginning of the function:

std::lock\_guard<std::mutex> lockGuard(lock);

or use atomic variables to make accessing shared data(primitive types) safe:

std::atomic<int> name;

Use condition\_variable to prevent threads from waiting to access data and wasting CPU cycles. We need a boolean, condition\_variable and lock to implement this:

std::mutex lock;

std::condition\_variable c\_variable;

bool notified = false;

Reporter thread {

std::unique\_lock<std::mutex> uniqueLock(lock);

if(!notified)

c\_variable.wait(lock);

do something..

}

Worker thread {

std::unique\_lock<std::mutex> uniqueLock(lock);

do something…

notified = true;

c\_variable.notify\_one();

}

std::async: Launch a thread that will not be blocked.

std::future: It is used to access the results that async operations produce.

Basic syntax;

std::future<int> asyncFunction = std::async(std::launch::async, &func, Args);

some code….

Int result = asyncFunction.get(); // We wait until the function is complete.

For example;

#include <iostream>

#include <future>

#include <chrono>

bool task1 (int x) {

for(int i = 0; i < x; i++) {

std::cout << "loading\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(x));

}

std::cout << "Loading completed\n";

return true;

}

int main() {

std::chrono::time\_point start = std::chrono::high\_resolution\_clock::now();

std::future<bool> control = std::async(std::launch::async, &task1, 20);

std::future\_status status;

while(true) {

std::cout << "Main is running\n";

status = control.wait\_for(std::chrono::milliseconds(1));

if(status == std::future\_status::ready){

std::cout << "Program is done\n";

break;

}

}

std::chrono::duration<double> eTime = std::chrono::high\_resolution\_clock::now() - start;

std::cout << "Elapsed time: " << eTime << "\n";

return 0;

}

<https://en.cppreference.com/w/cpp/thread>

**CODE OPTIMIZATION**

**(Notes from The Most Important Optimizations to Apply in Your C++ Programs - Jan Bielak - CppCon 2022** [**https://www.youtube.com/watch?v=qCjEN5XRzHc**](https://www.youtube.com/watch?v=qCjEN5XRzHc) **and more)**

1. **Compiler optimization (common flags):**

* MSVC: /O1 < /O2 < /Ox

cl /O2 main.cpp

<https://learn.microsoft.com/en-us/cpp/build/reference/o-options-optimize-code?view=msvc-170>

1. **Set target architecture:**

Compilers could be optimized for a specific CPU architecture. Determine the CPU(s) on which your code will be run.

* gcc: -march=native(code may not run on other CPUs) , -mtune=native (better performance on your CPU)

[https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/Submodel-Options.html#Submodel-Options](https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/Submodel-Options.html#Submodel-Options )

* MSVC:

/arch options: Target architecture for the program.

<https://learn.microsoft.com/en-us/cpp/build/reference/arch-minimum-cpu-architecture?view=msvc-170&redirectedfrom=MSDN>

and -MACHINE: Target platform for the program.

<learn.microsoft.com/en-us/cpp/build/reference/machine-specify-target-platform?view=msvc-170&redirectedfrom=MSDN>

1. **Use fast math:**

* MSVC:

Compilers use /fp:precise by default which preserves ordering and rounding properties.

/fp:fast ignores them and makes code faster and smaller. Be careful to use!

<https://learn.microsoft.com/en-us/cpp/build/reference/fp-specify-floating-point-behavior?view=msvc-170>

1. **Unity builds:**

Header files are parsed once, which means that the compiler is invoked fewer times. It makes linking faster because of the smaller total size of object files.

1. **Use constexpr, const, noexcept and assume**
2. **Mark pointers restrict, functions as pure(MSVC does not support it)**
3. **Function parameters**
4. **Avoid allocation and deallocation on the heap (new and delete are slow) (Prefer stack allocation)**
5. **Avoid unnecessary copying (exceptions, in lambda captures, in structured bindings)**
6. **Exploit data locality:**

Reuse data as much as possible rather than creating new copies.

Loop through an array linearly rather than jumping around randomly.

1. Avoid false sharing
2. Avoid indirected calls
3. Make branches predictable
4. Use branchless optimization
5. SIMD intrinsics
6. **Recursion is worse than iteration (Call stack problem)**
7. **Memory alignment:**

* Data structures should be aligned properly to cache lines. For example, int array[16] = 16x4 = 64 byte.
* The number of elements should be a power of 2.
* Use ‘’alignas(int constant)’’.

1. **Use cache-friendly structure:**

* Prefer structure of arrays to array of structures.

1. **Lock-free programming: Locking causes latency.**
2. **Use initializer lists and std::move.**
3. **Use reserve() method to avoid expensive reallocation.**

**LOW LATENCY**

**(**Notes from **CppCon 2017: Carl Cook “When a Microsecond Is an Eternity: High Performance Trading Systems in C++”** [**https://www.youtube.com/watch?v=NH1Tta7purM**](https://www.youtube.com/watch?v=NH1Tta7purM)**)**

The factors that have impacts on the program (apart from language):

Compiler and version

Machine architecture

Third-party library

Build and link flags

**Low latency programming techniques:**

**1-** Use Errorflags to check if there is an error instead of checking for all events:

If(!errorflags) Proceed;

else errorhandle(errorflags);

Error handling code should not be inlined.

**2-** Virtual functions make codes slower.

**3-** Lambda functions are faster.

**4-** Memory allocation is costly:

* Use a pool of preallocated objects.
* Reuse objects instead of deleting them.
* If deletion is a necessity then do this from another thread.

**5-** We can use exceptions because they are zero cost if they do not throw.

But don’t use exceptions for control flow.

**6-** Prefer templates to branches (if statements).

**7-** Multithreading is the best avoided for low latency:

* Locking is expensive.
* It is complex to implement parallelism correctly.

When we use multithreading:

* Keep shared data between ‘’hotpath’’ and everything else to a minimum.

(Multiple threads at the same cache line get expensive)

* Pass copies of data instead of sharing data.

(single writer, single reader lock-free queue)

* If we must share data, we should not use synchronization.

**8-** std::unordered\_map is faster container.

(hybrid hash map approach: both chaining and open addressing)

**9-** Use benchmark and measure your code (and profiling).

**10-** Don’t share L3: disable all cores except 1 core. Choose neighbours carefully.

**11-** Consider inplace\_function over std::function because std::function may allocate.

**12-** Keep the cache hot.

**13-** std::pow can be slow.

**14-** Don’t do system calls.