Here are some comments on C++ as described in “A Tour of C++ (2nd Edition)”. I ask the students to list what is new or surprising to them. What you see below is a representative sample. I use this as a diagnostics test to help focus my course.

I must emphasize that these are \*good\* students. All are volunteers in the sense that this isn’t a compulsory course. Almost all final-year undergraduates or first-year graduate students in the computer science department of Columbia University. Many (especially of the graduate students) have industrial experience. Most of us would be happy to have colleagues with their level of energy and education. In previous years, almost everyone has gained good grades after an ambitions final project.

Of course, I obtained their permission to share their answers in an anonymized form.

I share it to help give a sample of what today’s students know about C++.

Note that this is \*before\* I have taught them anything.

Draw your own conclusions.

PLEASE DON’T SHARE THIS OUTSIDE THE COMMITTEE

Xxxx

1.2

* Does the ability to run C++ source code on different hardware and operating systems sometimes produce different results for the same code? How is this handled? Is this problematic?
* What is the benefit of C++ using static type instead dynamic? Faster compiling time? Or just readability?

1.2.1

* I had no idea what the purpose of using std::. I would just do it to get the code to compile. Are there other parameters that can be specified instead?
* What is the difference between include vs import? I have seen both in iOS as well but never deep dived into it.

1.3

* I understand overloading, but does C++ also support generics? That way there can be a print function that takes in an Any, AnyType, Object, etc. and then internally the method can decide the type and print using the correct private overloaded print function?

1.4

* I understand types and memory.

1.4.1

* I have very little experience using bitwise operators. Is there a benefit? Is it worth learning?

1.4.2

* I did not know that you could initialize using braces instead of an =. And it is safer because it throws an error instead of converting types.
* I was unaware of the auto which can type inference. Very cool.

1.5

* I understand name spacing.

1.6

* I’ve used const but never heard of constexper.

1.7

* In the past, I’ve used \* and & almost interchangeably to get my code to work in C++ or C (or very rarely in Objective-C) when \* was not working. Usually for code that did swaps or something similar. I did not realize one pointed to the contents and the other was the address.

1.7.1

* I’ve never used or heard of nullptr. I’ve always used NULL.

1.9

* I was unaware that C++ is constructed to the hardware to allow such low-level control. This is definitely a language that I want to learn. I would like to transition from developing apps to engineering real software.

1.9.2

* I am still a little confused by the difference between assignment and initialization. Does assignment mean that a variable needs a hard-set value whereas initialization can also mean a pointer reference to another variable with a hard-set value?

2.1

* I am familiar with classes and enums.

2.2

* I have rarely used structs. I almost always use classes. In Swift, we have Structs which function almost identically to Classes, including methods. But I believe the difference is that a Struct in Swift cannot be used as a pointer. I know this course is about C++ but I am trying to translate some of my prior knowledge. Could you explain the difference in class in more detail? I understand that in 2.3 you state that a class is a struct with public and private methods and properties.

2.4

* I have never heard of a unions or variants in C++ or any other programming language.

3.2

* I am familiar with the concept of header files from Objective-C. Seems similar.

3.3

* I’ve never used modules. Can you explain it more in detail? Does it just create the library when compiled for other programs to use?
* What’s the difference between import and include? I understand that include is older and should be avoided but it is still unclear to me the benefit of import.

3.4

* I’ve never used namespace except for using namespace std to simplify my code. Could we go over the benefit of namespacing in greater detail?

3.5.1

* I’ve used try catch blocks and exception handling but never come across the concept of RAII.
* Never seen noexcept before either.

3.6.1

* It did not occur to me that usually large values are passed by reference instead of value.

3.6.2

* I did not realize it could be more beneficial to use + rather than add. I always considered them the same function.

3.6.3

* I am unfamiliar with structured binding.

4.2

* Concrete class is a term that I have never heard before. I did not realize that vectors and strings were not built in types. Especially since strings can be passed into methods instead of pointers.

4.2.1

* I’ve never used complex before or seen it.

4.2.3

* I have never used or seen static\_cast, reinterpret\_cast, or const\_cast.

4.3

* A protocol is an abstract class?
* The virtual label is new to me. I have never built an abstract class in C++.

4.4

* I did not know the implementation technique for compilers to breakdown virtual functions. vtbl are new concept to me. Interesting.

4.5.1

* I knew that objects were built bottom up but I did not realize that they were destroyed top down. I suppose it makes sense since you push to the stack to initialize then pop to de-initialize.

4.5.3

* I’ve never used unique\_ptr. Clever solution to memory leaks.

5.1.1

* Move assignments are a new concept to me.
* The rule of zero is new to me.

5.1.2

* The explicit label is new to me.

5.2.2

* I have very little experience with move constructors. Unfortunately, I usually just assign and return without thought of how expensive an operation is.

5.3

* I understand multi-threading, but it’s very complex and difficult to do well so I would love to learn best practices.
* I use ARC in iOS. Is it really worth handling your own memory management to get better performance? Or does it depend on what you are building? IE: An App crashing is no big deal vs. a smart car crashing is a very big deal
* I would love to learn more about RAII

6.2

* I’ve never created my own template in C++ (or any custom class in any language). But I am familiar with using it to create objects like a dictionary.

6.2.3

* I have new built a deduction guide but I like it. Could we go into more detail on best designs.

6.3.2

* Function objects are new to me. Like a completion block?

6.3.3

* Lambda expressions are a new term to me but I am familiar with the implementation.

6.4.3

* I’ve done compile time checks on iOS (Ex: if the OS is iOS or tvOS) but not with C++ so I was unfamiliar at how or why to do this.

7.2.1

* I have never used the C++ requirements clause

7.2.3

* Likewise, requires requires is new to me.

7.3.2

* Lifting is a new concept to me.

7.4.1

* Fold is a new term to me. But I am familiar with what it refers to.

Xxxx

* Several sections discuss the downsides of enforcing backwards compatibility with C (e.g. brace vs. equality initialization of ints vis a vis downcasting), although some backwards compatibility is not enforced (typecasting). I'm curious to know how backwards compatibility is treated as a design consideration. [page 8]
* The entire language seems geared towards compile-time programming (constexpr, generics, the template meta-language). It's amazing elaborate, but also intimidating.
* I didn't realize you could iterate over C-style arrays with known length using the enhanced for-loop. [page 12]
* Why is RAII really so important for exception handling? If an exception is thrown, doesn't the program almost always terminate? So leaking memory or dangling file pointers isn't a problem. [page 36]
* The array of core methods for classes is crazy. Constructors, copy constructors, move constructors, copy assignment, move assignment, destructors. I have heard about the rule of 3 and the rule of 5, but when should these be defined in user code?
* How does returning by value work in terms of optimization. For example, for something like

Vec Vec::operator + (const Vec & v) {Vec v2 = …; return v2};

that returns a Vector by value, does the compiler actually perform a full copy when it returns, or does the compiler perform RVO and elide the copy entirely, or is the move operator invoked to move v2 out of scope? Is there any danger to this kind of thing? Should I implement a move operator to be safe? Is there a default move operator to handle things like this? [page 66]

* When do we use virtual destructors?
* For template arguments, why do we do template<typename T, int n> instead of template<typename T> Foo{ Foo(int n); }? Is it just to enforce that n must be known at compile time? [page 82]
* Lambdas are cool. The syntax is really beautiful, and it's very neat being able to capture symbols from the external scope either by reference or value (although it might just be easier to capture everything by reference and copy as needed).

Xxxx

• 1.3 “If two functions are defined with the same name, but with different argument types,

the compiler will choose the most appropriate function to invoke for each call.”

• 1.4.2 “The = form is traditional and dates back to C, but if in doubt, use the general {}-

list form. If nothing else, it saves you from conversions that lose information.”

• 1.4.2 “With auto, we tend to use the = because there is no potentially troublesome type

conversion involved”

• 2.2 “The & in Vector& indicates that we pass v by non-const reference (§1.7); that way,

vector\_init() can modify the vector passed to it.”

• 2.4 Unions

• 4.3 “The curious =0 syntax says the function is pure virtual; that is, some class derived

from Container must define the function. Thus, it is not possible to define an object that is

just a Container.”

• 4.3 “The point is that use(Container&) has no idea if its argument is a Vector\_container,

a List\_container, or some other kind of container; it doesn’t need to know. It can use any

kind of Container. It knows only the interface defined by Container. Consequently,

use(Container&) needn’t be recompiled if the implementation of List\_container changes

or a brand-new class derived from Container is used.”

• 4.5.2 “We can ask ‘‘is this Shape a kind of Smiley?’’ using the dynamic\_cast operator:”

• 4.5.3 “One simple solution to such problems is to use a standard-library unique\_ptr

(§13.2.1) rather than a ‘‘naked pointer’’ when deletion is required.”

• 5.2.1 “Fortunately, the fact that Vector has a destructor is a strong hint that the default

(member-wise) copy semantics is wrong and the compiler should at least warn against

this example.”

• 5.3 “Resource handles, such as Vector and thread, are superior alternatives to direct use

of built-in pointers in many cases. In fact, the standard-library ‘‘smart pointers,’’ such as

unique\_ptr, are themselves resource handles.”

• 6.3.2 “One particularly useful kind of template is the function object (sometimes called a

functor), which is used to define objects that can be called like functions. “

• 6.3.3 “The notation [&](int a){ return a<x; } is called a lambda expression.

• 6.4.2 “In fact, every standard-library container provides value\_type as the name of its

value type .”

• 7.2.3 “A requires−expression is a predicate that is true if the statements in it are valid

code and false if they are not.”

• 7.3.2 “The process of generalizing from a concrete piece of code (and preferably from

several) while preserving performance is called lifting.”

• 7.4.1 “Fold is a very powerful abstraction, clearly related to the standard-library

accumulate(), with a variety of names in different languages and communities.”

Xxxx

1.4.2 Initialization

* Did not know {} initialization had type-safety unlike the traditional =.
* I think it’s important to address what happens when user uses “auto” with = and {} like:
“auto x = {true};”
In this case, “x” type will be deduced to be std::initializer\_list<bool>. This behavior would of course not be the case if we wrote:
“bool x = {true};”

3.3 Modules

* YESSS I fully agree with this approach. I gradually felt the power of Python over the years and wanted to port its features into C++ to create a script-like but compiled, hybrid-language. A similar feature that aims to be more Pythonic is “auto”, which is traditionally used to hide types.

3.6.3 Structured Binding

* Structured binding is also quite Pythonic. This seems cool, but difficult to maintain. If many functions use this structured binding on a user-class A and later we add another member to A, we have to change all of these functions to have a third binding. Whereas, the more tedious but safer approach of aliasing members such as:

auto& x = a.name;
auto& y = a.size;

will never have this issue.

6.2.3 Template Argument Deduction

* Did not know users could add template argument deduction rules; though I feel like most problems can be cleanly solved without this feature.

7.2.2 Concept-based Overloading

* Finally, a way to get rid of SFINAE!

Xxxx

|  |  |  |  |
| --- | --- | --- | --- |
| No. | content | chapter | page |
| 1 | single quote as digit seperator | 1 | 6 |
| 2 | **constexpr** function | 1 | 10 |
| 3 | **variant** | 2 | 26 |
| 4 | **module** (C++20) | 3 | 32-34 |
| 5 | **static assert** | 3 | 41 |
| 6 | structured binding | 3 | 45 |
| 7 | **std::initializer˙list** | 4 | 52 |
| 8 | **dynamic˙cast** for hierarchy navigation | 4 | 61 |
| 9 | move constructor, move assignment | 5 | 66 |
| 10 | **=default**, **=delete** | 5 | 67 |
| 11 | member initializers | 5 | 68 |
| 12 | user-defined literals | 5 | 75 |
| 13 | constrained template arguments | 6 | 81 |
| 14 | compile-time if | 6 | 91 |
| 15 | concepts (C++20) | 7 | 94 |
| 16 | concept-based overloading (C++20) | 7 | 96 |
| 17 | **requires**-expression | 7 | 97 |
| 18 | variadic templates **typename...** | 7 | 101 |
| 19 | fold expressions | 7 | 102 |

Xxxx

Chapter One

● Didn't know about std::iota (1.3)

● Narrowing conversions not allowed in bracket notation (1.4.2)

● Sticking an assignment into a conditional to check for 0 or the nullptr is something I have

never seen in code and I'm surprised it's preferred (1.8)

Chapter Two

● Some of the differences between enum class and C enum were new (2.5)

Chapter Three

● Modules/Import are new and sound pretty cool (3.3)

● I know about exceptions but have no experience using them (3.5.1)

● I didn't know that throwing exceptions from a constructor was okay (3.5.1)

● std::terminate is new to me (3.5.3)

● I wouldn't have assumed assert to be a debug-only check (3.5.5)

● static\_assert is interesting (3.5.5)

● I haven't seen a structured binding used on a class; that's interesting (3.6.3)

Chapter Four

● delete[] to delete an array (4.2.2)

● std::initializer\_list (4.2.3)

● dynamic\_cast is new to me (4.5.2)

Chapter Five

● Being explicit about copy and move operations if a class has a raw pointer member

(5.1.1)

● I didn't know all the literal type suffixes, such as s for string. (5.4.4)

Chapter 6

● Constrained templates and concept definitions (6.2.1)

● Value template arguments are new to me (6.2.2)

● Deduction guides are new to me (6.2.3)

● Compile time if (if constexpr) (6.4.3)

Chapter 7

● Concepts again (7.2)

● Requires (7.2.3)

● Fold-expression syntax (7.4.1)

● Forwarding (7.4.2)

Xxxx

Points of Surprise:

Chapter 1:

1. On page 22 of 1.4.2 Initialization, I was surprised by the fact that “{}” (curly brace) is an

alternative to “=” to initialize a value of an object. I was also quite impressed by the fact

that “{}” is superior over “=” intems of object value initialization as it prevents

narrowing conversion.

2. On page 24 of 1.5 Scope and Lifetime, I was surprised and impressed by the way C++

controls life cycle of objects by using destructor to deallocate the memory.

3. On page 27,28,29 of Pointers, Arrays, and References, I was surprised by co-existence of

references and pointers in C++. I was surprised by their difference and the advantage of

pointers over references.

4. On page of 29 30 of 1.7.1 The Null Pointer, I was impressed by how nullpointer works in

C++.

5. On page of 33,34 of 1.9.1 Assignment, I was surprised by how assignment works

differently on pointer and reference.

Chapter 2:

1. On page of 41 and 42 of 2.3 Classes, I was curiously about how different or similar public

class and private class in C++ compared to public class and private class in java.

2. On page of of 43 and 44 of 2.4 Unions

Chapter 3:

1. On page of 49 and 50 of 3.2 Separate Compilation, I was surprised by the way C++

support separate compilation by organizing source code into .h and .cpp files.

2. On page of 53 and 54 of 3.4 Namespaces, I was surprised by how C++ uses namespace

to organize and simplify large program.

3. On page of 55 and 56 of 3.5.1 Exceptions, I was curious about how try and catch

statement is different from try and except statement in python.

4. On page of 66 and 67 of 3.6.3 Structured Binding, I would like to know more about how

structured binding can be used in C++.

Chapter 4

1. On page of 79,80,81 of 4.3 Abstract Type, I was surprised by the use of abstract type in

C++.

2. On page of 82 and 83 of 4.4 Virtual Functions, ZI was surprised by the use of virtual

function in C++.

3. ON page of 83,84,85 of 4.5 Class Hierarchies, I was surprised by how a set of classes are

ordered in hierarchical relationships

Chapter 5

1. On page of 108 of 5.4.5 swap(), I was surprised by use of swap()

Chapter 6

1. On page of 119,120,121,122 of 6.3.3 Lambda Expressions, I was surprised by use of Lambda expression in C++.
2. On of 123,124,125 of 6.4 Template Mechanisms, I was surprised by use of template mechanism in C++.

Chapter 7

1. On page of 138 and 139 of 7.4.1 Fold Expressions, I was surprised by use of fold expression in C++.
2. On page o 139 and 140 of 7.4.2 Forwarding Arguments, I was surprised by use of forwarding argument in C++.

Xxxx

**Chapter 1**

1. Page 2-> Nonzero value, indicating failure and returned by main(), are

handled differently in different operating systems like Windows and Linux.

2. Page 5-> In function overloading, if two possible functions can be called

but neither is better than the other then the call is interpreted to be ambiguous.

3. Page 8 -> The {} form can be preferred over = as it saves from

conversions that can cause loss of information.

4. Page 10 -> concept of constexpr; value calculated by the compiler unlike

const which is evaluated at runtime

5. Page 12-> A reference can only be initialized once. We cannot have

uninitialized reference. Also, there is no null reference.

6. Page 15-> variable can be defined within ‘if’ statement to keep the scope

limited and errors minimized.

**Chapter 2**

1. Page 22-> Objects created using ‘new’ operator are allocated memory

from a place called the free store and lives until they are deleted.

2. Page 25-> Similarity of struct to class with only difference being members

are public by default.

3. Page 25-> concept of Union -> ‘’a **struct** in which all members are

allocated at the same address so that the **union** occu- pies only as much space as its

largest member. Variants can be used to avoid handling typed unions.

4. Page 27 -> Enumerators are strongly typed and they are scoped by their

enum class to avoid confusion between similar values from different enum classes.

**Chapter 3**

1. Modules (Section 3.3, page 32-33)-> I find it interesting how the makers

of C++ have incorporated the use of both old and new styles of making the code

modular to save time and cost of updating existing codebases in C++. The traditional

‘#include header.h’ is plagued with the problems of multiple processing of any header

file if its #include-d multiple times. Moreover, the order of inclusion of the header files

also matter-those that are included before can affect the code in those included

afterwards. ‘Import module’ is something that I have not come across before but

having a software engineering background, I can very well comprehend the genius

behind it in achieving object oriented programming. With the use of import module,

modules are compiled only once and modules can be imported in any order, achieving

certain degree of independence from each other from a programmer’s perspective.

2. Error Handling (Section 3.5) -> how simple techniques of abstraction can

be used to separate where error is detected and where it is handled, clearly illustrated

on page 36 where it specifies how ‘throw’ witnesses the error first and then calls the

handler to take appropriate actions to tackle the error thereafter. I also noticed that

while ‘noexcept’ can be used with functions that should never throw an exception, care

must be taken to avoid using it with functions which might inevitably throw one.

Invariants also seemed as a pretty smart technique in handling errors. Conditions

which will not change throughout the execution of the program should be checked in

the constructor itself. Further on page 39, it was interesting to note the various places

in which an error must be returned or an exception be thrown or termination occurs.

3. Static assertions (page 41) to simplify error handling by asserting

constants failure check at compile time.

4. Structured Binding (page 45) for returning multiple values from a function

efficiently.

**Chapter 4**

1. Concrete Types (Section 4.2, page 48) -> Use cases of concrete classeswhere

to use and where not to. They should be used for types that do not change often

and where local variables provide most of the clarity.

2. Page 50 -> A const member function can be invoked for both const and

non-const objects but a non const member function can only for invoked for non-const

objects.

3. Page 53 -> Initializer list object type for vector initialization.

4. Page 55 -> Explicit use of keyword ‘override’ to allow compiler to catch

mistakes like those made in the function names is a clever error avoiding strategy.

5. Page 56-> Remarkable difference between concrete type and abstract

type classes which can help make clever usage choices. Abstract type can be used for

types that can change often as it will not need to be compiled.

6. Page 57 -> Virtual Function Table (how call to virtual functions are

resolved).

7. Dynamic Cast (page 61) -> used for hierarchy navigation when a member

function is only provided by a particular derived class.

8. Unique pointers (page 62) - compiler implicitly causes destruction of the

unique\_ptrs

**Chapter 5**

1. Page 68-> Making use of explicit to avoid implicit conversion in

constructors taking a single argument was new information and seemed quite useful.

2. Page 69 -> It was pretty exciting to learn that even copying of a simple

entity like object can be carried out differently and with varying efficiencies and how a

programmer should keep it in mind while making a class.

3. Page 71 (Moving Containers)-> The use of “rvalue reference &&” Is

something that I have not seen before but it demonstrates a very effective way of

returning reference from a function.

\_

Mansi Khemka (mk4282)

4. Page 76 -> I did not know that C++ allows its users to create their own

literals. This seems pretty amazing.

**Chapter 6**

1. Page 81 -> Came across the term ’specialization or instantiation’ coined

for a template plus a set of template for the first time. These instantiations are later

generated code for by the compiler and type checked.

2. Page 83 -> We can make use of ’s’ letter to explicitly denote a string.

3. Page 84 -> The concept of deduction guide is something I came across

for the first time.

4. Page 85 (Functors or Function objects) -> enabling us to call objects like

functions.

5. Page 87 -> ‘Capture List’ in lambda functions was a new concept for me.

**Chapter 7**

1. Page 94-> Use of ‘concepts’ to constraint template arguments; a

predicate specifying how one or more types can be used.

2. Page 96-> Template overloading and how compiler choses from several

alternative functions for an argument.

3. Page 98 -> The idea of generic programming; abstracting from concrete

efficient algorithms to obtain generic algorithms.

4. Page 100-> The concept of lifting->generalizing from a piece of code

while preserving performance.

5. Page 100 (Variadic Template) -> The concept of variadic template was

also pretty new to me (template accepting an arbitrary number and types of

arguments).

Xxxx

* • Page 2, Section 1.2.1 – I was a little surprised by how the integer value returned by main() is the program’s return value to ‘‘the system.’’ And how if no value is returned, the system will receive a value indicating successful completion and non-zero value indicates a failure. I was also a little surprised by how this is taken into account on Linux machines only and why Windows machines rarely. If that’s the case, I was wondering what effects/advantages does this have in Linux machines as opposed to Windows machines?
* • Page 6, Section 1.4 – I found it very interesting how variable type sizes are implementation-defined and how they may depend on the compiler, machine, compiler options etc. It is really nice how we can use a single quote as a digit separator to make long literals readable for users instead of explicitly using string format functions like in other languages. This was pretty new to me and found it really interesting.
* • Page 8, Section 1.4.2 – One of the few things that I did not get to use much until now is the use of the general {}-list form and the fact that it saves us from conversions that lose information. I did not know until now how conversions that lose information, narrowing conversions, such as double to int and int to char, are allowed and implicitly applied when you use = (but not when you use {}).
* • Page 15, section 1.8 – One of the other things I have never used before was introducing variables in if-statements and testing them. And with respect to scopes, it is interesting how the scope of the introduced variable is on both branches of the if-statement.
* • Page 16, Section 1.9 – Although I learnt coding in C++ a few years ago, I never really realized the real and primary benefits of C++. The most important of them all being how well-suited it is for low-level and systems programming because of the fact that it operates very close to the hardware and hence giving the programs a huge performance boost. For example, how a C++ implementation of addition sees a machine’s memory as a sequence of memory locations into which it can place (typed) objects and address them using pointers.
* • Page 23, Section 2.2 – Element access in vectors became very clear to me after going through the basic idea of how the [] operator in overloaded for subscripting. I have used STL for a while now and never got to the depths of operator overloading and how it is used for subscripting while defining user-defined data-types. I also got a clear understanding of how the constructor initializes the Vector members using a member initializer list.
* • Page 26, Section 2.4 – Although I did use Unions before, I did not know before how we can use variants to eliminate most direct uses of unions and how maintaining the correspondence between a type field and the type held in a union is error prone. These programming constructs for better programming/performance are new to me and it is very interesting to know the underlying basics of different aspects of the language.
* • Page 32, Section 3.2 – I found it surprising how when two .cpp files include a header file consisting of an interface, a function call can be made from one .cpp file while the function definition is contained in the other .ccp file and how achieve modularity from doing so.
* • Page 32, Section 3.3 – The concept of Modules to achieve modularity for better compile-time performance and how they can be used to replace #include statements is completely new to me.
* • Page 37, Section 3.5.2 – I have never worked with invariants and the concept of invariants/class invariants is completely new to me. I did understand now how statement of what is assumed to be true for a class are called class invariants or simply invariants and what role the constructors and the member variables play in this context.
* • Page 40, Section 3.5.4 - Contracts and Static Assertions, Assert in debug mode and non-debug mode- Having used C++ for a while now, I never came across contracts and assertions before and was really surprised by how they can be used in particular scenarios.
* • Page 45, Section 3.6.3 – This is the first time I am coming across structured binding and how local names can be given to members of a class object by this mechanism. I have not seen this syntax where we can ‘‘unpack’’ a class’s members into local variables.
* • Page 52, Section 4.2.2 – Although I did know the basics of constructors and destructors, I never knew that the formal name for the technique of acquiring resources in a constructor and releasing them in a destructor was known as Resource Acquisition Is Initialization or RAII and its importance in eliminating ‘‘naked new operations,’’ to avoid allocations in general code and keep them buried inside the implementation of abstractions.
* • Page 53, Section 4.2.3 – I have never used the concepts of static\_cast, dynamic\_cast, reinterpret\_cast and const\_cast before and understood these concepts as I read the textbook and also realized that explicit/ undeclared casts are best avoided.
* • Page 57, Section 4.4 – The whole idea behind virtual function tables (table of pointers to functions) and how a base class object contains information to allow it to select the right
* derived class function to call at run time is new to me and I did not know the implementation details of this kind of polymorphism. I also did not know the idea behind the concept of using dynamic\_casts in class hierarchy to check if the object pointed to by the argument of dynamic\_cast is of the expected type or a class derived from the expected type.
* • Page 67, Section 5.1.1 – This is the first time I am coming across the usage of =delete to indicate that an operation is not to be generated, how it =delete makes an attempted use of the deleted function a compile-time error and how it can be used to suppress any function, not just essential member functions.
* • Page 71, Section 5.2 – I have never used and come across the copy assignment, move and move assignment functions in a class. Back when I learnt C++, I had only come across copy constructors and destructors. This is a new learning for me too.
* • Page 75 – Section 5.4.4 – This is the first time I am coming across the concepts of user-defined literals through which we can achieve flexibility and efficiency of built-in type initialization.
* • Page 85-87 – Section 6.3 – Although I did work with lambda functions in Java and Python, I have never come across them in C++. Also, template concepts such as function templates, function objects are pretty new to me at this point. I never imagined that C++ would have all the features that very high-level languages have today and I was surprised after I discovered some of these features.
* • Page 91, Section 6.4.3 – Having worked with multiple languages before, I have never come across anything like a Compile-Time if before and how we can discard branches of an if statement at compile-time based on a constant expression condition.
* • Page 94, Section 7.2 – Being a fairly new feature, I have never come across “Concepts” in C++ before where a template function requires that its first template argument is some kind of sequence and its second template argument is some kind of number and how they can be used in generic programming.
* • Page 99 – 100, Section 7.3.2 and 7.4 - The idea behind abstraction using templates and Variadic templates and their importance was very well demonstrated in the book and although these concepts are new to me, I was amazed by the power of generic programming and could very well relate to how C++ being highly type dependent can function like many other high level languages (or even better in certain scenarios).

Xxxx

Chapter 1

- I didn't know that floating point numbers could be expressed using exponent notation:

3e-2 instead of 0.03

- 0 prefix indicates octal — somehow, using a leading zero by itself seems like it would

cause problems

- I wasn't aware that a single quote could be used as digit separator

- Unary plus

- Turns an lvalue into an rvalue

- Forces a type conversion (promotes to an int from smaller types)

- Can be used as a signed int cast

- "initializer list notation"

- Didn’t know it could be used on basic types (not just struct types)

- What types of conversion does it do?

- "The = form is traditional and dates back to C, but if in doubt, use the

general {}-list form. If nothing else, it saves you from conversions that lose

information:" I don’t quite understand this

- "With auto, we tend to use the = because there is no potentially

troublesome type conversion involved, but if you prefer to use {}

initialization consistently, you can do that instead." Same thing, I feel like

I'm missing something.

- constexpr

- keyword used like const , but it means "to be evaluated at compile time"

- can be used on variables

- can also be used on functions

- must be pure (must have no side effects, can only use information passed

through arguments)

- required by array bounds, case labels, template value arguments

- required by constants declared using constexpr

- I wasn't aware that range-for can iterate directly over a curly-bracket list.

- One can use auto& x : v (reference) instead of auto x : v (copy value) to avoid copying

large items. I have definitely seen this before in production code, but had never

connected this specific pattern to avoiding copies.

- nullptr is the recommended value to assign to and check for null pointers, not NULL

- There is no such thing as a "null reference", and it's very difficult and not recommended

to actually create that situation.

- if (auto n = v.size(); n!=0)

- the return value inside the parentheses is the return value of n!=0

- like Scala functions?

- Advice: "[27] use unsigned for bit manipulation only."

Chapter 2

- Vector(int s) :elem{new double[s]}, sz{s} { } // construct a

Vector

- struct notation for constructors

- called a **member initializer list**

- "initialize the member elem to point to a new double array on the heap, and

initialize the member sz to be the value s"

- structs can still have constructors and other member functions. Their only difference with

classes is that all members are public by default. This makes sense from the C

perspective, where everything is public.

- For union types, a variable outside the union can hold an enum value that indicates

which of the elements of the union is in use

- This is called a tagged union.

- Is there any built-in language feature that does this automatically? It seems

troublesome to have to define an enum with each potential type every time.

- I'm not sure I really understand what variants are yet.

- enum MyValues means that I can refer to the different values normally within the

current namespace. enum class MyValues means I must refer to the values by

prefixing them with their enum's scope, eg MyValues::A .

- An integer can be used as a sort of index when initializing an enum to indicate which of

its values to use.

- Enums can have their own operations defined. This is really interesting! I remember

thinking enums were inconvenient in Java exactly because they didn't have defined

operations.

Chapter 3

- translation unit = a cpp file that is compiled by itself, including .h and #includes

- Mixing old and new includes with both headers and modules seems like it would get

messy, but modules are definitely a more elegant way of organizing code.

- "It's legal to define another int main() in a different namespace. The real main() isn't

assigned to any namespace." This seems a little confusing and potentially dangerous,

but it's interesting to know that there is no default namespace.

- using statements can be used as normal statements within a function -- just like in

Python.

- Namespaces are basically large named scopes: they are useful in large projects to

group related code and classes.

- noexcept denotes functions that can't throw anything

- if they try to throw something, it turns into a std::terminate() call

- What is re-throwing? Does it just mean passing the function to a higher-level caller in the

stack?

- static\_assert can be used with constexpr to generate compiler errors.

- Default function arguments are often just overloading but syntactically simpler.

- Return types can be inferred with the auto keyword, just like variable types.

- This seems a little obvious now that I know it exists, but I was still surprised.

- Does not provide a "stable interface" -- does this mean the compiler can't check

types as rigorously as it normally would?

- Structured binding "unpacks" values from a struct into variables

- Usually uses the auto keyword in front of the brackets

- for (const auto [k, v] : myMap) { … }

- Structs are used like tuples, which seems convenient.

- Structured bindings can retrieve private data values accessed through member functions

- How does this work?

Chapter 4

- Syntax is fixed by language, so unary operators must be unary, binary operators must be

binary, etc.

- One can't redefine operators for built-in types, only user-defined types

- This makes sense, because it could modify how standard library functions work,

and that would be very messy.

- "handle-to-data" model: hold a pointer to allocated memory that actually has data. Allows

for resizable containers and variably-sized data. In this case, the list can have different

lengths.

- CS3157's linked list contains void\* data. Is that an example of handle-to-data?

Not really, because the size of the object will likely not change. The list can be

used for different data types, but a node will most likely always point to the same

type.

- Being able to use static\_cast, reinterpret\_cast and const\_cast is convenient because

they show the programmer's intention. Instead of writing out the type and omitting const,

or writing out a numerical type that seems almost identical to the original type, the

change is made very clear.

- The virtual keyword is similar to the abstract keyword in Java.

- I had not considered the extra space and time overhead that virtual functions require.

- Using dynamic\_cast seems a bit tedious, but it makes sense that it would be used if type

information has somehow been lost by passing the object into a method for a base class

interface.

- "[18] A class with a virtual function should have a virtual destructor;" I'm not sure I

understand why abstract classes can have virtual destructors but not virtual constructors,

or even why they need either.

Chapter 5

- I wasn't aware the default or delete keyword could be used on essential operations.

- Move constructors/assignments are completely new to me.

- && as "rvalue reference": I'm not sure I understand why this is necessary. Why

wouldn't a regular reference suffice here?

- std::move(x) is really convenient -- it seems that even if a method takes a

parameter by value instead of reference, one can still avoid copying by passing in

the std::move() of the argument.

- User-defined literals

- need a literal operator to perform conversion (makes sense that this would be

constexpr), since literals are present at compile time.

- operator""{x} where {x} is a sequence of letters, defines the literal suffix

- std:swap() is really just three lines; it makes sense to define move operations whenever

possible so that all the stdlib containers and algorithms can be used on the items.

Chapter 6

- concepts define constrained arguments, which make up constrained templates.

- deducing types with make\_ functions and deduction guides

- The arrow syntax is a little confusing because it has so many other meanings

(dereference and get member value in C++, but also lambda in other languages).

- Once concepts are introduced in C++20, will deductions still be necessary in non-legacy

code?

- Even the "function call" operator can be user-defined! Learning this made me realize that

this is what makes first-class functions possible.

- Lambda expressions

- They are implicitly generated function objects. I assume this means their types

are inferred as well.

- I'm not quite sure I understand what goes inside the capture list brackets [ ] at the

beginning of a lambda. I assume that when the lambda is used, it is basically a

function call, and parameters need to be passed in just like with a regular

function. However, isn't this the purpose of the argument list that follows the

capture list?

- If auto is a parameter type, the lambda is called "generic". Like a template, it

can be used on multiple types.

- I'm not sure I understand what if\_pod<> does in the if constexpr example.

Chapter 7

- I didn't know variadic templates were possible!

- Fold expressions seem limited because one can't specify a range or an interval.

- Forwarding sounds very powerful, and I'm interested in learning more about how it

works.

Xxxx

A list of things that surprised me:

1. I didn't know that it is possible to initialize (page 8, chapter 1.4.2) with curly braces

**type var {values}**

2. I did not know that **constexpr** (page 10, chapter 1.6) exists. I used const from time to

time, but never constexpr.

3. I did not know that nullptr exists (page 13, Chapter 1.7.1), I was taught C, and I was

taught to use **NULL** .

4. Unions and variants are new to me (chapter 2.5, page 26)

5. Modules (page 32. Chapter 3.4)

6. Invariants in conjunction with exceptions (pages 35-40, chapter of Error Handling 3.5)

7. Use of a properly defined operators as a way to avoid copying is interesting. (page 44)

8. Page 49, compatibility with Fortran for complex() class

9. I was surprised that some operators are not defined within the class, page 50.

10. The idea of using abstract types (chapter 4.3, page 54), specifically the usage of

Container in functions.

11. The destruction process of derived objects, page 61

12. Didn't know that unique\_ptr exists, page 62

13. =delete on page 67

14. Copy and Move chapter, especially the part on containers. Pages 70-72.

15. I used iterators before, but did not know the equivalence between v.begin/end vs auto&

x : c in for loops. Page 75

16. Concept of literals, page 75-76

17. Notion of constraints for Templated classes, page 81-82

18. Argument deduction, page 83

19. Everything in chapters 6.3, 6.4 and the whole chapter 7 is new and a surprise for me.

Xxxx

**Chapter 1**

• Why does C++ permit declarations separate from definitions? While this may benefit compilers, it reduces the readability of code (Pg 4)

 • I never knew a digit separator existed! (page 6)

• Why does a unary plus exist (page 6)

• Why is it unfortunate that order of evaluation of function arguments is unspecified (page 7)

• Why are’nt variable sized arrays permitted on the stack like they are in C? (page 11)

• It is possible to introduce a variable in an if statement! I guess this allows you to limit scope of data only used for conditionals (page 15)

 • Why should unsigned only be used for bit manipulation? (page 19)

**Chapter 3**

• My problems with separation of declaration and definition (page 4) are confirmed! (page 32) Modules seem very exciting and modernizing for C++.

• I did not know you could use the using keyword on a specific name.

• I did not know that noexcept keyword existed (page 37)

• I see try blocks often. Why are try blocks considered a sign of bad code? (page 38)

• Is it superior that an exception facility of a language is not used to provide an alternate mechanism of returning values? (page 39)

• I want to learn more about contracts in c++20! (page 40)

• I did not know C++ had default function arguments (page 42)

• I like the idea of “20th century style” (page 44)

• Is structured binding python style tuple unpacking in C++? (page 45)

**Chapter 4**

• I did not know casts got their name from the fact that there use is somewhat broken (page 53)

• I did not know about dynamic casting (page 61)

**Chapter 5**

• Compilers elide copies when constructing objects (page 66)

• Implicit conversions can be defined by singe argument constructors (page 67)

• I did not know cpp had an option for garbage collection (page 73)

• I did not comparisons had preferred semantics in relation to assignment (page 74)

• I did not know you can make user defined literals (page 75)

• If the default is to declare explicit, why not have the default be explicit and have a keyword nonexplicit (page 77)

**Chapter 6**

• why did we switch from class T to typename T (page 80)

• I did not know c++20 was introducing constrained templates (page 82)

• I did not know template arguments could be deduced (page 83)

• deduction guides seem tricky! (page 84)

• not sure i really understand what is going on with generic lambdas and use of auto (page 88)

• ive never seen “if constexpr” (page 91)

**Chapter 7**

• Why do concepts get such a special name, rather than just interfaces or even remain a nameless part of the templates ? (page 94)

• Concepts are hard to read! (page 95)

Xxxx

1. page 6 - minor- integer literals can be expressed as binary, hex or oct, didn’t know that these could be used in addition to decimals
2. page 32 - chapter 3 modules is all new concept that I wasn’t familiar with
3. page 34 - namespaces seem similar to packages in java
4. page 40 - assert for debug mode only
5. page 44 - structured binding seems similar to destructuring in javascript ES6
6. page 62 - unique/shared ptr being automatically managed by the system and cleared when out of scope seems super useful and something new for me
7. page 67 - use of delete keyword to disable certain constructors and functions. Wondering if copy and move constructors are deleted then how does assignment work then since it relies on those two.
8. page 69 - class having a destructor is a strong hint that default member wise copy semantics is wrong as the copy then shares some memory with original object
9. page 70 - The notation used by copy constructor to init member variables outside of the constructor body(following : after signature) is something new to me.
10. page 84 - interesting how deduction guides are used to fill in for the lack of concepts in std library for C++17, looking forward to reading/learning more about that.
11. Page 85 - Function objects are new to me and seem quite useful and clear when used as predicates.
12. page 91 - “compile time if” - need more examples/explanation to understand this
13. page 95 - concepts seem like advanced version of java interfaces at first, but after reading chapter 7 fully, not so much. Instead it’s a lot more powerful and allows for much more features than interfaces which are just abstract classes with with abstract only functions.
14. page 100 - variadic templates - surprising that what print can do, which is to take in any number of any type of arguments is available for anyone to use as language feature.
15. page 101 - is sizeof...(tail) correct or it should be sizeof(tail...)
16. page 101 - also need better understanding of compile time if referenced from section 6.4.3
17. page 103 - Fold expressions and their use to convert to vector
18. page 103 - Forwarding arguments in general is new and interesting and a bit confusing.
19. Chapter 6 & 7 - overall most of template, concepts etc for generic programming is new to me.

Xxxx

Ch1 :

1. **Section 1.4.2 , Page 8** : I have never used the full spectrum of methods to initialize variables and it was a good refresher to see them all in one place for reference

2. **Section 1.6, Page 9-10** : The correct usage of constexpr and const as well as the different scenarios where they can be used.

3. **Section 1.7.1, Page 13** : The characters in a string literal are immutable, so to handle count\_x("Hello!"), I declared count\_x() a const char∗ argument. 4. Use nullptr rather than 0 or NULL. I like the examples used to illustrate this.

Ch2 :

1. **Section 2.4, Page 26** : The concept of variants was new to me and I feel the example was simple enough to help me understand this new concept.

2. **Section 2.5, page 26** : Even after having worked with enumerations since my school days, I had never paid attention to the minute details talked about in the text. It helped me learn how to correctly use enums.

Ch3 :

1. **Section 3.2, Page 31** : I love how the example on Vector is structured. It explains even the simplest of details on how a program should be modularized and organized.

2. **Section 3.2, Page 32** : I was unaware of certain technical terms like translation unit, and it feels nice to come up to speed in this respect as well.

3. **Section 3.3, Page 32** : The entire section on modules was new to me. The example was written in a way that was so easy to understand. And the differences between headers and modules were explicitly stated which helped me understand the enormous impact that the use of modules can have on my code.

4. **Section 3.5.1, Page 36** : Statements like “I caught the exception by reference to avoid copying and used the what() function to print the error message put into it at the throw-point” helped me understand why things are being done the way they are being done. I really appreciate the explanation of the small details that often go unnoticed.

5. **Section 3.5.2** : Invariants; It was a new concept for me.

6. **Section 3.5.3 :** I absolutely loved this section because not only did suggest ways to handle errors but also helped me understand why some methods are preferred over other methods in certain scenarios. Though their application might still be tough, but this section gave me a clearer picture in my head.

7. **Section 3.6.3** : The example in this section was somewhat surprising for me. I did not know “return {s, i};” was a valid statement in c++ as well.

Ch4 :

1. **Section 4.2.1 Page 50** : Explanation on when and why the keyword const is used was really helpful.

2. **Section 4.2.3** : Explanation of static\_cast. This was a new concept for me. As were reinterpret\_cast and const\_cast.

3. **Section 4.3** : The curious =0 syntax says the function is *pure virtual . I had never come across this syntax.* The use of override is optional, but being explicit allows the compiler to catch mistakes, such as misspellings of function names or slight differences between the type of a virtual function and its intended overrider. I felt this piece of advice was specially helpful.

4. **Section 4.4** : The basics of how virtual functions are organized was new to me and helped me visualize the entire concept better.

5. **Section 4.5.2** : I found hierarchy navigation new and interesting.

6. **Section 4.5.3** : The use of unique\_ptr to replace raw pointers In order to avoid forgetting to delete objects created using new. I had not thought about unique pointers this way before.

Ch5 :

1. **Section 5.1.1**, **Page 67** : A =delete makes an attempted use of the deleted function a compile-time error; =delete can be used to suppress any function, not just essential member functions.

2. **Section 5.2.2** : The correct and efficient method to move containers.

3. **Section 5.4.4, Page 76** : I was unaware that it is possible to create user-defined literals. The fact that it can be done so easily was somewhat surprising to me.

Ch6 :

1. **Section 6.2 Page 81** : Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to hand-crafted code. I had just never thought about it this way.

2. **Section 6.2.1** : Constrained template arguments (new concept)

3. **Section 6.2.3** : When and how to use deduction guides

4. **Section 6.3.2, Page 86** : How to use function objects correctly. Explanation of the example on Less\_than{x}

5. **Section 6.3.3, Page 87** : The use of unique pointers in lambda expressions

6. **Section 6.3.4, Page 88** : When and how to create and use generic lambdas

7. **Section 6.4.3** : Compile-Time if was an overall new concept for me.

Ch7 :

1. **Section 7.2** : Concepts (new topic)

2. **Section 7.4** : How and when(or more importantly when not) to use variadic templates

3. **Section 7.4.1** : Left and right folds

4. **Section 7.5** : Explanation of the template compilation model(including duck typing)

Xxxx

Ch 1

* Parameters in function declarations don’t need names
* Print function takes many different argument types (not just string, but also int, float, etc)
* Single quote ‘ can be used to separate digits for readability
* Unary plus + exists even though it’s not necessary
* Initialization with braces instead of equal sign is preferred because it prevents type narrowing (e.g. double d = {2.3})
* Can specify the compute precision of numbers (complex<double> z = 1)
* *auto* can infer type of the object it is assigned to (When does auto fail to detect the type of an object? Does it throw a compile time or runtime error?)
* *const* vs *constexpr* differs in compile time versus runtime calculation. Although constexpr can be back cast to const and constexpr can be called with non-constexpr arguments. (Can you go into more examples as to why one would prefer const over constexpr or vice versa?)
* copy by reference in for loop is more efficient
* no 0 or Null, instead now nullptr is used to denote a null value (how is nullptr represented in memory?)
* can assign variable and test for Boolean expressions at same time within an if statement (e.g. if (auto n = v.size(); n != 0))

Ch 2

* iteratively building up our understanding of the Vector class by adding more functionalities in each chapter is interesting
* member initializer syntax is new to me (e.g. :elem{new double[s]}, sz {s}). Why use a member initializer when you can initialize the members within a constructor?
* Union object – where all members of a class are all the same address so the object only stores the largest member (one member stored at each time) – seems useful if a class member is allowed to be different types
* Enum initialization can also use same bracket initialization (Color y {6})
* Can you change enum underlying type to not be int?
* Must make enum class explicit to minimize confusion (Light::red vs. Color::red)

Ch 3

* Implementation of .h files can be independently compiled from user .cpp files (separate compilation)
* How is a module imported only once if the import statement is stated in every translation unit? The compiler checks that it’s already imported and stops?
* Why would I prefer a namespace over a class if I can also use a class to specify functions with same name but different purpose? (e.g. if I have two classes A and B I can create function A.foo() and B.foo() that do different things)
* Can optimize copies to moves
* Don’t return large objects by returning a pointer
* Can return a struct object with syntax like return {a, b}
* C++ allows for unpacking variables

Xxxx

 **CHAPTER 1:**

* • **1.2:** >> operator puts the second argument into the first. I thought it was simple the convention used for cout
* • **1.3:** when two alternative functions can be called an error: ambiguous gets triggered
	+ • 1.4: o Sizes of different types are multiples of the size of char in the machine
	+ o Initialization can be made using braces and type conversions will not happen when using them. Ex: double d {3.2}; int I {7.5}; //error. Remember that ints truncate doubles.
	+ • 1.6: o Const vs. constexpr: const get calculated at runtime and constexp get calculated at compile time and put in read only memory
	+ o A constexpr can be used for non-constant arguments but when that is done it returns not a constexpr
	+ o A function that returns a constexpr can only modify local variables
	+ • 1.7: o range-for-statement. Ex: for (auto x : v) {} //Note that this places **a copy of** the item from v in x. Instead you can use: for (auto**&** x : v) {} which creates a reference of the element in v and assigns it to x.
	+ o A reference is like a pointer, but you don’t need to use \* to access its value. They are useful in function parameters because they allow us to make sure we are modifying the structure and not a copy of it ★ This can also be used to reduce the cost of copying
	+ o Use **nullptr** instead of NULL when referring to a null pointer to eliminate confusion
	+ o >> operator: get from
	+ • 1.9: o you can declare a variable inside an if statement

**CHAPTER 2:**

* • **2.4: unions:** a struct in which all members are allocated to the same address, a memory space that is of the size of the largest element and can only hold the value of one item at a time. A **variant** is like a union but safer to use because it is tagged.
* • **2.5:** enum class cannot be assigned ints, but can be initialized to an int using {}, and we can define operators for them. Plain enum can be assigned to ints.

**CHAPTER 3:**

* • **3.2: a translation unit** is a .cpp file that gets compiled by itself and is part of a program
	1. • **3.3: modules:** o Module;
	2. o Export module vector;
	3. o Export class vector{ // define classs }
	4. o Define class functions
	5. o Export non-member functions

Shayel Encaoua Design Using C++ 01/29/2020

* + o In file: import vector
	+ o A module is compiled once only, and its meaning is not affected by importing other modules
	+ o Import is not transitive
	+ o “using” can be specified inside a specific scope
	+ o **Noexcept:** a function that should never throw an exception. Ex: void a() noexcept{} ▪ **Alternatively:** do throw – terminate
	+ o **Throw x{“**write text to describe the exception**”}**
	+ o **Error handling alternatives:** ▪ **Error code:** a failure is normal and expected and can be handled by the caller
	+ ▪ **Terminate:** we cannot recover from the error, the system requires restarting upon that error
	+ o **Static\_assert:** like assert but with a second argument that is the error message to be sent
	+ o **Structured binding:** to pack: return {a,b}; to unpack: auto [x,y] = f();
* • **3.4: namespaces:** can be used to establish differences and relationships between different types of codes. I can declare my own: namespace My\_code {}
* • 3.5:
* • 3.6:

**CHAPTER 4:**

* + • 4.2: o **inline** functions are those that do not call other functions
	+ o **Initializer list**
	+ o Avoid explicit casts.
	+ o Const\_cast casts away consts
	+ o The word **virtual** means ‘‘may be redefined later in a class derived from this one.’’\
	+ o **=0** implies the function is **pure virtual**: some class derived from the class that contains it must define the function.
	+ o A class with a pure virtual function is an **abstract class**, and it can nly serve as the interface to a class that implements its virtual functions.
	+ o A class that provides an interface to many other classes is called a **polymorphic type**
	+ o Abstract classes do not have a constructor ▪ Objects are constructed bottom up – base first, then derived
	+ o Abstract classes have virtual destructor ▪ The destructor is also overridden, even though it does not have the same name

Vector::Vector(std::initializ er\_list lst) // initialize with a list

:elem{new double[lst.siz e()]}, sz{static\_cast(lst.siz e())}

{

copy(lst.begin(),lst.end(),elem); // copy from lst into elem (§12.6)

}

Use and check the output of static\_cast to explicitly convert the size of the initializer list to an int

* • **4.3-4.5:**

Shayel Encaoua Design Using C++ 01/29/2020

* + ▪ Destructors implicitly call on the destructors of its members
	+ ▪ Objects are destructed top down – derived first
	+ o Class implementation\_of\_x : public x {} ▪ Public = derived from
	+ o **Dynamic cast:** dynamic\_cast<Type\*>(object) – evaluates object at runtime, if it is not of type Type or derived from it, returns nullptr ▪ Use pointers if the derived classes are also accepted, if not use reference (&)
	+ o **Unique\_ptr:** unique\_ptr<Type> ▪ Prevents us from having leaks because of not having a destructor for a pointer, the compiler will generate it and call it when the object goes out of scope

**CHAPTER 5**

* + • 5.1: o =delete: suppress member functions and avoid the utilization of defaults
	+ o “explicit” when written before the definition of the constructor prevents implicit construction from happening
	+ o Recall: ▪ X(Sometype); // ‘‘ordinary constructor’’: create an object
	+ ▪ X(); // default constructor
	+ ▪ X(const X&); // copy constructor
	+ ▪ X(X&&); // move constructor
	+ ▪ X& operator=(const X&); // copy assignment: clean up target and copy
	+ ▪ X& operator=(X&&); // move assignment: clean up target and move
	+ ▪ ˜X(); // destructor: clean up
	+ • 5.2: o Move constructor: ▪ && means right value reference – a reference so something nobody else can assign to

Vector::Vector(Vector&& a)

:elem{a.elem}, // "grab the elements" from a

sz{a.sz}

{

a.elem = nullptr; // now a has no elements

a.sz = 0;

}

**CHAPTER 6:**

* + • 6.2: o Useful expressions: ▪ return x.size() ? &x[0] : nullptr; // pointer to first element
	+ ▪ return x.size() ? &x[0]+x.size() : nullptr; // pointer to one-past-last element
	+ o **template<Element T>:** used to create constrained templates
	+ o valuetype:

template struct Buffer { Shayel Encaoua Design Using C++ 01/29/2020

* 1. o Use s suffix after a string literal to make it a string and not a char\*
	2. o **6.2.3 – all new**
	3. o The function called operator() implements a function call
	4. o A **predicate** is something we can invoke to get a true or false value
	5. o **6.3.3 – all new**

using value\_type = T;

constexpr int size() { return N; }

T[N];

// ...

};

* • 6.3:

**CHAPTER 7**

* • **7.2 – all new material**
	1. • **7.4 – all new material** o **…** : a parameter declared with a … is called a parameter pack. <typename… x> indicates that x is a sequence of types. And x… y indicates that y is a sequence of values of the type x