**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.
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..}}

**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.

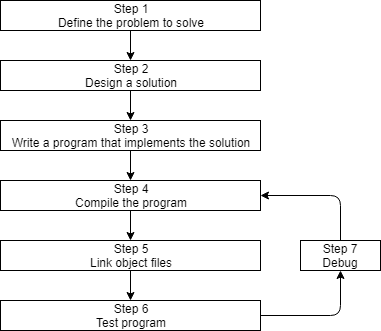
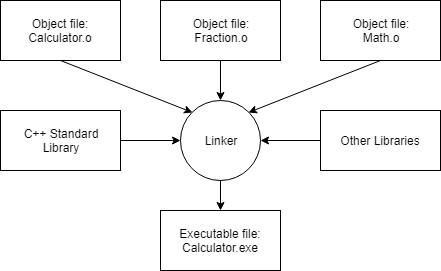
2. C++: Bjarne Stroustrup. Bell Labs. 1979. Most interesting extension - Object Oriented Programming Concept.

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.