**Start:** Nov-16-2018

Tutorials: https://www.learncpp.com/cpp-tutorial/introduction-to-these-tutorials/

**VS Code TIPS**

1. VS Code setup: <https://code.visualstudio.com/docs/languages/cpp>
2. Use tasks.json to set command and action.

Cmd+p => command palate

Cmd+p => task <name of task to run>

**Tip**

1. Learn how to program

2. Lean how not to program

3. Mainly learn to debug.

Practice. Practice. Practice.

**Best Practices**

1. Problem solving approach
   1. Define vision. (vague)
   2. Drill down and define requirements (specifics)
   3. Start simple. 1 area at a time. Add features incrementally
   4. TDD have proved very useful for mission critical software
2. Usually 20% on dev and 80% on other activites.
   1. Document code
   2. Simplified code
   3. Modular and organized
   4. Robust with edge cases
3. Treat warnings as errors. In g++ -Werror
4. Comments
   1. At the library, program, or function level, describe *what*
   2. Inside the library, program, or function, describe *how*
   3. At the statement level, describe *why*.
5. Variables, Functions
   1. Uninitialized is a common trap
   2. Use namespace. Avoid global naming - avoid name conflicts and for clarity
6. Headers
   1. For standard library, use with “.h” extension and from namespace (ex: #include <iostream> and std::cout
   2. Have .h or .hpp extension
   3. Use **header guard** to avoid duplicate inclusion. #ifndef UNIQUE\_NAME #define …. #endif
   4. Don’t do definition in header file. Only declare.
   5. Avoid variables. Have only constants.
   6. Avoid relative path, include Search Path / Include Path (-I)
   7. Have same names for header files as source definition file. Ex: calculator.h, calculator.cc
7. Macros, Header Guards
   1. Avoid using for function or text substitution.
   2. Used mainly as header guards.
   3. Header guards are local!
8. Variables
   1. Prefer direct or uniform initialization.
   2. Define variables as to its usage as possible for code clarity.
9. Integer Type
   1. Preferable to use signed. Use unsigned cautiously. C++ freely converts between them.
   2. Use fixed width integer types. #include <cstdint> .. std::int16\_t val;
   3. Optionally use fast/least variants. Std::int\_fast16\_t or std::int\_least16\_t
10. Float
    1. Use ‘f’ suffix to make it explicit
    2. Use exponent representation. 9.234e-2 for 0.09234
    3. Careful about precision. Use double wherever possible.
    4. Common pitfall -> Rounding errors.
11. Const, Constexpr, Symbolic constants
    1. Don’t use magic numbers
    2. Prefer constexpr
    3. Prefer putting all constants as constexpr in a header under a namespace.
    4. Any variable that should not change values after initialization and whose initializer is known at compile-time should be declared as constexpr.
    5. Any variable that should not change values after initialization and whose initializer is not known at compile-time should be declared as const.
12. Operators
    1. Use parenthesis to make order of evaluation explicit.
    2. When using bitwise operators, always use “unsigned integers”.
13. Variable Scope
    1. Be careful about nested blocks and “**shadowing**” of variables! Do not use same names.
    2. Generally, use all CAPS or “g\_” for global variables and “s\_” for static. Avoid name conflicts with local and also clarify it is global
    3. non const global variables are evil.
    4. Always use namespace.
14. Type conversion
    1. It is always risk for numeric conversion (higher to lower). Data loss, rounding errors, loss of floating digits etc. are unpredictable.
    2. Avoid c-style cast as there is no compile time check. (float)I / J.
    3. Use static\_cast<float>(i). Provides compile time checks.
15. String
    1. If reading numeric values with std::cin, it’s a good idea to remove the extraneous newline using std::cin.ignore()
16. Enum
    1. Always use enum class instead of just enum. With scope operator, naming conflicts, comparison is handled correctly.
    2. Enum class color {RED, BLUE}; color red\_color = color::RED;
17. Typedef
    1. Use typedef to have meaningful type names. Example – typedef int error\_t; so that, error\_t read\_file();
    2. Use “\_t” for typedef name.
    3. Even better use “aliasing”. Using error\_t = int;
    4. Usually this is helpful to write platform independent code as well.
       1. Typedef char int8\_t;
18. Structs
    1. “struct”. Can have members. No memory is allocated when struct is defined. Memory is created when a variable of type struct is declare.
    2. Can have default values.
    3. Can use initialization like struct Employee{short id, short age, float salary}; Employee e1 = {1011, 28, 150000};
    4. Can have struct inside struct. Then initialization becomes {outer struct vars, {inner..}}
19. Control Flow
    1. Always write tests to cover EVERY execution path.
    2. Always prefer signed integer for looping conditions.
20. Input validation
    1. Use cin.fail(), cin.clear() and cin.ignore(32767, ‘\n’) to make programs robust.
21. String
    1. Don’t use <cstring> i.e., c-style string. i.e., using character array.
22. Pointers
    1. Always initialize to null ptr. Int \*ptr { nullptr }
    2. Pointer arithmetic can cause program crash if length of list is not handled correctly. Use caution.
    3. Cout print just address if int\*, it deciphers char\* as string. This may have inadvertent result in case char\* c = &another\_char;
    4. Be careful about return addresses on local variable from function. They become dangling ptrs.
    5. Use “p->member” instead of “(\*p).member”
    6. Use delete operators efficiently to avoid memory leaks.
    7. Gracefully handle memory allocation issue. int \*iPtr = new (std::nothrow) int;
    8. Delete is for 1 variable ptr, delete[] for array ptr. If you use delete for delete[] then memory leaks and crash happen.
    9. Avoid void ptrs unless absolutely necessary.
    10. Avoid pointer to a pointer to pointer… strategy unless absolutely necessary. Have extra precaution to delete[] these pointer references.
23. References
    1. Safer than pointers (requires initialization with valid address). Prefer this when no dynamic allocation is necessary.
24. Costly data copies
    1. Pass non-pointer, non-fundamental data type variables (such as structs) by (const) reference*.*
    2. In for-each loops element declarations, if your elements are non-fundamental types, use references or const references for performance reasons.
25. Static length list => Prefer to use std::array
26. Dynamic length list => Prefer to use std::vector
27. Functions
    1. Prefer pass by reference over pass by value or pass by address.
    2. Use const in function parameters if no change is intended.
    3. “Arrays get decayed to pointers”!!
    4. Careful when returning by reference. Don’t return by reference/address for variables on stack. Either use new or static or non-fundamental type variables.
    5. Be aware of inline functions, but modern compilers should inline functions for you as appropriate, so there isn’t a need to use the keyword.
    6. Function overloading – Use it to make things simpler. Naming consistency is more important!!
    7. Default params – Prefer in forward declaration / header.
    8. Prefer using c++11 function pointer style. #include <functional>; std::function<int(int, double)> fptr.
    9. Always try to avoid ellipsis (“…”) for variable inputs. It does not have type checks and length is important parameter to know.
28. Use “static\_assert(cond, msg)” instead of c style assert. All “static\_\*” are compile time!! i.e., value should be known, cannot have static\_assert on a variable.

**Object Oriented Programming – Best Practices**

1. Struct -> Data only DS. Class -> Data + operations
2. Class name should begin with Caps. Member variable names should preferably be m\_...
3. Ideally, member variables are private and member functions are public.
4. Only provide access functions when it makes sense for the user to be able to get or set a value directly.
5. Getters should usually return by value or const reference, not non-const reference
6. Use atleast one constructor. Even if it is just a default constructor.
7. Use member initializer lists to initialize your class member variables instead of assignment
8. Use constructor delegation rather than duplicating initialization. Donot call a constructor inside another, use member initialization way to call other constructors.
9. Use destructors whenever there is a dynamic memory allocation. Be careful, program crash or exit() will not call destructor resulting in memory leak.
10. Always prefer declaring class and its functions in header file and implementation separately in .cc file. .h and .cc file name ideally be same name as class.
11. Mark member functions that don’t modify variables as “const”. Show the intention.
12. Try to avoid usage of static variables and functions – shared and changes are not controlled.
13. Prefer regular function for overloading operators, over, friend functions. Less functions touching internals of a class is better.
14. A good class usually overloads arithmetic operators, << and >> input/output operators.
15. “explicit” Consider making your constructors and user-defined conversion member functions explicit to prevent implicit conversion errors
16. “delete”. Just making a function private does not fully guarantee protection on things you don’t want to do. For example, member functions, friend functions can still call private. Use “delete” when you absolutely want to disallow a behavior.
17. Be extra cautious about overloading assignment operator for cases – self assignment, dynamic allocated members. Use Deep Copy.

**Introduction-to-programming-languages**

1. Machine code / Instruction set: 0s and 1s. Specific to CPU architecture.

2. Assembly language: notation. Ex: mov al, 016h

Assembly -> Assembler -> Machine code

3. High level language: C/C++. Ex: a = 16;

Program -> Compiler -> Executable. (Efficient, compiler optimization, less flexible, C/C++)

Program -> Interpreter -> Execute. (Moderately efficient, less optimizations, more flexible, Python/JS)

**Introduction to C++**

1. History: C. 1972, Dennis Ritchie, Bell Lab. Objective-Cross hardware, high level, developer friendly, fine grained control on memory and resources.

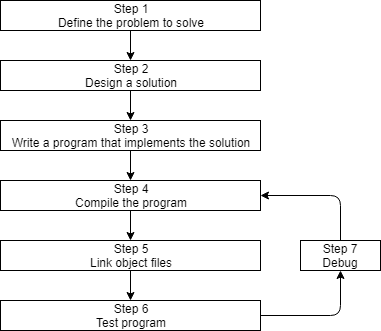
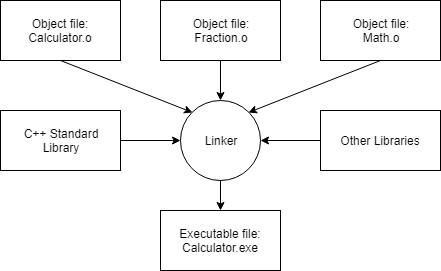
2. C++: Bjarne Stroustrup. Bell Labs. 1979. Most interesting extension - Object Oriented Programming Concept.

3. Philosophy - “Trust the developer”. You control. Hard but good. Programs that requires fine grained control - ML, DB, low latency programs, Embedded etc.

4. C++03 -> Standard published in 2003. C++11 -> Widely used. Published in 2011. Latest C++17 -> 2017

**Programming in C++**

1. Can have .cpp or .cc extension
2. Output of compiler is object file with .obj or .o extension
3. Linker links user programs, libraries used and creates one executable.
4. There is a standard library which is mostly used. Iostream is one functionality in standard library. Referred to as std::
5. Linkers automatically link standard libraries.
6. In large software system, people use “Make” or “CMake” to define how to compile and link CPP programs.

* Always define clearly **what** you want to solve.
* Have basic idea of **how**.
* Usually 20% on dev and 80% on other activites.
  + Document code
  + Simplified code
  + Modular and organized
  + Robust with edge cases

**Structure of a Program**

1. Things end with “;”
2. Statement => Expression (requires evaluation) => Function
3. Main is necessary. Have main.cc or project\_name.cc for file having main()
4. Preprocessors at the top. Special directive to compiler. #include makes it to include the “header file”.
5. Comments – Single line (//), multi line (/\*….\*/)
6. Variables – Define, Initialize, Assignment. There is no default initialization.
7. std::cout, std::endl. std::cout << x
8. std::cin >> x
9. Functions -> Reusable sequence of statements.
10. Functions -> Nested functions are not allowed in CPP. Default is pass by value.
11. Variables -> Cannot start with capital letters or numbers. Only “\_” is allowed.
12. Literals -> A value. X = 5; X is a variable, 5 is a literal. A literal evaluates to themselves.
13. Forward declaration / pure declarations. Functions and variables declared earlier and defined later.

**Functions, Namespace**

1. Each file is compiled independent of each other. Forward declaration or include header file is necessary.
2. Header files – Only declarations to avoid tedious forward declarations. Can have (.h, .hpp or no extensions). Declare once and use everywhere.
   1. Use angled brackets to include header files that come with the compiler. Use double quotes to include any other header files.
   2. Angular bracket tells it is a standard runtime library header.
   3. .h extension in standard library is for backward compatibility. Without extensions are new one. This happened when things got moved to std:: namespace.
   4. Use header guard to avoid duplicate inclusion. #ifndef UNIQUE\_NAME #define …. #endif
   5. Preprocessor just copies the content.
   6. Angular brackets (<…>) for standard libraries, quotes (“…”) for user defined.
3. Preprocessor
   1. Starts with #
   2. Macros and Includes
   3. Macros – Text substitution and function definition.
      1. Avoid both.
      2. Use mainly for header guards.
      3. Scope for that file only.
4. Fundamental Types
   1. Short, int, long, long long
   2. Float, double, long double
   3. Char, chart\_16t, char\_32t
   4. Bool
   5. Void
5. Initialization
   1. Copy Initialization: Ex: int val = 5;
   2. Direct Initialization: Ex: int val(5);
   3. Uniform initialization: Ex: int val{5};
      1. Prefered.
      2. Initializes to default if given empty. Ex: int val{}; //val => 0
   4. Prefer uniform initialization.
6. Integer:
   1. Signed or unsigned
   2. Signed preferred
   3. Size\_t => unsigned int => Type to represent size of data types. i.e., sizeof(int) return value is of type size\_t.
   4. Controversy – Size of int change based on the architecture.
   5. Solution – From C++ 11 – <cstdint>. Fixed width integers. Int8\_t, int16\_t, int32\_t, int64\_t, uint8\_t, uint16\_t, uint32\_t, uint64\_t
   6. But, slow, hence, some more types. fast/least variants. Std::int\_fast16\_t or std::int\_least16\_t
7. Floating:
   1. Large or small numbers. With decimal.
   2. Float, double, long double.
   3. Ex: float 123.33; float 123.33f; float 1.2333e2
   4. Default cout precision – 7.
   5. #include <iomanip> ; std::setprecision(17)
   6. 1.2 x 104, 1.2 is the significand and 4 is the exponent
   7. The digits in the significand (the part before the E) are called the **significant digits**. The number of significant digits defines a number’s **precision**. The more digits in the significand, the more precise a number is.
   8. INF -> Infinity
   9. NaN -> Not a number
8. Boolean:
   1. Int under the hood.
   2. True -> 1, false -> 0
   3. Cin silently fails if neither 0 or 1 is taken as input;
9. Char:
   1. 1 byte signed integer
   2. ASCII – 0 to 127. 0-32 unused. ‘a’ -> 97. ‘A’ -> 65
   3. Static\_cast<int>(var)
   4. Char16\_t, char32\_t => used for UNICODES. UTF-16 OR UTF-32
10. Const, constexpr, symbolic constant
    1. When you know initialization value – const. ex: const int pi(3.14159);
    2. When you don’t know initialization value – constexpr. Ex: constexpr int age = today – dob;
    3. Don’t do - #define MAX\_STUDENTS 30
    4. Prefer putting all constants in a header file as constexpr
11. Logical Operators:
    1. && (AND) has higher precedence than || (OR)
    2. !x && !y is not same as !(x && y)
12. Bitwise:
    1. Signed and unsigned
    2. 1st digit on left most used as signed bit.
    3. -5 in binary => 5 in binary (0000 0101). Invert and add 1. 1111 1010 + 1 => 1111 1011.
13. Scope, Duration, Linkage
    1. Scope -> where accessible, duration->where created/destroyed.
    2. Local variable, global variable,
    3. local hides global. Use :: scope operator to refer to global. Ex: ::value
    4. Linkage
       1. Internal => filescope => static const int a => const global by default are file scope;
       2. External => globalscope => extern int a => non const global by default are global.
14. Namespace
    1. Primarily to avoid collision
    2. Can have same namespace across multiple files.
    3. Do not create nested namespace
15. Casting
    1. C-style cast. (float) I / J; Should be avoided because there is no compile time check.
    2. Static cast. Use static\_cast<float>(i). Provides compile time checks.
16. String
    1. #include <string>
    2. To read from input stream. Std::getline(std::cin, my\_str);
    3. Be careful if mixing reading numeric with str. Std::cin >> my\_int; std::cin.ignore(32767, “\n”); // Ignore upto to 32767 characters till you read new line. Because, when taking a numeric input stream will be “4\n”. So after reading 4, ‘\n’ is still in the buffer, you need to ignore it.
17. Enum
    1. User defined type.
    2. Used with ‘enum’
    3. Usually use enum name prefix and all Caps
    4. You cannot have same name enum types in the same namescope.
       1. Enum Color{BLUE, RED}; Enum Feeling{HAPPY, BLUE}; // THIS IS NOT ALLOWED.
    5. Automatically, sets integer starting with 0s.
    6. Use enum class instead of enum. That avoids automatic conversion, naming conflicts, wrong comparisons etc.
18. Control Flow
    1. If, if..else, if..else if…else,
    2. Switch{case:..break;….default:…}
    3. While(cond){….cond\_modification}
    4. Do{…}(while(condition);
    5. For(init;condition;increment/decrement){..}
    6. Break and continue
    7. For each. For (auto &member : list)
19. Random number generation
    1. <cstdlib> = std::srand(seed); std::rand();
    2. Seed is usually used as time. <ctime> => static\_cast<unsigned int>(std::time(nullptr))
    3. <random> has many random generation algorithms. Std::uniform\_int\_distribution<>
20. Input validation
    1. If (cin.fail()) cin.clear(); cin.ignore(32767, ‘\n’)
21. Arrays
    1. Static array. Int array[10]; Length should be known at compile time.
    2. When passing to a function, address are copied. If you intended not to modify, make function param as const.
    3. #include <algorithm> has std::sort(array, last element address)
    4. #include <utility> has std::swap(x, y)
22. Pointers
    1. & -> Address of , \* -> Deferencing
    2. Always initialize to null ptr. Int \*ptr { nullptr }
    3. New, delete
    4. Use delete operators efficiently to avoid memory leaks.
    5. Gracefully handle memory allocation issue. int \*iPtr = new (std::nothrow) int;
    6. Static scope. (Program life time).
    7. Delete for 1 element pointer.
    8. Delete[] for array pointer.
    9. A const variable can have only const pointer.
    10. A const pointer can point to non const variable.
    11. Use “->” for deferencing.
23. Memory allocation
    1. Static, Automatic, Dynamic
    2. Static, automatic => Stack memory, limited (in MBs)
    3. Dynamic => Heap memory, large (in GBs)
24. References
    1. Keeping addresses of another variable. int value = 10; Int &ref = &value; ref = 11;
    2. Should be initialized with a valid reference.
    3. Cannot be used with dynamic memory allocation.
    4. Safer than pointers when there is no need for dynamic memory, but, only address passing.
    5. They do automatic deferencing i.e., ref = 10 will be changed to (\*ref) = 10
25. Array
    1. Std::array<int, 3> a = {1,2,3};
    2. #include <array>
    3. Length is required.
    4. Has a.begin(), a.end(), a.rbegin(), a.rend()
    5. Works with std::sort()
    6. Always pass by reference to function or in for each loop.
26. Vector
    1. Built for dynamic list usecases
    2. #include <vector>
    3. Std::vector<int> a = {1,2,3}.
    4. A.size(), a[0], a.at(2}
    5. A.resize(5)
    6. A.push\_back(..)
    7. A.pop\_back(..)
    8. A.reserve(..) is a good choice if you are planning with push\_back and pop\_back, so that dynamic memory allocation will be reduced.
27. Functions
    1. Large inputs / output – Prefer pass by reference or address.
    2. Pass by reference is widely used.
    3. You can return multiple values with a struct or tuple.
       1. #include <tuple>; auto[a, b] = std::make\_tuple(5, 5.5); std::tuple<int, double> t;
    4. Inline functions – Keyword “inline”. Modern compilers automatically do this better and also can ignore.
    5. Overloading
       1. By input params.
       2. Great tool for simplifications. Consistent naming is critical.
       3. Return values are not used
       4. Automatically promotes. Char->int, float->double, int->float etc.
       5. Preference for exact match
    6. Default params
       1. Right to left.
       2. Cannot define in both forward declaration and definition. Prefer declaration.
       3. Not considered in overloading.
    7. Function pointers
       1. Use #include <functional>; std::function<ret\_type(input\_t1, input\_t2)> fptr = func;
       2. Great use for callbacks.
    8. Data segments
       1. Code segment + data segment (static, global) + heap + stack.
       2. Creating memory on heap is slow.
    9. Error handling
       1. Validate inputs
       2. Have mechanism to return error code and take actions.
       3. Use static\_assert(cond, “error message”)
28. Command line inputs:
    1. Int Argc and char\* argv[]
    2. Argv[0] => program name
    3. Always string
    4. Use <sstream>; std::stringstream buf(argv[1]); int val; buf >> val;
    5. String stream is a string buffer from which we can extract value in to other type of data. Analogous to “cin”.
    6. Avoid taking variable input with ellipsis (…). Ellipsis use va\_list, va\_start(list, count), int var = va\_arg(list, int) and va\_end(list)

**Object Oriented Programming**

1. Representing properties + behavior.
2. Encapsulate, Abstract, Inherit, Polymorphism.
3. Access specifiers = Public, Private, Protected.
4. Private by default.
5. \*\* Access specifiers are per class basis not per object i.e., two objects of same type can access private members through member function.
6. Encapsulation
   1. Information hiding
   2. Keeping data and operation together
   3. Helps in
      1. Expose only what is needed.
      2. Easy to debug, change, reuse and standardize. Like an API
7. Constructors
   1. Same name as class name. No return types/values.
   2. No parameters => default constructor
   3. Member variable’s constructor are called before the enclosing class constructor.
   4. Don’t use constructor to re-initialize an already created object.
   5. Use initializer list in constructor rather than assignment. Construct(int val1, double val2) : m\_val1(val1), m\_val2(val2) {}
   6. Initializer list can initialize const vars in the class.
   7. Use constructor delegation to incrementally build constructors on top of other: A(int val1, int val2):m\_val1(val1), m\_val2(val2){} A(int val1):A(val1, 0){}
8. Destructors
   1. Same name with ~ prefix.
   2. No return types or input parameters.
   3. Typically for dynamically allocated class members.
9. This
   1. Usually hidden for developers. Pointer to the object.
   2. Obj.method(param1) is converted to obj.method(&obj, param1)
   3. In class. Void method(int param1) is converted to void method(const\* A this, int param1)
   4. Explicit use of this is useful in disambiguating param names and chaining calls. For chaining calls, return by reference for “this”. Calc& add(int val) {… return \*this; }
10. Classes and headers
    1. All functions defined inside a class are treated like inline. Defined outside are treated as regular functions.
    2. Suggested to declare in class, implement outside using “::” operator. Ex: void A::method()
    3. Have header file with class declaration and .cc file with implementation. Helps in managing like a library. (share .h and precompiled .cc file), less bloated code (not inline)
11. Const class objects and member functions
    1. You can mark a class object as constant. After this, you cannot modify any state of this object, not even public members.
    2. Mark member functions that don’t modify values as constant. Ex: void member(int val1) const {…}
    3. You can also overload function based on const ness.
12. Static Member Variables and Function
    1. Like static variables are global to a file, static variables in a function is global to the class.
    2. All objects of the class share the same variable.
    3. Typically used in usecases like unique ID per object, lookup table.
    4. There is static member function to access class’s static variables (public/private). Available across class. You call it with class classifier. ClassName::static\_function()
    5. There is no this pointer with static as expected hence, cannot access object member variables.
13. Friend Functions / Class / Class member functions
    1. Allowing an external party (function, another class, or a function in another class) to access private member in the master class.
    2. Master should say who is friend.
    3. Master cannot access friend. Friend has access to master.
    4. No access to this pointer of the master. Hence, to the friend, you need to pass object of master.
14. Operator Overloading
    1. Using “operartor+”, “operator-“ etc..
    2. Should be declared as a friend function.
    3. A + B => will invoke operator+(A, B)
    4. You can also overload as a normal function. Just declare the prototype in header file declaring class.
    5. Prefer regular function for overloading operators, over, friend functions. Less functions touching internals of a class is better.
    6. You can also overload cout and cin (<< and >>) operators. Std::ostream and std::istream are types for cout and cin.
    7. If you’re overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.
    8. If you’re overloading a unary operator, do so as a member function.
    9. If you’re overloading a binary operator that modifies its left operand (e.g. operator+=), do so as a member function if you can.
    10. If you’re overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function or friend function.
    11. Prefix and Postfix operators are distinguished by a dummy integer variable added by compiler. Ex: ++a => operator++(a); a++ => operator++(a, int)
    12. Overloading subscript operator (“[ ]”) is greatly useful. However, this does not work with pointers.
    13. Overloading () is super useful for more than one input operators, ex: matrix[0][4] cannot be written by overloaded []. However, you can do matrix(0,4) by overloading ( ) operator.
    14. Type operator can be overloaded. Operator int() () { .. }
    15. Copy Constructor
        1. Constructor that takes its own type.
        2. Makes member wise initialization by default.
        3. You can overload to modify this behavior.
        4. You can make it private to prevent your class from being copied.
        5. Fraction(const Fraction &fraction) { ..}
        6. Compiler by default adds this, if you don’t provide one.
    16. “explicit”
        1. Disallow any implicit conversion.
        2. Usually used with constructors
        3. Ex: A string class, having int constructor for the purpose of length. But, user using it with char literal. Use explicit on int constructor.
    17. “delete”
        1. Constructor(int) = delete. Explicitly mark it as deleted function.
    18. Assignment operator
        1. Be cautious on self assignments. Handle this case by checking if \*this == &param.
        2. Be careful on classes with dynamic memory allocation.
        3. You can also mark them as delete[].
15. Object relationship
    1. Composition: Is part of. Container manages the part. Part doesnot exist outside container. No part is member of two container. Ex: Int array with values. Human and heart.
    2. Aggregation: Has a. Container has the part. But, part will manage its own life time. Can be part of more than one container. Ex: Car and Engine. Ball and Color, Teacher and college.
    3. Association: Uses. One object uses other. Can be bi-directional. Life time is self-managed. Ex: doctor and patient.
16. Std::initializer\_list
    1. It is like a vector.
    2. Good practice to provide as constructor and also overload assignment operator.
17. Inheritance
    1. C++ prevents classes from initializing inherited member variables in the initialization list of a constructor. In other words, the value of a variable can only be set in an initialization list of a constructor belonging to the same class as the variable.
    2. Constructor base to derived
    3. Destructor derived to base
    4. Derived do not have access to base private
    5. Inheritance mean, base public things are available from the derived object. This is not something like friend function with access to private.
    6. Protected – derived class only not from outside.
    7. Public inheritance
       1. All access remains same. Preferred
    8. Private inheritance
       1. Through the derived object, base public/private becomes private
       2. Rarely used. Mostly used when through derived, you have to block base functionality.
    9. Protected inheritance
       1. Public becomes protected, rest remains same.
       2. Through derived, base public is now not accessible.
    10. You can change access specifier of base members (variables, functions) in the derived class.

private:

using Base::public\_member; // make base public member as private via derived object.

* 1. You can specifically ask compiler to use base class functions with :: operator.
     1. BaseClass::member() // will force compiler to to call base class’s version of the function.
  2. Member resolution will happen from the called object (derived?) and up the chain.
  3. You can inherit multiple class but avoid it.
     1. Creates problem in ambiguity. If both base class same member function or variable.
     2. Diamond problem: Multiple copies of Base in the most derived class.

1. Virtual Functions, Virtual Table
   1. **virtual function** is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class. This capability is known as **polymorphism**. A derived function is considered a match if it has the same signature (name, parameter types, and whether it is const) and return type as the base version of the function. Such functions are called **overrides**
   2. This is perhaps the biggest benefit of virtual functions -- the ability to structure your code in such a way that newly derived classes will automatically work with the old code without modification!
   3. To help address the issue of functions that are meant to be overrides but aren’t, C++11 introduced the **override specifier**. Override can be applied to any override function by placing the specifier in the same place const would go. If the function does not override a base class function, the compiler will flag the function as an error.
   4. If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class. These are called **covariant return types**.
   5. *Whenever you are dealing with inheritance, you should make any explicit destructors virtual.*
   6. Virtual Table
      1. Every class which has a virtual function or derived from class that has virtual function, will have a “vtable” created for it.
      2. vTable will have one entry for each virtual function.
      3. There will be 1 virtual table pointer (\*\_\_vptr) to point to VTable. This is the extra space burden with virtual functions.
      4. Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the \*\_\_vptr to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call.
   7. Pure Virtual function, Interface only Class, Virtual base class
      1. End function with “= 0”
      2. All functions in a class are pure virtual function, no member variables => Interface only class
      3. Virtual Base Class: Inherit as “class Derived: public virtual Base
         1. Helps avoid diamond problem.
         2. Most derived class should call constructor of virtual base class.
2. Object Slicing
   1. Note – Polymorphism works with reference and pointers. If you directly assigned a derived class to base class type, then only base part of the derived class will be copied. This is object slicing.
   2. Another pitfall is in vector. Like creating a vector of Base type.
   3. You should always use std::reference\_wrapper when dealing with vector of objects. Reference\_wrapper is in <functional>.
   4. Be careful about franken object! Using Base class pointer/reference to point to different derived objects. Only base portion will be copied during assignment.
3. Dynamic cast
   1. Dynamic\_cast<Dervied\* > d = base\_ptr;
   2. Base\_ptr first of all should be pointing to derived object.
   3. Returns null on cast failure.
4. Virtualizing << operator
   1. Because we typically implement operator<< as a friend, and friends aren’t considered member functions, a friend version of operator<< is ineligible to be virtualized.
   2. As usual create friend function for <<. And then, make it call a member function that is virtualized!